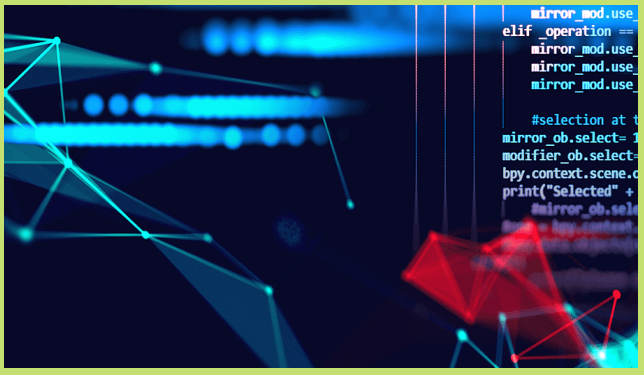

Shilpa C N
Dr. Lokesh Kumar

SOFTWARE DESIGN AND ARCHITECTURE

ALEXIS PRESS
JERSEY CITY, USA

SOFTWARE DESIGN AND ARCHITECTURE

SOFTWARE DESIGN AND ARCHITECTURE

Shilpa C N

Dr. Lokesh Kumar

ALEXIS PRESS

Published by: Alexis Press, LLC, Jersey City, USA
www.alexispress.us

© RESERVED

This book contains information obtained from highly regarded resources.
Copyright for individual contents remains with the authors.
A wide variety of references are listed. Reasonable efforts have been made
to publish reliable data and information, but the author and the publisher
cannot assume responsibility for the validity of
all materials or for the consequences of their use.

No part of this book may be reprinted, reproduced, transmitted,
or utilized in any form by any electronic, mechanical, or other means,
now known or hereinafter invented, including photocopying,
microfilming and recording, or any information storage or retrieval system,
without permission from the publishers.

For permission to photocopy or use material electronically
from this work please access alexispress.us

First Published 2022

A catalogue record for this publication is available from the British Library

Library of Congress Cataloguing in Publication Data

Includes bibliographical references and index.

Software Design and Architecture by *Shilpa C N, Dr. Lokesh Kumar*

ISBN 978-1-64532-882-7

CONTENTS

CHAPTER 1

BASICS OF SOFTWARE DESIGN AND ITS ARCHITECTURE

Shilpa C N, Associate Professor
Department of Computer Science and Engineering, Presidency University, Bangalore, India
Email Id-shilpa.cn@presidencyuniversity.in

ABSTRACT:

Since the inception of the discipline, software engineering research has been centered on software architecture. This essay examines the main characteristics of this research area and illustrates why design will remain a relevant area of study. Concept creation, the utilization of experience, and the ways of documenting, reasoning, and directing design work are some of the fundamental components of software design that are included in both process and product conversations. A variety of stakeholders are engaged in the design, which is portrayed as an ongoing way that lasts for the majority of a system's lifespan and involves a number of important decisions that form the architecture of the application. This essay will provide a complete outline of software engineering and its design in the future.

KEYWORDS:

Computer Science, Software Engineering, Software Design, SDLC, Computer Science.

INTRODUCTION

The core of software engineering is design. Design may be used as a verb or a noun. It is a crucial activity we carry out and output. Such direct assertions may also come out as apparent or even a little narrow-minded, if not inaccurate. Design is at the core of software engineering, which is a profession aimed at creating software systems, just as it is in any other productive endeavour, whether it be the construction of buildings, cars, toasters, or metropolitan areas. Therefore, it is not unexpected that during the last forty years or more, a large number of software engineering researchers or individuals familiar with software development have researched and written on software design. The development of great designers was included by Fred Brooks in his 1975 list of "promising assaults on the conceptual essence."

The key integrating activity that links the other activities in software engineering is design, which is applicable to all of them. The core emphasis of software engineering will always be design. Design is discussed in The Sciences of the Artificial in relation to "artificial" disciplines like software development. The interface between the inner and exterior worlds is the focus of the artificial world, which is focused with achieving objectives through modifying the former to the latter. The correct study for individuals who are interested in the artificial is how this adaptation of methods to surroundings is accomplished, with the design process at its core. In terms of software engineering, the inner environment refers to the collection of software languages, components, and tools we have available for system development, whereas the outside environment refers to the world of needs, objectives, and desires.As researchers in software engineering, we are always pushing the bar and building new tiers of infrastructure for future advancements. According to Simon, the inner environment, or the means, are always evolving and growing. However, as the floor rises, so do our goals and wants. Despite the improvements in design that have been made over the years, there will always be new problems to solve.

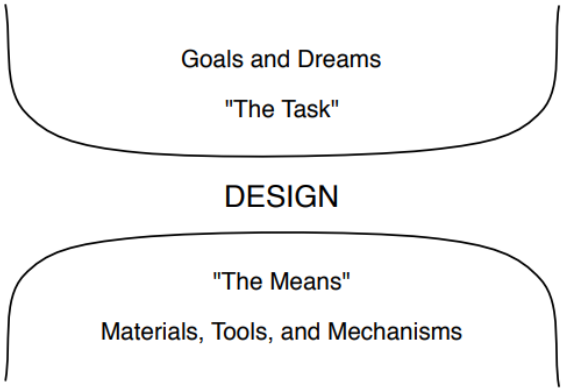

Figure 1: Illustrated that the Continuing Place of Design.

In Figure 1 shown the Continuing Place of Design. The obstacles for software design, however, remain the same at a sufficiently abstract level as they were forty years ago. They are the inherent difficulties in design: How to describe new notions, how to assess them, and how to design objects to achieve objectives. Twenty years after the first publication of the Mythical Man-Month, Brooks made this remark. It stated that the unique issues facing software engineering today are exactly those he had previously outlined, including "How to design and build a system out of a collection of programmes, How to design and build a programme or system into a robust, tested, documented, and supported product; and How to maintain intellectual control over complexity in large doses." That remark remains true ten years later. Possibly all of the main lines of study in software engineering are focused on enhancing our capacity to handle the difficulties of programme design. Work in requirements engineering contributes to Simon's "outer environment," while process research deals with the coordination of all activities aimed at developing, implementing, and iterating designs. Empirical studies enhance our capacity to evaluate design artefacts and the processes by which they were created. Analysis research enhances our capacity to evaluate potential designs.

Although design has been and will continue to be the main concern in software engineering, the kind of design on which our efforts have been concentrated has been somewhat uneven. Our attention has mostly been on software design in and of itself. That is, we concentrate on the features of software and its structure, such as taking into account which parts and connections make up a system and which restrictions control how they interact. The design of software, which shows characteristics to its users, has a smaller role in software engineering. What kind of "interactive vibe," for instance, does the programme provide its users? What kind of "style" is it? What distinguishing behavioural traits or branding does it have? Making the difference evident may be done by using the example of car design: The mechanical layout and structure of the vehicle are the focus of design research of the first kind; design research of the second kind focuses on modifying the car's look, performance, sound, and smell.

Despite the fact that they are inextricably linked, doing well in one area of design does not automatically translate into performing well in another. Both have significance and may legitimately be the focus of study into software design. Over the last several decades, work on the first type's design has undoubtedly produced a broad variety of significant outcomes. There have been various development methodologies advocated, many of which are based on the definition and usage of design "principles" such modularity and planning for change. Design representation methods, domain-specific methodologies, and auxiliary tools have all

been developed. Particular improvements have been achieved in recent years with respect to product families and the meticulous design of system architectures. The second form of design has often been neglected by software engineering scholars in favour of being left to either other computer science sub disciplines, particularly those studying human-computer interaction, or to working engineers in the private sector.

Both styles of design are becoming more and more valued as important business and societal assets. No matter which shore one is standing on, product design is seen as a task that cannot be successfully off-shored. The success of an organisation may depend on its unique design. Effective design skills are often the companion of innovation skills. The rest of this essay will examine where and how to go beyond the state-of-the-art. We then go on to look at some significant historical strands of design study before identifying a few noteworthy modern developments. These parts provide as much than just background information; they also directly imply several important paths for software development. We next look more closely at the nature of design using the viewpoints of these two parts as a guide. The rest of the article specifically outlines a number of avenues for more study before offering some difficulties and a "long perspective" of the bright future of software design.

A system's architecture, as seen in Figure 2, specifies its key elements, their connections (structures), and how they work together. There are several contributing variables to software architecture and design, including business strategy, quality characteristics, human dynamics, design, and IT environment.

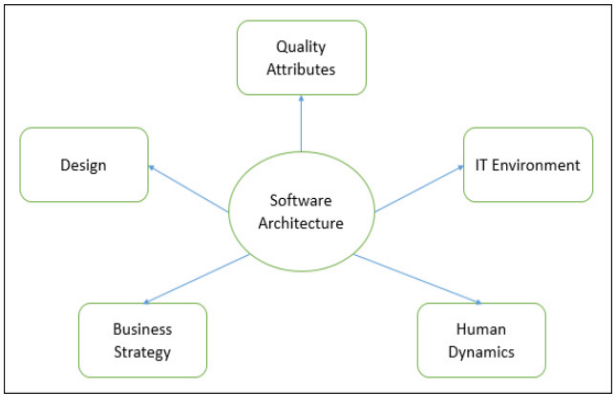

Figure 2: Reprinted that the Parts of Software Architecture.

Software Architecture and Software Design may be divided into two separate stages: Software Architecture and Software Design. The functional needs in architecture cast and separate nonfunctional considerations. Functional needs are fulfilled through design.

Software Architecture

A system's architecture acts as a blueprint. It establishes a communication and coordination mechanism among components and offers an abstraction to control the complexity of the system.

i. It establishes a disciplined approach to address all technological and operational needs while maintaining standard quality characteristics like performance and security.

ii. An addition, it entails a number of important organisational decisions connected to software development, and each of these choices may have a substantial influence on the end product's quality, maintainability, performance, and overall success. These selections include:

a) The choice of the structural components and their interconnections, which make up the system.

b) The conduct that is required by the interactions between those components.

c) Combining these behavioural and structural components into a sizable subsystem.

d) Architectural choices are in line with corporate goals.

e) The arrangement is guided by architectural styles.

Software Design

Software design offers a design strategy that details how a system's components fit and interact to meet the system's requirements. The following are the goals of having a design plan:

a) To discuss system specifications and establish expectations with clients, marketers, and management staff.

b) Serve as a development process blueprint.

c) Oversee the implementation duties, which include testing, integration, coding, and thorough design.

Goals of Architecture

Identification of requirements that have an impact on the application's structure is the architecture's main objective. A sound architecture lowers the business risks involved in developing a technological solution and creates a link between what is needed on the technical and business sides. The following are some of the additional objectives:

a) Make the system's framework public, but keep the specifics of its implementation hidden.

b) Recognize every circumstance and use-case.

c) Make an effort to meet the demands of diverse stakeholders.

d) Manage the demands of both functionality and quality.

e) Decrease the ownership objective while enhancing the organization's standing in the market.

f) Enhance the system's quality and usefulness.

g) Increase public trust in the system or organisation.

Role of Software Architect

A Software Architect provides a solution that the technical team can create and design for the entire application. A software architect should have expertise in the following areas –

Design Expertise

1. Expert in software design, including diverse methods and approaches such as object-oriented design, event-driven design, etc.

2. Lead the development team and coordinate the development efforts for the integrity of the design.
3. Should be able to review design proposals and trade-off among themselves.

Domain Expertise

1. Expert on the system being developed and plan for software evolution.
2. Assist in the requirement investigation process, assuring completeness and consistency.
3. Coordinate the definition of domain model for the system being developed.

Technology Expertise

1. Expert on available technologies that helps in the implementation of the system.
2. Coordinate the selection of programming language, framework, platforms, databases, etc.
3. Methodological Expertise
4. Expert on software development methodologies that may be adopted during SDLC (Software Development Life Cycle).
5. Choose the appropriate approaches for development that helps the entire team.

Hidden Role of Software Architect

1. Facilitates the technical work among team members and reinforcing the trust relationship in the team.
2. Information specialist who shares knowledge and has vast experience.
3. Protect the team members from external forces that would distract them and bring less value to the project.

Deliverables of the Architect

A. A clear, complete, consistent, and achievable set of functional goals
B. A functional description of the system, with at least two layers of decomposition
C. A concept for the system
D. A design in the form of the system, with at least two layers of decomposition
E. A notion of the timing, operator attributes, and the implementation and operation plans
F. A document or process which ensures functional decomposition is followed, and the form of interfaces is controlled

Quality Attributes

Quality is a measure of excellence or the state of being free from deficiencies or defects. Quality attributes are the system properties that are separate from the functionality of the system. Implementing quality attributes makes it easier to differentiate a good system from a

bad one. Attributes are overall factors that affect runtime behavior, system design, and user experience. They can be classified as:

i. Static Quality Attributes

Reflect the structure of a system and organization, directly related to architecture, design, and source code. They are invisible to end-user, but affect the development and maintenance cost, e.g.: modularity, testability, maintainability, etc.

ii. Dynamic Quality Attributes

Reflect the behavior of the system during its execution. They are directly related to system's architecture, design, source code, configuration, deployment parameters, environment, and platform. They are visible to the end-user and exist at runtime, e.g. throughput, robustness, scalability, etc.

iii. Quality Scenarios: Quality scenarios specify how to prevent a fault from becoming a failure. They can be divided into six parts based on their attribute specifications:

a. Source: An internal or external entity such as people, hardware, software, or physical infrastructure that generate the stimulus.

b. Stimulus: A condition that needs to be considered when it arrives on a system.

c. Environment: The stimulus occurs within certain conditions.

d. Artifact: A whole system or some part of it such as processors, communication channels, persistent storage, processes etc.

e. Response: An activity undertaken after the arrival of stimulus such as detect faults, recover from fault, disable event source etc.

f. Response measure: Should measure the occurred responses so that the requirements can be tested.

LITERATURE REVIEW

C. Venters et al. illustrated that the Context Modern societies are heavily dependent on intricate, massive, software-intensive systems that function in a climate of continuous availability and are difficult to maintain as well as adapt to as stakeholder objectives as well as system needs change over time. Software architectures serve as the building blocks of all software systems as well as a method for deliberating on essential factors for software quality. For software architecture study and practice, sustainability or the ability to persist in dynamic circumstances a crucial topic. Problem As a natural and progressive element of the entire software design and development process, accidental software complexity grows over time. From the standpoint of software architecture, this permits a number of problems to overlap, including, but not limited to: the absorption of technical debt design decisions made by specific components and systems resulting in coupling and cohesion issues; the application of tacit understanding and knowledge leading to unsystematic and unpublished design decisions; architectural knowledge vaporization of design choices and the continued ability of the organization to understand the architecture. To ensure efficient and effective management and evolutionary change, sustainable processing data must develop across the entire lifespan of the system, from the beginning of the original design to the end of the life. Method This article offers a foundation and vocabulary for framing the discussion on software architectures and viability by outlining broad perspectives and concepts on

sustainability with reference to software systems. It presents some of the most current research trends and techniques with respect to explicitly addressing sustainability in the area of software architectures, with a focus upon that ability of software architectures and architectural design decisions to persist through time [1].

M. Jaiswal discussed about the Software architecture is described as the strategic design of a task concerned with world needs, and its implementation, including programming paradigms, stunning architecture, component-based software engineering guidelines, architectural patterns, security, scalability, integration, and legal norms. Algorithms, design patterns, programming idioms, refactoring, and low-level implementation are just a few examples of the local requirements regulating what a solution accomplishes that are addressed in the process of functional design, also known as tactical design. In this essay, I'd want to discuss some ideas related to software architecture, software design, and their interactions [2].

Z. Dragičević et al. illustrated that the ability to adapt quickly to change and maintain fluidity are essential prerequisites for the next wave of IT technologies in the digital age. On the one hand, we are seeing a somewhat unfathomable increase in the use of technology in daily life, and on the other, software delivery is getting increasingly faster. This has raised expectations considerably and led to the adoption of agile practices and techniques, which has started to shift the pendulum of programming language from traditional to agile processes and strategies. Agile architecture is a new approach that uses agile practices to deliver a flexible architecture, adaptable to changing demands, and tolerant of changes, which is the result of the iterative development of the agile process of software development. Agile architecture is the result of the transformation of an adoption of agile approach to software development. Micro services have emerged as the outcome of a recent change in practice and research away from people and processes and towards integration technologies and application hosting, which has also raised interest in software design and structure. As a result, new methods for generating Agile architecture have emerged and developed. These methods, which fundamentally have similar objectives, include Continuous Architecting, Lean Architecting, and Evolutionary Architecting. In this regard, it is essential to investigate the origins of Agile architecture, as a unique approach in software development, to identify current policies and developments that are adapted to the modern digital environment. This will help us better understand the concept and indeed the new role of Agile architectural history in the digital era. With the aim of expanding the use of and improving the agile software development life cycle, researchers and practitioners will benefit from a better comprehension of Agile architecture, its operation, current trends, future development directions, and practices that are particularly beneficial for the creation of complex software [3].

S. Engelenburg et al. illustrated that the Governments must collect and interpret data gathered from businesses in order to protect public safety and security. Using this knowledge, one may assess if transported products have to be physically inspected since they may be suspect. Businesses are required to submit certain material, but many are hesitant to provide more information out of concern that doing so might expose them to liability and/or put them in violation of the law. In the creation of software architectures for knowledge transfer, these factors are often ignored. In the current study, we developed a software architecture for marketing information exchange using a design-science methodology. We identified the conditions that a sharing of information architecture should satisfy in order to be acceptable to companies based on the literature and a case study. The architecture was then created, and it was assessed against the criteria. The architecture comprises of a blockchain that controlled over a wide range and commercially managed information sharing regulations. Two parties

encrypt the Merkle root for every event using their private keys to verify that they are aware the data are accurate. This makes it simple to determine if information is correct and whether a situation merits acceptance. The context-based sharing of information is made possible through access control, metadata, and important details. This is used in conjunction with data encryption and decryption to do is provide access to certain data inside of an enterprise [4].

T. Sharma et al. illustrated that the architecture of a software system reflects the fundamental design choices, hence its quality is critical to maintaining the product's serviceability. Based on their granularity, extent, and relevance, code smells are characterized as indications of quality control problems in a software system. Despite the abundance of research on fragrances that has already been accomplished, there hasn't been a thorough examination of architectural smells, their traits, and their connections to smells at discrete intervals. The goal of this research is to analyze the features of architectural smells and the interactions between architecture and design smells in terms on correlation, collocation, and causation. We brought seven architectural smells into place to facilitate scent detection. We analyses the links between seven designs and 19 design smells using material from 3 073 open-source repositories that include more than 118 million lines of C# code. We discover that the size of something like the repository has no influence on scent density. Overall, there is a strong correlation between design odors and structural scents. The bulk of the architectural and architectural scent pairings do not show collocation, according to our evaluation of collocation. Our study of temporality concludes that design stinks because infrastructure stinks [5].

O. Bilal stated that the separating the data plane from the control plane, software-defined networking offers a variety of advantages for networking. Scalability, dependability, and availability of networks continue to be key problems. Network of interconnected architectures are crucial for SDN-enabled networks as a result. A thorough review of SDN involvement of different architectures is provided in this work. It introduces SDN and its primary deployment, OpenFlow. The contrasts between various forms of network of interconnected designs, including the distribution technique and the communications networks, are then thoroughly explained. Additionally, it offers examples of involvement of different architectures that have been deployed and are still being studied by outlining their design, communication method, and performance outcomes [6].

I. Akyildiz et al. illustrated that the designing flexible network architectures that can be implemented by the software defined communication paradigm is one of the major components and a significant problem for 5G cellular networks. Commercial cellular systems now in use depends on rigid hardware-based designs, both in the core network and at the radio frontend. The implementation as well as creation of new approaches to improve network capacity and, subsequently, coverage are severely hampered by these issues, which also impede the provision of really differentiating offerings that can adapt to changing, unequal, and highly unpredictable traffic patterns. This article introduces SoftAir, a brand-new software-defined architecture for fifth-generation (5G) cellular connections. To create a scalable, versatile, and robust network architecture, the unique principles of network function codification and software defined networking are specifically explored. More specifically, the fine-grained base station decomposition, seamless integration of Openflow, mobility-aware control traffic attempting to balance, resource efficient software defined networking, and distributed and collaborative traffic classification are discussed as essential enabling technologies for assisting and manage the proposed architecture. Furthermore, by offering software-defined traffic development services, the key characteristics of SoftAir architecture and its associated technologies are highlighted [7].

M. Raspopović et al. discussed the needs for successful teaching and learning approaches are developing and changing along with technology's continued development and evolution. This tendency has resulted in an increasing use of learning management systems (LMSs), often in positioned to capitalize on technology advancements. However, some LMSs are rather customizable and may limit users with their array of tools and features. In order to enhance the effectiveness of learning, new needs indicate the necessity for integration with external tools and systems. Technology used properly may be a tool to expand the opportunities for educational institutions. This research focuses on the cloud computing infrastructure and software architectural design for connecting institutional e-learning, educational management, and social learning environments. In addition to giving a summary of the variety of technologies and tools involved in the proposed design, this study suggests a software architecture that facilitates functions that encourage successful teaching and learning [8].

Y. Jararweh et al. illustrated that the Internet's present and future state are represented by the "internet of things" (IoT). Due to the enormous number of things linked to the Internet, there is a tremendous volume of data that needs to processed and transformed into valuable information. Additionally, in order to enhance and improve the performance of the IoT network, fresh solutions in the architecture and administration of the network are needed to organize and handle this vast amount of data. A new paradigm termed as "software defined systems" has appeared to conceal all complexity in conventional system design by abstracting all controls and performance management from the underlying devices and placing them behind an application level, or "software layer." By trying to combine the software defined network, software defined storage, and software defined security into a single application defined based modelling approach, a thorough software defined based framework model is proposed in this work to streamline the IoT performance management and offer a critical solution for the struggles in the traditional Iot ecosystem to forward, store, and secure the identify future research from the IoT objects [9].

N. Kaliyamurthy at al. illustrated that the old network design is being destroyed by software-defined networking, which focuses on its inadequacies from a narrow angle. Programming as well as networking were always seen to be two distinct fields, but thanks to SDN, they are now becoming more connected. This is an effort to address the problems currently plaguing the networking industry and to put forward functional, affordable, and successful alternatives. Considering the quantity of linked devices and the amount of data being kept collectively, changes to the current network architecture are unavoidable. Decoupled architecture and personalization inside the network are brought about by SDN, making administration, management, and troubleshooting simple. This study focuses on the software-defined networking, a developing network architecture. This work addresses various architectural perspectives, making it an intermediating work between the review of the research and implementation. It contributes to factors like design, programmability, security, security behavior patterns, and security lapses in contrast to a generic view on the evolving network, which renders the work as a review. This article also examines numerous weak places in the design and develops attack vectors for each plane, drawing a conclusion that will allow for future improvement in understanding the effects of the assaults and suggesting countermeasures[10].

DISCUSSION

Word software and architecture make up software architecture. We must first comprehend what these two sentences and architecture mean. As a result, we will comprehend the meaning of software architecture better. A software program, often known as software, is a set of

instructions that teaches the computer how to operate and carry out tasks or operations. The software depicts a physical image of a computer in the opposite manner to computer hardware. An application, script, or programming that runs on a device or machine may all be referred to as a "computer algorithm" using nonexclusive words.

All information processed by computer systems, processes, and data is referred to as computer software in computer science or software engineering. Application software, libraries, patches, and associated non-executable data, including online help or digital media, are all considered to be computer software. Software and hardware for computers are interdependent and cannot be operated independently. Architecture is a method, end result, or design that stems from the building of buildings. Obviously, architecture will not be used to construct buildings but rather to handle project management in the computer world. For a very long time, specialists that concentrate in software development solution architects have discussed and argued about the meaning of software architecture and design. Software architecture, according to some experts, is the most essential or foundational component of the software product now under development. Software architecture, according to some experts, is the manner that high-level components communicate with one another.

Software architecture is frequently referred to as a system's blueprint. It offers an abstraction for controlling system complexity and setting up coordination and communication channels between parts. As a result, the team will be given clear and accurate instructions in accordance with the goals to be met throughout the application development process. The following are issues linked to software architecture's role as a blueprint that we need to comprehend. The solution will be structured according to the blueprint's specifications in order to satisfy all operational and technical needs. Additionally, the blueprint must be able to enhance generic qualities like performance and security. Additionally, this blueprint will require a number of essential business decisions regarding the software development process, and each of these choices can have a big effect on the final product's quality, upkeep, performance, and overall success. How we address speed, high availability, scalability, and reliability of the application we design will depend on the architecture we choose.

CONCLUSION

One of the topics of this paper is that software design and architecture have been, and will continue to be, intellectually challenging tasks. As our capacity to successfully design for one set of problems increases, a new set of problems arises that need even greater advancements. The definition of software serves as the cycle's cap. Many fascinating chances for contributions from software design researchers arise when we broaden our understanding of what software is in a very liberal sense. For these opportunities, software designers have some clear advantages over designers from many other domains. Think about the challenge that car designers are now facing with interface design. One may argue that automakers are selling a driving experience rather than just sheet metal and rubber. A systematic combination of sights, sounds, sensations, and fragrances make up such an experience. Driving an automobile includes interacting not just with the controls within the car, but also with the occupants, with aural and visual inputs, with other cars, with the law, and with traffic control systems outside the car. What representation does that design have if the design challenge is to provide that interactive experience? Fundamentally, the relationship is intangible.

Traditional design disciplines, like car design, are deeply rooted in the tangible components of their classic products, and designers are not used to working with an ethereal end result. Software designers, on the other hand, have always worked with an immaterial product; we are used to developing, simulating, and evaluating a wider range of realities. Therefore, if we

expand the idea of software beyond the conventional parameters of the computer to include very unlike intangible goods, like this example car interaction, an exciting new world becomes possible. New types of intangible goods may be conceptualised, modelled, and constructed with the help of software designers. It is possible to develop new types of very complicated systems that are now almost feasible by collaborating with designers from various disciplines.

REFERENCES

[1] C. C. Venters *et al.*, "Software sustainability: Research and practice from a software architecture viewpoint", *J. Syst. Softw.*, vol 138, bll 174–188, Apr 2018, doi: 10.1016/j.jss.2017.12.026.

[2] M. Jaiswal, "Software Architecture and Software Design", *SSRN Electron. J.*, 2019, doi: 10.2139/ssrn.3772387.

[3] Z. Dragičević en S. Bošnjak, "Agile architecture in the digital era: Trends and practices", *Strateg. Manag.*, vol 24, no 2, bll 12–33, 2019, doi: 10.5937/StraMan1902011D.

[4] S. van Engelenburg, M. Janssen, en B. Klievink, "Design of a software architecture supporting business-to-government information sharing to improve public safety and security", *J. Intell. Inf. Syst.*, vol 52, no 3, bll 595–618, Jun 2019, doi: 10.1007/s10844-017-0478-z.

[5] T. Sharma, P. Singh, en D. Spinellis, "An empirical investigation on the relationship between design and architecture smells", *Empir. Softw. Eng.*, vol 25, no 5, bll 4020–4068, Sep 2020, doi: 10.1007/s10664-020-09847-2.

[6] O. Blial, M. Ben Mamoun, en R. Benaini, "An Overview on SDN Architectures with Multiple Controllers", *Journal of Computer Networks and Communications*. 2016. doi: 10.1155/2016/9396525.

[7] I. F. Akyildiz, P. Wang, en S.-C. Lin, "SoftAir: A software defined networking architecture for 5G wireless systems", *Comput. Networks*, vol 85, bll 1–18, Jul 2015, doi: 10.1016/j.comnet.2015.05.007.

[8] M. Raspopović, S. Cvetanović, D. Stanojević, en M. Opačić, "Software architecture for integration of institutional and social learning environments", *Sci. Comput. Program.*, vol 129, bll 92–102, Nov 2016, doi: 10.1016/j.scico.2016.07.001.

[9] Y. Jararweh, M. Al-Ayyoub, A. Darabseh, E. Benkhelifa, M. Vouk, en A. Rindos, "SDIoT: a software defined based internet of things framework", *J. Ambient Intell. Humaniz. Comput.*, vol 6, no 4, bll 453–461, Aug 2015, doi: 10.1007/s12652-015-0290-y.

[10] I. Omoronyia, U. Etuk, en P. Inglis, "A Privacy Awareness System for Software Design", *Int. J. Softw. Eng. Knowl. Eng.*, 2019, doi: 10.1142/S0218194019500499.

CHAPTER 2

SOFTWARE MODELING AND DESIGN METHODS

Mr. Manjunath KV, Assistant Professor
Department of Computer Science and Engineering, Presidency University, Bangalore, India
Email Id-manjunathkv@presidencyuniversity.in

ABSTRACT:

This chapter explains how to model and build various software architectures, such as object-oriented, client-server, service-oriented, segment, concurrent and real-time, and computing and software architectures, using UML-based techniques. The book outlines particular factors to be taken into consideration for each kind of software architecture and offers an integrated method to building software architecture. Future readers who seek to create agile methodologies using a structured UML-based methodology starting with requirements modelling using use cases, moving through static and dynamic modelling, and ending with application development based on architectural and engineering patterns will find this book helpful.

KEYWORDS:

Computer Science, Software Engineering, Software Design, Software Modeling.

INTRODUCTION

Programs were often executed in the 1960s with little or no systematic requirements analysis and design. Flowcharts in particular were often utilized, either as a tool for documentation or for preparing a thorough design before coding. Subroutines were first developed to enable the sharing of code by invoking it from many places within a program. They were quickly used as a project management tool and acknowledged as a way to build modular systems. A program might be broken down into modules, each of which could be created by a different individual and used as a subroutine or function. Top-down design and stepwise refinement became well-known as program design techniques with the rise of structured programming in the early 1970s, with the aim of delivering a methodical strategy for structured program creation. This was the first design approach to deal with the design of an operating system, which is a concurrent system.

Data flow-oriented design and data structured design are two distinct software design methodologies that rose to popularity in the middle to end of the 1970s. One of the earliest thorough and well-documented design methodologies to emerge was the data flow oriented-design methodology utilized in Structured Design. Consideration of the data flow through the system was supposed to help in understanding how the system worked. It offered a methodical process for creating data flow diagrams for systems and mapping those diagrams to structure charts. The coupling and cohesion criteria for assessing a design's quality were established by Structured Design.

This method placed a focus on the definition of module interfaces and functional decomposition into modules. The first phase of structured design, based on the production of data flow diagrams, was improved and expanded into a thorough analytical technique called structured analysis. Data-structured design was an alternate method of software development. According to this perspective, analysis of the data structures is the best way to get a thorough

grasp of the problem's structure. As a result, the focus is on developing the data structures first, followed by building the program structures based on the data structures.

In the field of databases, the idea of logical and physical data separation was crucial to the creation of database management systems. Early systems suffered greatly from the broad usage of global data, which made these systems prone to mistake and challenging to alter, even in many of those planned to be modular. A method for significantly decreasing, if not completely removing, global data was presented by information masking.

The development of the MASCOT notation and subsequently the MASCOT design process in the late 1970s made a significant contribution to the design of concurrent and real-time systems. MASCOT codified the way activities interact with one another, either via channels for message transmission or through pools, based on a data flow paradigm. A task can only directly access the data that a channel or pool maintains by invoking the access methods that they offer. In order to hide any synchronization problems from the caller task, the access methods also synchronize access to the data, generally via semaphores.

In the 1980s, a number of system design approaches were introduced along with a general maturing of software design methodologies. The system, which is structured as a network of concurrent tasks with each real-world entity represented by a concurrent task, is thought of as a simulation of the actual world. JSD also advocated a middle out behavioral approach to software design, which went against the then-prevailing notion of top down design. This method served as a paradigm for object interaction modelling, a crucial component of contemporary object-oriented programming.

Evolution of Object-Oriented Analysis and Design Methods

Several object-oriented design methodologies first appeared in the mid- to late 1980s as a result of the development and popularization of object-oriented programming. Modeling the issue area, informational hiding, and inheritance were the three core focuses of all these techniques. Parnas promoted the utilization of information concealing to create more self-contained modules that could be altered with little to no consequence on other modules. In the element design of Ada-based systems, Booch first introduced object-oriented concepts into design by using information beating. Later, he expanded this to encompass information hiding, classes, and inheritance in object-oriented design, and he also introduced object-oriented principles into analysis. In comparison to indeed, it is commonly believed that the object-oriented approach offers a more seamless transition from analysis to design.

Object-oriented analysis techniques combine object-oriented ideas into the software life cycle's analysis stage. The focus is on locating unassembled components in the area of the issue and translating them to application components. The first effort at object modelling was a static modelling method with roots in required element, more specifically entity-relationship (E-R) modelling or, more broadly, semantic data modelling, as used in database application architecture. The information-intensive items in the issue area that make up entities in E-R modelling. The focus is fully on data modelling, and the entities, characteristics of each entity, and interconnections among the entities are defined and shown on E-R diagrams. The E-R model is translated to something like a database during design, often a relational one.

In an object-oriented analysis, the issue domain's objects are discovered, represented as programming classes, and their properties as well as their connections to one another are established. Entity types in E-R modelling and classes in classical object-oriented modelling vary primarily in that the former have operations while the latter do not. Static object

modelling also models various problem domain classes, while information modelling solely depicts persistent entities that are to be kept in some kind of a database. Aggregation and generalization/specialization from advanced information modelling are also employed. Because it entails identifying the classes to which entities belong and displaying classes and their connections on class diagrams, static object modelling was also known as class building and object modelling. Static modelling of the issue domain is often referred to as domain modelling. Information shielding and inheritance were used in the early object-oriented analysis and design methodologies to stress the complex components of software development while ignoring the dynamic aspects.

The OMT made a significant contribution by highlighting how crucial dynamic modelling was. Along with establishing the object diagrams' static modelling language, OMT demonstrated why dynamic modelling might be carried out using state charts to illustrate the state-dependent behaviors of active objects and sequence diagrams to illustrate the order of interactions between them. Conditional probability diagrams are another tool used to describe active things. Booch first utilized object diagrams to illustrate instance-level interactions between objects. Later, to more clearly illustrate connection among objects, the interactions were sequentially numbered. Booch also invented the use case idea for replicating the functional needs of the system. Hansen also utilized the sequence diagram to explain the order in which the items involved in a use case interact with one another. All stages of Jacobson's object-oriented software engineering life cycle relied on the use case notion. The use case idea has had a significant influence on the creation of current object-oriented software.

Impact and Use

Our 3rd epiphany introduced a strong impact in current architecting practices as the value of capturing and using design decisions has already been reported in a few empirical studies that provide some results on the following aspects:

i. The perceived value of designs decisions and design rationale for different kinds of stakeholders, as different items for representing and recording the information of design decisions may not have the same importance for all stakeholders [17]. Hence, we should decide which type of knowledge would better fit each type of users.
ii. The effort in capturing decisions during the early development stages pays off in later maintenance and evolution phases, thus expecting return of the investment when decisions are captured for the first time [18]. The experiences described there also highlight the benefits of using specific tool support for capturing, managing, and documenting architectural design decisions.

A complementary perspective in which decisions are entangled with design for each architectural view, leads us to think about a “new” architectural view, called the “decision view”. This new perspective extends the traditional views by superimposing the design rationale that underlies and motivates the selection of concrete design options. Figure 1 depicts a graphical sketch of the decision view, in which design decisions are attached in the “4+1” view model.

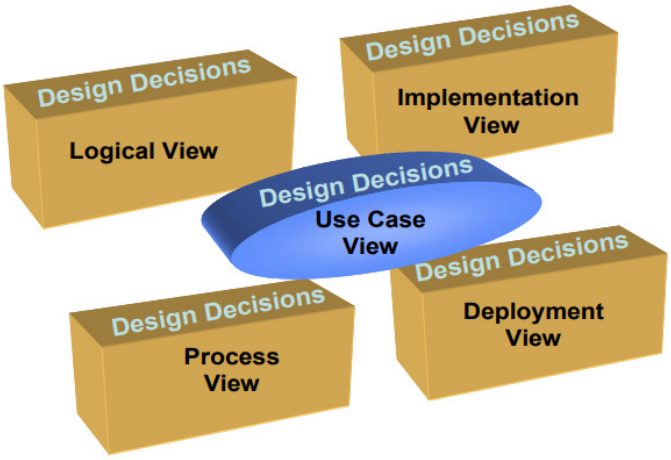

Figure 1: Represented that the "4+1" View Model.

Texture of a "Decision View"

The traditional representation of architectures in terms of views and viewpoints varies when decisions have to be described. Hence, architects interested in capturing the decisions and rationale should know how a decision view can be built, that is, the texture of decisions and how these are represented. As a first approach, we can refer to the classical methods for architectural assessment; most of them rely on the development of scenarios, their projection against several candidate architectures, and addition of information to the architectural components; later this information is aggregated and evaluated for each candidate architecture. We presented a case of architectural assessment and definition of design decisions on a product-line architecture for medical equipment where the decisions relate to the economic impact of changes of each architectural component. The authors focused on the economical attributes of each component in the implementation view (from the 4+1 model), and their decision view is composed by the decisions, rationale and actual data on the architectural components.

Focusing on the capture and representation of decisions itself, as a guide to help architects to document the decisions in their architectures, we propose the following actions:

i. Decide which information items are needed for each design decision such that name of the decision, description, rationale, pros and cons, status, category, etc. Then, decide which representation system will better ease the recording and organization of the decisions that is templates, ontologies. A strategy to capture the items such that codification, personalization, hybrid should be also selected.

ii. For each decision, define links to the requirements that motivate it,

iii. In case of alternative decisions that need to be evaluated, provide mechanisms to change the status of the decision such that approved, rejected, obsolete and category such that from alternative decision to a main decision,

iv. In case a decision may depend on previous ones, define these relationships in order to support internal traceability among them,

v. Once a set of significant set of decisions are made, link them to the architecture that results of such decisions. These links provide the connection to traditional architecture views,

vi. After all the decisions are made and captured, share them by means of communication and documentation mechanisms. Extra items and functionality can be added to this list, but we believe these are enough for a quick start in capturing design decisions and its underpinning rationale alongside with their architectures.

LITERATURE REVIEW

O. Bilal stated that the separating the data plane from the control plane, software-defined networking offers a variety of advantages for networking. Scalability, dependability, and availability of networks continue to be key problems. Network of interconnected architectures are crucial for SDN-enabled networks as a result. A thorough review of SDN involvement of different architectures is provided in this work. It introduces SDN and its primary deployment, OpenFlow. The contrasts between various forms of network of interconnected designs, including the distribution technique and the communications networks, are then thoroughly explained. Additionally, it offers examples of involvement of different architectures that have been deployed and are still being studied by outlining their design, communication method, and performance outcomes [1].

N. Kaliyamurthy at al. illustrated that the old network design is being destroyed by software-defined networking, which focuses on its inadequacies from a narrow angle. Programming as well as networking were always seen to be two distinct fields, but thanks to SDN, they are now becoming more connected. This is an effort to address the problems currently plaguing the networking industry and to put forward functional, affordable, and successful alternatives. Considering the quantity of linked devices and the amount of data being kept collectively, changes to the current network architecture are unavoidable. Decoupled architecture and personalization inside the network are brought about by SDN, making administration, management, and troubleshooting simple. This study focuses on the software-defined networking, a developing network architecture. This work addresses various architectural perspectives, making it an intermediating work between the review of the research and implementation. It contributes to factors like design, programmability, security, security behavior patterns, and security lapses in contrast to a generic view on the evolving network, which renders the work as a review. This article also examines numerous weak places in the design and develops attack vectors for each plane, drawing a conclusion that will allow for future improvement in understanding the effects of the assaults and suggesting countermeasures[2].

Z. Dragičević et al. illustrated that the ability to adapt quickly to change and maintain fluidity are essential prerequisites for the next wave of IT technologies in the digital age. On the one hand, we are seeing a somewhat unfathomable increase in the use of technology in daily life, and on the other, software delivery is getting increasingly faster. This has raised expectations considerably and led to the adoption of agile practices and techniques, which has started to shift the pendulum of programming language from traditional to agile processes and strategies. Agile architecture is a new approach that uses agile practices to deliver a flexible architecture, adaptable to changing demands, and tolerant of changes, which is the result of the iterative development of the agile process of software development. Agile architecture is the result of the transformation of an adoption of agile approach to software development. Micro services have emerged as the outcome of a recent change in practice and research away from people and processes and towards integration technologies and application hosting, which has also raised interest in software design and structure. As a result, new methods for generating Agile architecture have emerged and developed. These methods,

which fundamentally have similar objectives, include Continuous Architecting, Lean Architecting, and Evolutionary Architecting. In this regard, it is essential to investigate the origins of Agile architecture, as a unique approach in software development, to identify current policies and developments that are adapted to the modern digital environment. This will help us better understand the concept and indeed the new role of Agile architectural history in the digital era. With the aim of expanding the use of and improving the agile software development life cycle, researchers and practitioners will benefit from a better comprehension of Agile architecture, its operation, current trends, future development directions, and practises that are particularly beneficial for the creation of complex software [3].

M. Jaiswal discussed about the Software architecture is described as the strategic design of a task concerned with world needs, and its implementation, including programming paradigms, stunning architecture, component-based software engineering guidelines, architectural patterns, security, scalability, integration, and legal norms. Algorithms, design patterns, programming idioms, refactoring, and low-level implementation are just a few examples of the local requirements regulating what a solution accomplishes that are addressed in the process of functional design, also known as tactical design. In this essay, I'd want to discuss some ideas related to software architecture, software design, and their interactions [4].

D. Budgen discussed that the complexity and abstract character of software itself is a major contributor to the challenges involved in creating big software-based systems, and nowhere is this more apparent than in the difficulties experienced when attempting to build standardized processes for developing software. This article first looks at the characteristics of software and the design techniques that go into its creation, paying special attention to how software design methodologies attempt to systematize them. Then, we introduce the usage of what we have dubbed the D-matrix as a way of defining "software design models" and use it to investigate the shapes of the models that are created by following the steps of a variety of well-established software design methodologies. We evaluate these models and discuss the constraints on the practices that may be used in such procedures, as well as the degree to which more recently established design methods can reduce these consequences, before drawing a conclusion [5].

C. Venters et al. illustrated that the Context Modern societies are heavily dependent on intricate, massive, software-intensive systems that function in a climate of continuous availability and are difficult to maintain as well as adapt to as stakeholder objectives as well as system needs change over time. Software architectures serve as the building blocks of all software systems as well as a method for deliberating on essential factors for software quality. For software architecture study and practice, sustainability or the ability to persist in dynamic circumstances a crucial topic. Problem As a natural and progressive element of the entire software design and development process, accidental software complexity grows over time. From the standpoint of software architecture, this permits a number of problems to overlap, including, but not limited to: the absorption of technical debt design decisions made by specific components and systems resulting in coupling and cohesion issues; the application of tacit understanding and knowledge leading to unsystematic and unpublished design decisions; architectural knowledge vaporization of design choices and the continued ability of the organization to understand the architecture. To ensure efficient and effective management and evolutionary change, sustainable processing data must develop across the entire lifespan of the system, from the beginning of the original design to the end of the life. Method This article offers a foundation and vocabulary for framing the discussion on software architectures and viability by outlining broad perspectives and concepts on

sustainability with reference to software systems. It presents some of the most current research trends and techniques with respect to explicitly addressing sustainability in the area of software architectures, with a focus upon that ability of software architectures and architectural design decisions to persist through time [6].

I. Akyildiz et al. illustrated that the designing flexible network architectures that can be implemented by the software defined communication paradigm is one of the major components and a significant problem for 5G cellular networks. Commercial cellular systems now in use depends on rigid hardware-based designs, both in the core network and at the radio frontend. The implementation as well as creation of new approaches to improve network capacity and, subsequently, coverage are severely hampered by these issues, which also impede the provision of really differentiating offerings that can adapt to changing, unequal, and highly unpredictable traffic patterns. This article introduces SoftAir, a brand-new software-defined architecture for fifth-generation (5G) cellular connections. To create a scalable, versatile, and robust network architecture, the unique principles of network function codification and software defined networking are specifically explored. More specifically, the fine-grained base station decomposition, seamless integration of Openflow, mobility-aware control traffic attempting to balance, resource efficient software defined networking, and distributed and collaborative traffic classification are discussed as essential enabling technologies for assisting and manage the proposed architecture. Furthermore, by offering software-defined traffic development services, the key characteristics of SoftAir architecture and its associated technologies are highlighted [7].

T. Sharma et al. illustrated that the architecture of a software system reflects the fundamental design choices, hence its quality is critical to maintaining the product's serviceability. Based on their granularity, extent, and relevance, code smells are characterized as indications of quality control problems in a software system. Despite the abundance of research on fragrances that has already been accomplished, there hasn't been a thorough examination of architectural smells, their traits, and their connections to smells at discrete intervals. The goal of this research is to analyze the features of architectural smells and the interactions between architecture and design smells in terms on correlation, collocation, and causation. We brought seven architectural smells into place to facilitate scent detection. We analyses the links between seven designs and 19 design smells using material from 3 073 open-source repositories that include more than 118 million lines of C# code. We discover that the size of something like the repository has no influence on scent density. Overall, there is a strong correlation between design odors and structural scents. The bulk of the architectural and architectural scent pairings do not show collocation, according to our evaluation of collocation. Our study of temporality concludes that design stinks because infrastructure stinks [8].

L.Thomas et al. et al. illustrated that the common method for determining the size or density of biological populations is distance sampling. The program Distance is used in the vast majority of research and many distance sampling strategies. We briefly discuss distance sampling and its unfounded assumptions, provide an overview of Distance's background, structure, and capabilities, and offer some tips for using it. For accurate findings to be obtained, an effective questionnaire design is a vital need. Distance contains a survey design engine with a built-in geographic information system that enables the development of survey plans and the modelling of the attributes of various suggested designs. Modeling the likelihood of detection is the first stage in the data processing process from distance sampling. For this, Distance has three progressively more complex analysis engines: mark-recapture distance sampling, which loosens the assumption of certain detection at zero

distance, multiple-covariate path length sampling, which models detection probability as a function of distance from the transect, and segmentation techniques sampling, which represents detection probability as a function of distance first from transect and assumes all objects are detected. All three engines provide stratified estimate of density or availability, with related measures of accuracy determined analytically or through a bootstrap. The use of multipliers to enable analysis of indirect survey results, the density surface modelling analysis engine for geographical and habitat modelling, and relevant data about directly accessing the analysis engines from other software are some of the advanced analysis topics covered. Applicability and synthesis. A crucial methodology for estimating abundance and density in difficult field circumstances is distance sampling. To deal with realistic estimating scenarios, the theory underneath the approaches keeps growing. Modern software that implements these ideas and makes them available to working ecologists is explained in line with scientific advancements [9].

M. Raspopović et al. discussed the needs for successful teaching and learning approaches are developing and changing along with technology's continued development and evolution. This tendency has resulted in an increasing use of learning management systems (LMSs), often in positioned to capitalize on technology advancements. However, some LMSs are rather customizable and may limit users with their array of tools and features. In order to enhance the effectiveness of learning, new needs indicate the necessity for integration with external tools and systems. Technology used properly may be a tool to expand the opportunities for educational institutions. This research focuses on the cloud computing infrastructure and software architectural design for connecting institutional e-learning, educational management, and social learning environments. In addition to giving a summary of the variety of technologies and tools involved in the proposed design, this study suggests a software architecture that facilitates functions that encourage successful teaching and learning [10].

S. Engelenburg et al. illustrated that the Governments must collect and interpret data gathered from businesses in order to protect public safety and security. Using this knowledge, one may assess if transported products have to be physically inspected since they may be suspect. Businesses are required to submit certain material, but many are hesitant to provide more information out of concern that doing so might expose them to liability or put them in violation of the law. In the creation of software architectures for knowledge transfer, these factors are often ignored. In the current study, we developed a software architecture for marketing information exchange using a design-science methodology. We identified the conditions that a sharing of information architecture should satisfy in order to be acceptable to companies based on the literature and a case study. The architecture was then created, and it was assessed against the criteria. The architecture comprises of a blockchain that controlled over a wide range and commercially managed information sharing regulations. Two parties encrypt the Merkle root for every event using their private keys to verify that they are aware the data are accurate. This makes it simple to determine if information is correct and whether a situation merits acceptance. The context-based sharing of information is made possible through access control, metadata, and important details. This is used in conjunction with data encryption and decryption to do is provide access to certain data inside of an enterprise [11].

Y. Jararweh et al. illustrated that the Internet's present and future state are represented by the "internet of things" (IoT). Due to the enormous number of things linked to the Internet, there is a tremendous volume of data that needs to processed and transformed into valuable information. Additionally, in order to enhance and improve the performance of the IoT network, fresh solutions in the architecture and administration of the network are needed to

organize and handle this vast amount of data. A new paradigm termed as "software defined systems" has appeared to conceal all complexity in conventional system design by abstracting all controls and performance management from the underlying devices and placing them behind an application level, or "software layer." By trying to combine the software defined network, software defined storage, and software defined security into a single application defined based modelling approach, a thorough software defined based framework model is proposed in this work to streamline the IoT performance management and offer a critical solution for the struggles in the traditional Iot ecosystem to forward, store, and secure the identify future research from the IoT objects [12].

DISCUSSION

There are a great many definitions of the notion of a Software Architecture. Many of the early definitions focus on the end result of the architecture design process: the solution, the global design, represented as a set of components and connectors. Since about 10 years, the WHY of the solution: the set of design decisions, the rationale for these decisions, and the knowledge captured by those decisions, has become prominent, and to many replaced the solution-based definition of Software Architecture. The first such definitions appeared in 1992 and then in 2004. We may observe a steady increase in the number of papers that take architectural design decisions as starting point. A systematic literature review of the topic is mentions an increase from about 10 per year around 2005 to more than 25 in 2011. This naturally also brings us from the result of the architecting process, the set of design decisions, to the actual process of making these decisions. To be more precise about the flimsiness: could software architects possibly be biased or make decisions without careful considerations. A further question is the architects take decisions about. There is an ongoing debate about the intertwining between requirements and architecture. We may argue what these decisions are about are they about the solution the architecture, or about the problem the requirements and finally, architecting involves a number of decisions. This paper describes the historic evolution of software architecture representation, and the role it can play, through a set of epiphanies which guides the reader from the initial architecture views to a new “decision view”, expressing the need for capturing and using architectural design decisions and design rationale as first-class entities. We summarize new activities that arise during the architecting process when design decisions are explicitly recorded and documented, and illustrate how this architectural knowledge constitutes a new crosscutting view that overlaps the information described by other views.

CONCLUSION

Recent research has led to a new approach in the sense of how to communicate logic and creative and challenging, which is altering the perception the software architecture society views architectural design choices as an impairment loss that is challenging to document and convey. In order to prevent knowledge dissipation, the conventional gap between various software engineering outputs has shown the necessity to effectively and accurately record and reflect options available and the underlying logic for subsequent usage. We also consider it vital to record and retain important design choices, as opposed to the work needed to record and keep all the smaller decisions being made during the lifecycle of a software system. To avoid cutting and pasting efforts in recording design decisions and to facilitate the gradual launch of new activities dealing with design justification, some of which are associated to the distributed team decision-making process, methods like those mentioned in the sidebar need to be improved, adapted, or better integrated. We have noticed that there is a slight difference between the explicit expression of a design choice and form of online and the description of a well-known style sheet. Many times, a design choice is only a more refined expression of the

user demands without differing from the need that drove the choice. As a consequence, since users of the abovementioned tools often record the same data, the effort involved in collecting such judgments is seen to be duplicative. As a result, correct procedures should be offered to prevent the situation in which the same data is captured and to simplify the data gathering process. These processes will rely on improved duplicated detection and tracking methods. This same main objective of this study is to address the importance and influence of design justification in software architecture activities in particular and in software engineering in general. A better understanding of our software systems will be possible thanks to the inclusion of documented design decisions, despite the difficulties in capturing the design rationale. Software architects and developers will also see the value in treating decisions as first-class entities and working to improve integration with the other 12 software engineering artefacts.

REFERENCES

[1] O. Blial, M. Ben Mamoun, en R. Benaini, "An Overview on SDN Architectures with Multiple Controllers", *Journal of Computer Networks and Communications*. 2016. doi: 10.1155/2016/9396525.

[2] L. Li *et al.*, "An integrated hardware/software design methodology for signal processing systems", *J. Syst. Archit.*, 2019, doi: 10.1016/j.sysarc.2018.12.010.

[3] Z. Dragičević en S. Bošnjak, "Agile architecture in the digital era: Trends and practices", *Strateg. Manag.*, vol 24, no 2, bll 12–33, 2019, doi: 10.5937/StraMan1902011D.

[4] M. Jaiswal, "Software Architecture and Software Design", *SSRN Electron. J.*, 2019, doi: 10.2139/ssrn.3772387.

[5] D. Budgen, "'Design models' from software design methods", *Des. Stud.*, vol 16, no 3, bll 293–325, Jul 1995, doi: 10.1016/0142-694X(95)00001-8.

[6] C. C. Venters *et al.*, "Software sustainability: Research and practice from a software architecture viewpoint", *J. Syst. Softw.*, vol 138, bll 174–188, Apr 2018, doi: 10.1016/j.jss.2017.12.026.

[7] I. F. Akyildiz, P. Wang, en S.-C. Lin, "SoftAir: A software defined networking architecture for 5G wireless systems", *Comput. Networks*, vol 85, bll 1–18, Jul 2015, doi: 10.1016/j.comnet.2015.05.007.

[8] T. Sharma, P. Singh, en D. Spinellis, "An empirical investigation on the relationship between design and architecture smells", *Empir. Softw. Eng.*, vol 25, no 5, bll 4020–4068, Sep 2020, doi: 10.1007/s10664-020-09847-2.

[9] L. Thomas *et al.*, "Distance software: Design and analysis of distance sampling surveys for estimating population size", *Journal of Applied Ecology*. 2010. doi: 10.1111/j.1365-2664.2009.01737.x.

[10] M. Raspopović, S. Cvetanović, D. Stanojević, en M. Opačić, "Software architecture for integration of institutional and social learning environments", *Sci. Comput. Program.*, vol 129, bll 92–102, Nov 2016, doi: 10.1016/j.scico.2016.07.001.

[11] S. van Engelenburg, M. Janssen, en B. Klievink, "Design of a software architecture supporting business-to-government information sharing to improve public safety and security", *J. Intell. Inf. Syst.*, vol 52, no 3, bll 595–618, Jun 2019, doi: 10.1007/s10844-017-0478-z.

[12] Y. Jararweh, M. Al-Ayyoub, A. Darabseh, E. Benkhelifa, M. Vouk, en A. Rindos, "SDIoT: a software defined based internet of things framework", *J. Ambient Intell. Humaniz. Comput.*, vol 6, no 4, bll 453–461, Aug 2015, doi: 10.1007/s12652-015-0290-y.

CHAPTER 3

MODULES OF SOFTWARE DESIGN METHODOLOGY

Dr. Abdul Rahman, Professor
Department of Computer Science and Engineering, Presidency University, Bangalore, India
Email Id-abdul.rahman@presidcneyuniversity.in

ABSTRACT:

Everything that existing in any file system, both at design time and well beyond, is essentially a module. Packages, classes, and database tables are a few examples of modules. These module components serve as the foundation for the components and connection components. The work might become complicated without a modular procedure. A broad variety of stakeholders are said to collaborate in design, which spans the duration of a system's lifespan and requires making a number of critical decisions that comprise up the architecture of the application. In future this paper will be helps to another students and authors for their information and knowledge. This chapter will helps them in his/her study and research.

KEYWORDS:

Computer Science, Software Engineering, Software Design, SDLC, Waterfall Model.

INTRODUCTION

Many software development projects have been known to incur extensive and costly design errors. The most expansive errors are often introduced early in the development process. This underscores the need for better requirement definition and software de sign methodology. Software design is an important activity as it determines how the whole software development task would proceed including the system maintenance. The design of software is essentially a skill, but it usually requires a structure which will provide a guide or a methodology for this task. A methodology can be defined as the underlying principles and rules that govern a system. A method can be defined as a systematic procedure for a set of activities. Thus, from these definitions, a methodology will encompass the methods used within the methodology. Different methodologies can support work in different phases of the system life cycle, for example, planning, analysis, design and programming, testing and implementation. Svoboda (1990) developed the idea of a methodology further by proposing that there should be at least four components:

A. A conceptual model of constructs essential to the problem,
B. A set of procedure suggesting the direction and order to proceed,
C. A series of guidelines identifying things to be avoided, and
D. A collection of evaluation criteria for assessing the quality of the product.

The conceptual model is needed to direct or guide the designers to the relevant aspects of the system. The set of procedure provides the designer a systematic and logical set of activities to begin the design task. The evaluation criteria provide an objective measurement of the work done against some established standard or specifications.

A software design methodology can be structured as comprising of the software design process component and the software design representation or diagrammatic component. The process component is based on the basic principles established in the methodology while the representation component is the "blueprint" from which the code for the software will be built. It should be noted, that in practice, the design methodology is often constrained by existing hardware configuration, the implementation language, the existing file and data structures and the existing company practices, all of which would limit the solution space available to develop the software. The evolution of each software design needs to be meticulously recorded or diagramed, including the basis for choices made, for future walk-throughs and maintenance.

The Design Process

Design is a formation of a plan of activities to accomplish a recognized need. The need may be well defined or ill defined. When needs are ill-defined, it is likely due to the fact that neither the need nor problem has been identified. The design process is a process of creative invention and definition, it involves synthesis and analysis, and thus, is difficult to summarize in a simple design formula. Design is an applied science. In a software design problem, a number of solutions exist. The designer must plan and execute the design strategy taking into account certain established design practices. The designer often has to fall back on previous experience gained and has to study the existing software methodologies designed by others, to analyze their advantages and disadvantages. It is useful also to review the basic parameters, especially the requirements and system specifications.

A designer must constantly improve and enrich his store of design solution. The development of design alternatives should be a regular design activity aimed at seeing the most rational solution. In the design of software, often there are different design methodologies that can be used to derive a software solution, this is called the design degree of freedom. A design solution can also have several degrees of freedom, which implies that there is possibly at least one solution in the solution space, often there is more than one solution. It must be noted that there are no unique answers for design. However, that does not imply that any answer will do. Some solutions are more optimal than others. A design is subject to certain problem-solving constraints, for example, in a vacation design problem, the constraints are money and time. Note also that there are constraints on the solutions, for example, users must be satisfied[1].

Then in synthesis, it is necessary to begin with the solution of the main design problems and separate secondary items from the main ones. Successful design often begins with a clear definition of the design objectives. While in analysis and evaluation, the designer using these knowledge and simple diagrams, makes a first draft of the design in the form of diagrams. This has to be t thoroughly analyzed to verify that it meets the design objectives and that it is adequate but not over designed. A systematic approach is useful only to the extent that the designer is presented with a strategy that he can use as a base for planning the required design strategy for the problem at hand. A design problem is not a theoretical construct. Design has an authentic purpose - the creation of an end result by taking definite action or the creation of something having physical reality. The design process is by necessity an iterative one. The software design should describe a system that meet the requirements [2].

Most software engineers agreed that software design integrates and utilizes a great deal of the basics of computer science and information systems; unless one can apply the knowledge gained to the design of quality software, one cannot truly be considered to have mastered software design.

The Role of Design Methodology

The role of the software design methodology cannot be overemphasized. Software design methodology provides a logical and systematic means of proceeding with the design process as well as a set of guidelines for decision-making. The design methodology provides a sequence of activities, and often uses a set of notations or diagrams. The design methodology is especially important for large complex projects that involve programming-in-the-large; the use of a methodology establishes a set of common communication channels for translating design to codes and a set of common objectives. In addition, there must be an objective match between the overall character of the problem and the features of the solution approach, in order for a methodology to be efficient.

Design Phase in Software Systems Development

It has described the software development process as consisting of three broad generic phases the definition, development and maintenance phases. The definition phase defines the "what" of the software system, the development phase define the "how" and the maintenance phase defines the support and future necessary changes. Accordingly, software design is placed in the development phase of Software Life Cycle model. There are other models of the Software Development Life Cycle (SDLC) model. Almost every text on software development includes a SDLC model, there are some variations but, in general, the basic phases or activities are always present. The basic phases that are ever present are the analysis, design, testing, implementation and maintenance phases [3].

The analysis phase involves the requirement definition, from which the software specifications are derived. Design then translates the specification into a set of notations that represents the algorithms, data structures, architecture and interfaces. The representations are then coded and tested for defects in function, logic and implementation. When the software is ready, it is implemented and maintained by support personnel. While each of the analysis, design, testing, implementation and maintenance phases need to be performed in all cases, there are a number of ways their interactions can be organized. The major models that are in use are the Waterfall model, Prototyping Model and the Spiral model.

In all the different models, design plays a central role in the models and Software design is thus a major phase in the software system development. In this research project, the design process of the SDLC will be considered, which includes requirement definition or system analysis, system or requirement specifications, logical or system design, and detailed or program design and development. Even though pure software design consists of architectural and detailed design. Pure software design cannot proceed without the requirement definition and specification stages. Architectural design deals with the general structure of a software system. It involves identifying and decomposing the software into smaller models or components, determining data structures and specifying the relationship among the modules. According to a Software Engineering Institute report, detailed design involves the "formulation of blueprints for the particular solution and with modeling the detailed interaction between its components." It is concerned with the detailed "how to" for packaging all the components together[4].

Software Design Approaches

Software designers will group problems that have strong similarities while examining computer problems that need to be resolved. A problem domain is the name for this collection. A proposed application is a set of issues that each software design approach is best equipped to solve. As a result of the application domain's lack of definition, it is challenging

to compare one software design methodology to another. However, a categorization system will be useful when having specific approaches.

A few factors, such as the attributes of the systems to be built, the kind of software representation, and how formal it is, may be used to characterize techniques. The properties of the software structure, such as its level, hierarchy, and functional modularity, are supplementary sorts of criterion. The parameters of the design process are another important. The greatest categorization guide, nevertheless, is often the methodology's foundation. For instance, the three different perspectives of a system serve as the foundation of the current categorization approach for real-time systems published by the Software Engineering Institute [5]. You may sum it up like this:

A design technique's underlying perspective on the system, and hence the system that is depicted in a design based on that framework, might be functional, structural, or behavioral. According to the functional perspective, the system is seen as a series of components, each of which performs a distinct function and directly addresses a portion of the demand. Each development of body and how it interacts with the other components are described in the design. According to the structural approach, the system may be broken down into a variety of distinct components that can all be built, tested, and combined to form a functional whole. Every structural element ought to serve a functional purpose. The system is seen from a psychological perspective as an active entity that displays distinct behaviors, has internal state, changes state in response to stimuli, and produces consequences as a consequence of state changes.

Software design approaches may be characterized to aid in its generalization, explanation, and awareness as well as to aid in the choice of the best software design methodology to use. Software design methodology' specifics might differ substantially. Some are made up of a number of rules, while others are composed of stringent regulations and a synchronized set of diagrammatic representations. Diverse strategies have been put out for software design. Since they consist of a collection of strategies aimed towards each other and supporting a single, overarching rationale, several of these methodologies are really fully developed software design methodologies. The level-oriented, data-flow-oriented, data-structure-oriented, and object-oriented design methods are the basic design philosophies[6].

For various issues, several methods have been applied to produce software solutions. However, to provide a software solution for specific issues, many methodologies have been organically merged or blended. As a result, the various approaches to software design are not always complementary. For instance, designers have used the top-down decomposition lowering technique to divide a huge, complicated system into smaller, more approachable modules. The software for each module was then designed using a different approach. There are several methods to characterize software design techniques, guess it depends on the criteria being employed. The foundations of the approach, the features of the design process, and the kinds of software structures used to generate the solution are used to categorise software design methodology in this section.

Level-Oriented Design

Two generic or broad techniques may be applied in the level-oriented design approach. The first approach begins with a broad concept of the problem's solution before developing a specific solution step by step this is called Stepwise Refinement. This is essentially a top-down mechanism that is reliant on the system needs. The alternative strategy is to begin with a simple solution to the issue then, by modelling the issue, build up or expand the resolution by include more features.

Starting at the top level, the top-down method divides the system into smaller modular components via the use of implementation. Smaller modules are simpler to design, develop, and analyze. But the need of having a thorough grasp of the issue or system at hand is fundamental in the top-down approach. If not, it could need major revision in the future. Making judgements on the design structure early on affects the top-down approach as well. Varied design structures will be produced by different early choices. Each level is broken down into a more specific lower level in the repeated "break down" process of functional decomposition, known as stepwise refinement. In order to identify whether the atomic level has been reached and whether additional decomposition is required, a method must be available for each breakdown. There is no innate process or set of rules for this. If stepwise refinement is not carried out properly or "right," there is also a chance of duplication; this will happen at the conclusion of the process, that is, at the lower levels. The top-down technique is often employed in the first stages of the design process to break down the individual components or modules of a system since doing so may be expensive, particularly if there are many distinct designers or programming teams working on a single system. Other design techniques have also used the top-down method as a first step. The many designers or design teams may separate the system's modules after they have been established.

The design by composition technique entails building on the preceding stage's solution to evolve a new solution. This method allows for the inclusion of new features as the solution develops. The fundamental or original solution is used as the starting point for this technique, and additional modules are added or expanded via iterative composition. The outside-in model, in which the implementation of the system's internal functions is defined as the top-level decision and what the end users see as its external functions are defined as the lower-level decisions, is an example of a bottom-up design where the lowest level solution is developed first and gradually builds up to the highest level. This concept was developed to combat designers' propensity for giving end-user demands scant consideration. The inside-out paradigm, which is an alternative to the outside-in model, prioritizes internal system choices above exterior system functionality. Another model is based on the most-critical-component-first method, where the most limited system components are designed first in order to satisfy the critical parameters. The remaining system parts are then designed. Because model integration is often required in actual design efforts, these models are frequently conceptual and should not be strictly enforced.

Data Flow-Oriented Design

Information flow characteristics are used to determine software structure in the data flow-oriented design technique, also known as Structured Design. The processing or operations carried out on the da ta are the focal point of the data flow-oriented methodology. Information drives design. In the input-output stream, information may be seen as a continuous flow that is modified as it moves from node to node. A design model that employs a data flow diagram (DFD) may potentially be employed in the software development project since software is best represented by a DFD. When information is handled without a hierarchy, the data flow-oriented method is very useful. Transform analysis or transaction analysis are the two methods that may be used to map a DFD into the design structure. When the boundaries of the data flow in the input-output stream are distinct, transform analysis is used. Three fundamental modules input, process, and output are given control over the DFD when it is mapped into a structure. When a single piece of information causes flow to diverge along one of several avenues, transaction analysis is used. A substructure that obtains and assesses transactions is mapped to the DFD, while another substructure manages all data processing operations based on transactions. The Systematic Activity Modeling Method

(SAMM), Structured Design, and Structured Analysis and Design Technique (SADT) are a few illustrations of structured design or data flow-oriented design techniques.

Data Structure-Oriented Design

The data structures of input data, internal data for instance, databases, and output data are used by the data structure-oriented design methodology to create software. The focus is on the data structure, which is the object, in the data structure-oriented methodology. The so-called data structure of information has a significant influence on the sophistication and effectiveness of algorithms created to process information. The data architecture of the system has a tight relationship with both the design of the program; for instance, alternative knowledge will call for a conditional processing element, recurring data will call for a control flow statement feature, and hierarchical data will call for a structured software structure. Applications with a well-defined, hierarchical arrangement of information are the ideal candidates for knowledge structure-oriented design.

There are parallels across data flow and data structure orientated design techniques since they are both founded on factors in the information domain. Both rely on the carry out these activities to provide the groundwork for subsequent actions. Both undertake to convert information together into software structure and are information-driven. Data flow diagrams (DFD) are of limited use in data structure-oriented design, because translation and transaction processes are not taken into account. Information structures are expressed using hierarchical diagrams. Evaluation of the data structure's qualities, representation of the data in its most basic form, such as memorization, sequence, or selection, mapping of the data representation into a software control hierarchy, sophistication of the software control hierarchy, and development of a procedural explanation of the software, are among the tasks involved in data structure-oriented design. The Jackson System Development (JSD) and the Data Structured Systems Development are two instances of something like the data structure-oriented design methodology (DDSD).

Object Oriented Design

The use of the three software design principles of abstraction, information hiding, and modularity is unique to the object-oriented design methodology. In essence, objects are either consumers or producers of information. A private data structure and associated activities that might change the data structure make up the object. A message, or request to the object to carry out one of its operations, may call upon the procedural and control components that are included in operations. Additionally, the object has an interface where messages are sent to define the intended operation on the object. The object that receives the message will then choose how to carry out the specified action. Information concealment, or keeping implementation details concealed from all components outside the object, is accomplished by this method. Additionally, objects and their actions are by nature modular, which means that software components such as data and processes are grouped together via a clear interface mechanism.

LITERATURE REVIEW

T. Tang et al. stated that the effectiveness of the introduction of health information technology depends on user cooperation. It is still challenging to efficiently and meaningfully include clinician users in sophisticated healthcare organizations. In order to facilitate the care of complex hospitalized patients by an inter-professional team of clinicians, we developed and tested a clinical communication and platform that allows. The goal of this paper is to share our practical experience with using a variety of user participatory methods in this

process. Methods In a large community teaching hospital, we constructed and put into use an electronic platform for clinical interaction and collaboration. Both technological and medical experts made up the design team. In in order to encourage quick iterative design and user feedback, agile software development technique was adopted. Utilizing a range of user-centered, user co-design, and participatory conceptual design, we included clinical users at every level of the development lifecycle. Consumer engagement has improved user interface design, revealed software flaws, led to the development of various extensions that aided workflow, and early recognition of the need to alter the project's scope. A complicated health IT solution's design and execution benefited greatly from the complementing nature of a number of user participation techniques. These techniques may be used in collaboration with agile software development methodologies to translate hypotheses into operational healthcare systems that encourage iterative improvements [7].

L. Li et al. illustrated that the conception and implementation of signal processing applications on system-on-chip platforms are covered in this dissertation using a novel technique. The technique is based on the use of simple interfaces that allow applications to implement dataflow conceptual designs at various abstraction levels. Software implementation, implementation details, hardware-software co-design, and optimal application mapping are the production steps that are included in our methodology. The suggested technique simplifies the development of and integrate hardware and software subsystems for signal processing that use a wide range of programming languages and platforms. Researchers provide a dataflow-based deep neural network architecture for vehicle classification that is optimized for includes specialized on embedded system-on-chip devices as an implementation of the indicated design methodology. With both the help of the suggested technique, we design and incorporate a number of dataflow graph optimizations that are necessary for effectively transplanting the DNN system into a resource-limited architecture that makes use of cooperation multicore CPUs and practice area gate array subsystems. Through experiments, we show whether various design transformation may be implemented and connected across various sizes of the desired software system with flexibility and performance [1].

D. Parnas illustrated that the Similar to hardware families, programme families are groups of programmes with many shared characteristics that make it desirable to investigate these characteristics before examining specific members. The premise that, while constructing the first three approaches to the development, one should take the set as a whole if one intends to build a collection of related programmes over time is explored. Information concealing modules are refined and specified sequentially compared to the traditional method known as sequential development. The two approaches are then compared in further detail. It is shown via a number of instances that although the two approaches share similar principles, they provide complementing benefits [8].

S. Henery and D. Kafura discussed that the Software system production is guided in a disciplined and structured manner by structured design techniques. The technique does not, however, provide a clear, quantitative foundation for these judgments, even while it outlines and records the moments at during which they are made. The only rules typically provided to the designer are qualitative, often nebulous ones like utility, data transparency, or clarity. This study proposes and analyzes a set of software metrics suitable for assessing the topology of large-scale systems, similar like numerous previous studies. These measurements of knowledge transfer between system components serve as the foundation for any of these metrics. For process complexity, module complexity, and modular coupling, specific metrics are given. The evaluation, which uses the UNIX operating system's source code,

demonstrates that the complexity measurements are positively linked with the frequency of changes. Further, it is possible to evaluate the measurements for processes and subsystems to identify different kinds of development and design errors [9].

K. T. Al-Sarayreh et al. illustrated that the software process is a transitional stage for improving and enhancing the design for various sorts of software products and aids developers in translating the defined requirements into prototypes that actually carry out the design. The purpose of this article is to provide small business software developers, designers, and engineers a standards-based process improvement strategy that includes specific processes for incorporating small software systems into their organizations. The main output of this research is a precise step-by-step technique for improving the software design process that can be used by developers in small businesses to assess and create simple software at a low cost and in a short amount of time. In conclusion, this study offered a strategy to streamline the design process for various software systems utilizing a template-based approach to minimize the expense, time, and effort required during the implementation phase in small businesses. Since system and software engineering developers are typically in charge of analyzing, designing, implementing, and testing their software systems throughout the entire software life cycle, the scientific justification for a template-based implementation makes it easier for them to use this template in their small businesses[10].

R. Jolak et al. stated that the developing software is a communal and participatory endeavor. It takes a lot of work for software engineers to collaborate and share expertise. As a result, effective communication is critical for the success of a project. More in-depth empirical research on this occurrence is required to better understand the impact of communicating on project success. Our goal is to determine how employing a graphical design description vs a textual one affects communication about co-located software design. We looked at the effects of graphical vs textual design representations on the competence to Explain, Understand, Recall, and Actively Communicate information. We discovered that the graphical design description promotes Active Discussion amongst developers and enhances the Recall of design details better than the textual. Likewise, when utilized for the same length of time as its unmodified counterpart, a textual design explanation that is well-organized and prompted improves recollection of design specifics and boosts the number of active conversations at the expense of diminishing the satisfaction level of explaining.

H. Al-Matouq et al. discussed that one of the most crucial properties of high-quality software is protection. The goal of software security is to create safe software that prevents the integrity, confidentiality, and transparency of its code, data, or service from being compromised. Organizations often treat security as a convenience, which exposes them to security vulnerabilities. All stages of the Software Development Life Cycle must take security into considerations in order to create safe software (SDLC). Several strategies have been created to raise the quality of software, such Capability Maturity Model Integration (CMMI). However, problems surrounding software security have yet to be dealt with appropriately, and it is still challenging to integrate security procedures into the SDLC. In this article, a framework is created with the intention of helping software development firms build safe solutions more effectively. A Multifocal Literature Review (MLR) was carried out to find the good studies in both the formal and informal literature in order to accomplish this goal. The findings of the case study show that the Secure Software Design Maturity Model may be used to measure an organization's readiness for the secure design phase of the SDLC. Organizations may assess and enhance their software design security practices with the help of the Secure Software Design Maturity Model. It will also provide engineers a base upon which to build fresh software security features.

L. Carvajal et al. illustrated that the HCI community has proposed certain features for software applications during the last 20 years in order to deal with some of the most significant usability issues. However, for software engineers who have not acquired usability training, implementing such usability characteristics into computer programs may not be a simple procedure that is determining when, how, and why usability features should been considered. To assist computer programmers in include certain usability elements in their systems, we have created a set of usability principles for software development. In this study, we concentrate on the guidelines' artefacts for software design. We go into depth about the best design artefacts' structure and how to utilize them in compliance with each application's software architecture and software development process. Our proposals have been evaluated in a classroom context. The usage of the guidelines cuts development effort, enhances the caliber of the emerging designs, and greatly lowers the perceived sophistication of the usability characteristics from the developers' viewpoint, according to experimental validatio.

N. Nazar etb al. illustrated that the Common species issues in software architecture and design have conventional solutions referred to as computer design patterns. Design interpretation may be accelerated by being aware of the design patterns that a certain module uses. Researchers have advocated for automated design pattern identification strategies since manually identifying design patterns is laborious and time-consuming. For certain design patterns, these approaches function poorly, however. In this study, we provide a design pattern detection method (DPDF) that improves the state-of-the-art by automatically generating a design pattern detector utilizing code parameters and machine learning classifiers. Using the code features and call graph, DPDF generates a semantic information of the Java source code. The word-space structural analysis of the Java source code is then created by using the Word2Vec algorithm to the representation of knowledge. Then, using a machine learning classifier built by DPDF and trained on a labelled dataset, it is able to identify software design patterns with over 80% precision and over 79% recall. Additionally, we examined DPDF with Feature Maps and MARPLE-DPD, two currently used design image detection methods. Empirical findings show that our technique exceeds the methodological research in the area of precision by around 35% and 15%, respectively. The usefulness of our classifier in real-world applications is also confirmed by run-time results.E. Ozcan et al. ststed that one of the most important elements of a software system that connects system requirements and coding is system design. During the life of the software system, testing, maintenance, and further upgrades are all significantly influenced by the system design. All parties involved in the software system should be able to clearly and easily comprehend how the software design reflects all essential elements of the requirements. Separation of concerns in software design is advised to identify system components. In this regard, the system design papers do not now clearly represent the identification of the user tasks, i.e., the tasks that must be completed by the user. Our basic premise in this research is that by explicitly separating user tasks from those that must be completed by the computer system itself, software quality may be considerably enhanced. Additionally, what we suggest could better represent user needs and the system's primary goals in the software design, leading to an improvement in software quality. This study's primary goal is to provide a unique notation for software developers within the context of a UML Activity Diagram (UML-AD), which allows designers to recognize user activities and specify them independently of system duties. For this reason, the UML-ADE (UML-Activity Diagram Extended) version of the UML-AD was suggested. It was then put into practice in a scenario of a serious game where the description of user duties is crucial. The findings motivate researchers to create distinct design representations for task design in order to enhance system quality and to carry out more analyses of how these designs affect each of the aforementioned possible advantages for software systems.

DISCUSSION

Ideas and methods used throughout modern software design are still disorganized, odd, and poorly integrated. Despite detailed research, many experts still find it hard to understand, use, teach, and learn the concepts of software design. Although they may perform rather well, they are less acceptable cognitively and probably economically than the germination of seeds hypothesis of universal gravity.

We might very well solve some of our problems and using options theory for insights, as a framework for explanation, and possibly for decision-making, as well as by treating software design as determining how to make an irreversible capital outlay in risky software assets. Early data seem to have corroborated this notion. Without considering all of the probable consequences, it is still difficult to see the value of delaying design process, the distinction between vintage and non-legacy systems, and the rationality of using encrypted pictures in the face of change that is likely. Future research ought to concentrate on whether it is profitable to use mathematical option theory in conjunction and mathematical methodology to assist software design.

CONCLUSION

The design challenge develops beyond the computation's algorithms and information structures as software systems increase in complexity; it also includes creating and defining the platform's overall structure. The choice among both design alternatives is among the structural issues, together with the functional area and global control structure, protocols for correspondence, interconnection, and data access, functional appointment to design elements, physical distribution, composition of styling cues, scaling and acting skills. This level of design is for the software architecture. A sizable amount of work has been accomplished on this subject, including institutional arrangements of component interconnection networks, module interconnection languages, and templates and frameworks for systems that cater to the requirements of certain domains. However, there is an implicit volume of labor in the form of informal descriptive terminologies used to represent systems. Furthermore, although there isn't yet a standard nomenclature or notation to describe architectural structures, skilled software engineers often apply design techniques while creating intricate software. Many of the concepts are examples of adages or expressions that have developed across time informally. Others have more thorough documentation as professional and academic requirements. It has become more and more obvious that expertise in architectural software design is necessary for effective software engineering. First, it's critical to be able to find common paradigms in order to appreciate the high-level linkages between systems and to create new systems as modifications of existing ones. Second, selecting the proper architecture may have a devastating impact on the success of a software system design. Third, thorough knowledge of software architectures empowers the engineer to pick from design options based on moral principles. Fourth, it's often necessary to analyze and describe the high-level characteristics of a dynamic network using an architectural system representation.

REFERENCES

[1] L. Li *et al.*, “An integrated hardware/software design methodology for signal processing systems”, *J. Syst. Archit.*, 2019, doi: 10.1016/j.sysarc.2018.12.010.

[2] K. Kautz, “Investigating the design process: Participatory design in agile software development”, *Inf. Technol. People*, 2011, doi: 10.1108/09593841111158356.

[3] N. Cross, “Science and design methodology: A review”, *Research in Engineering Design*. 1993. doi: 10.1007/BF02032575.

[4] S. De en V. Vijayakumaran, "A Brief Study on Enhancing Quality of Enterprise Applications using Design Thinking", *Int. J. Educ. Manag. Eng.*, 2019, doi: 10.5815/ijeme.2019.05.04.

[5] I. Arab, S. Bourhnane, en F. Kafou, "Unifying Modeling Language-Merise Integration Approach for Software Design", *Int. J. Adv. Comput. Sci. Appl.*, vol 9, no 4, 2018, doi: 10.14569/IJACSA.2018.090402.

[6] C. W. Starr, E. R. Starr, en E. Worzala, "The impact of software company size and culture on commercial real estate location and design", *J. Corp. Real Estate*, 2019, doi: 10.1108/JCRE-11-2018-0043.

[7] T. Tang, M. E. Lim, E. Mansfield, A. McLachlan, en S. D. Quan, "Clinician user involvement in the real world: Designing an electronic tool to improve interprofessional communication and collaboration in a hospital setting", *Int. J. Med. Inform.*, vol 110, bll 90–97, Feb 2018, doi: 10.1016/j.ijmedinf.2017.11.011.

[8] D. L. Parnas, "On the Design and Development of Program Families", *IEEE Trans. Softw. Eng.*, vol SE-2, no 1, bll 1–9, Mrt 1976, doi: 10.1109/TSE.1976.233797.

[9] S. Henry en D. Kafura, "Software Structure Metrics Based on Information Flow", *IEEE Trans. Softw. Eng.*, vol SE-7, no 5, bll 510–518, Sep 1981, doi: 10.1109/TSE.1981.231113.

[10] H. F. Martins, A. C. de Oliveira, E. D. Canedo, R. A. D. Kosloski, R. Á. Paldês, en E. C. Oliveira, "Design thinking: Challenges for software requirements elicitation", *Inf.*, 2019, doi: 10.3390/info10120371.

CHAPTER 4

SOFTWARE ARCHITECTURE

Dr. Jayachandaran. A, Professor& HOD
Department of Computer Science and Engineering, Presidency University, Bangalore, India
Email Id-ajayachandaran@presidencyuniversity.in

ABSTRACT:

A system's overall system organization and behavior are expressed by the software structure of the system. Stakeholders may better understand and examine a system's architectural by looking at how essential characteristics like modifiability, availability, and security will be maintained. Software architecture is simply how a system should work. This structure encompasses all features, their interactions, the location in which they function, and the design ideas that guided the program. Throughout many instances, it may also reference to how the program will progress in the future.

KEYWORDS:

Computer Science, Software Engineering, Software Design, Software Architecture, Waterfall Model.

INTRODUCTION

A system's software architecture shows how the system is arranged or structured and explains its functionality. A system is the aggregate of elements that carry out a single function or a series of related functions. In other words, modern software architecture offers a reliable base for the creation of software. The system's overall success, quality, reliability, and maintainability are all influenced by a number of architectural choices and trade-offs. Your system may be at danger if common issues and long-term effects are really not taken into account. Modern systems frequently rely on a variety of high-level design patterns and ideas. Architectural styles are often used to describe them. Rarely is the architecture of a software system constrained to even one architectural design. Instead, so this whole system is often made up of a mix of styles [1].

The design challenge evolves beyond the computation's algorithms and structures of data as software systems grow in size and complexity; it also includes creating and defining the program's overall structure. The choice respectively design alternatives is one of the structural issues, along with the general organization and global control configuration, protocols for communication, connectivity, and data access, functional assignment to design elements, physical distribution, composition of design elements, ramping and performance. This level of design is for the software architecture. A sizable quantity of work has been done on this subject, incorporating formal models of component interconnection networks, module interconnection languages, and templates and frameworks for systems that cater to the demands about certain domains. Additionally, there is an implicit corpus of labor in the type of informal descriptive terminologies used to characterize systems. Furthermore, even though there isn't yet a standard nomenclature or language to describe building elements, skilled

software engineers often apply architectural concepts while creating intricate software. Many of the characteristics are examples of maxims or idioms that have developed through time informally. Others have more thorough documentation as professional and academic qualifications. It has become more and more obvious that expertise in architectural software design is necessary for efficient software engineering. First, it's important to be able to identify common assumptions in order to comprehend the high-level linkages between systems and to create new systems as alterations of existing ones [2].

The appropriate architecture may have a devastating impact on the achievement of a software system design. Third, thorough knowledge of software architectures enables the engineer to choose among design options in accordance with ethical principles. Fourth, it's often necessary to examine and describe the high-level characteristics of a complex system who used an architectural system representation. We provide a concise summary of software architecture in this essay. The purpose is to investigate how architectural design might affect software design and to demonstrate the status of both the field today. The information described here is taken from the authors' semester-long course, Optimization techniques for Software Systems, of the kind that they taught at CMU. Naturally, a short essay like this one can only skim the essential terrain characteristics. This choice emphasizes more on informal descriptions and leaves out a lot of the course's content on specification, assessment, and design alternative selection. Nevertheless, we think that this will assist to clarify the nature and importance of this burgeoning discipline. In the section that follows, we list many popular architectural trends that are presently the cornerstone of many systems and showcase how several trends may coexist seamlessly in a single design. Then, we demonstrate through the application of six case studies how architectural representations of software systems might enhance our knowledge of complicated systems. Finally, we review some of the current issues facing the field and discuss a number of the most exciting development research avenues [3].

Relation between Software Architecture and Software Design

i. Software architecture conceals the implementation specifics while exposing a system's structure. The interaction between the parts and aspects of a system is another area where architecture concentrates. Software design digs more deeply into the system's implementation specifics. The choice of data structures and algorithms, as well as the specifics of how each component is implemented, are all design considerations.

ii. Concerns in architecture and design often cross over. It makes more sense to mix architecture and design than to apply strict guidelines to separate the two. Certain choices are unmistakably more architectural in character. In other situations, choices greatly emphasise design and how it contributes to realising that architecture.

iii. It's crucial to remember that although all design is not architectural, architecture is design. In reality, the architect is the one who establishes the distinction between detailed design and software architecture (non-architectural design). Despite efforts to codify the difference, there are no laws or standards that apply in every situation.

iv. The way software architecture is going right now assumes that a system's design will change over time and that a software architect won't be able to completely architect it at first. Typically, while a system is being implemented, the design

changes. The software architect keeps up with new information and evaluates the design against needs from the actual world.

Architecture Analysis Solve the Problem

Software defects that lead to security problems come in two major flavours:

a) Bugs in the implementation and

b) Flaws in the design.

At least half of the whole software security issue is produced by implementation defects in the code. The second half concentrates on a different sort of software flaw that develops during the design stage. Design defects and bugs are split around 50/50. To ensure the confidentiality of your software, both must be protected. Even with the most powerful equipment available to mankind and the greatest code review systems in the entire world, it's improbable that you will be able to identify and correct errors in this approach:

a) Analyze fundamental design principles.

b) Assess the attack surface.

c) Enumerate various threat agents.

d) Identify weaknesses and gaps in security controls.

Finding and fixing technical errors early on in the development process is far more cost-effective than fixing incorrect design implementations after implementation. Design defects may be discovered and rectified using tools like architecture risk analysis (ARA), threat forecasting, and security control design analysis (SCDA). SCDAs are a simple implementation of ARA. Compared to typical ARA evaluations, they may be processed faster and by a much wider skill pool. Most significantly, a whole application spectrum may be covered by the lightweight methodology since it is so effective.

Organizations who wouldn't include architecture and design inspections into the project development are often shocked to discover that there software has systemic flaws both in the implementations and the design. Many times, the flaws found during vulnerability assessments might have been found more promptly and readily using other methodologies earlier in the life cycle. Testers often benefit more when they incorporate the findings of infrastructure analyses to inform their job.

LITERATURE REVIEW

H. Sun et al. illustrated that the engineering design methodologies often concentrate on the technology and struggle to effectively accommodate user actions in product design. Although both business and university agree that a product's ability to reach out to individuals is crucial to its success, there are few tools and methods available to designers to assist them in putting into consideration these aspects as they synthesizing their designs. This chapter discusses the creation of a behavioral design technique to aid designers in optimizing product performance all throughout design process by paying attention to usage requirements and situations. It also examines multi-trade engineering design. With this approach, design work combines user and product behavioral data. These two behaviors are characterized as tasks for the user and the product to do. So before adopting a final answer, a selection of solutions might be taken into account. In order to assist and enable a systematic implementation of the behavioral design method, a software program is being created and will be embedded into the designer's everyday activities. We provide an application to showcase the viability of our approach.

Required to comply with industry: In order to analyze and evaluate ergonomically in product or machine design, industrial engineers, designers, and ergonomists might utilize the behavior design methodology described in the chapter. The method has been built for practitioners of all ability levels who desire to improve the effectiveness of their products by combining user's behaviour and company behaviour all throughout design process [4].

C. Hofmeister et al. stated that the contrast five industrial software architecture engineering practices and draw a generic software architecture design methodology from their points in common. We evaluate the objects and activities the five methodologies employ or suggest using this broad framework, highlighting similarities as well as contrasts. The five methods have a significant amount in common and roughly follow the "ideal" pattern we established after we get beyond the vocabulary and description differences. We create an evaluating grid from the ideal arrangement that may be used to additional technique assessments [5].

V. Garousi et al. illustrated that with the rising complexity and scale of software systems, there is an ever-increasing demand for sophisticated and cost-effective software testing. To meet such a demand, there is a need for a highly-skilled software testing workforce in the industry. To address that need, many university educators worldwide have included software-testing education in their software engineering or computer science programs. Many papers have been published in the last three decades to share experiences from such undertakings. The main objective of this paper is to summarize the body of experience and knowledge in the area of software-testing education to benefit the readers in designing and delivering software-testing courses in university settings and to also conduct further education research in this area. This paper provides educators and researchers with a classification of existing studies within software-testing education. We further synthesize challenges and insights reported when teaching software testing. The paper also provides a reference to the vast body of knowledge and experience in teaching software testing. Our mapping study aims to help educators and researchers to identify the best practices in this area to effectively plan and deliver their software testing courses, or to conduct further education research in this important area [6].

M. Dadkhah et al. stated that software testing is the process of evaluating a software program to ensure that it performs its intended purpose. Software testing verifies the safety, reliability, and correct working of the software. The growing need for quality software makes software testing a crucial stage in Software Development Lifecycle. There are many methods of testing software, however, the choice of method to test a given software remains a major problem in software testing. Although, it is often impossible to find all errors in software, employing the right combination of methods will make software testing efficient and successful. Knowing these software testing methods is the key to making the right selection. This paper presents a comprehensive study of software testing methods. For each Testing Level and Testing Technique, examples of some testing types and their pros and cons were given with a brief explanation of some of the important testing types. Furthermore, a clear and distinguishable explanation of two confused and contradictory terms verification and validation, and how they relate to software quality was provided [7].

T. Maxime et al. stated that due to the poor adoption of the sound software test procedure, such as test automation, countless software projects in Cameroon and worldwide fail to provide acceptable quality output. In terms of addressing the basic concerns of what regional constraints exist to using sophisticated techniques to provide test cases and what challenges stand in the way of automating software testing, this report investigates software testing procedures in Cameroon. The main objective is to provide proposals on how to focus automated testing research to create solutions that encourage the implementation of effective

testing techniques in undertakings while being aware of the scarce human and financial resources available in emerging economies. To achieve this, research on businesses that specialize in activities of software development was conducted. The analysis of the results obtained reveals several interesting elements, among which over 80% of the respondents would not ensure that there has been a test other than that of the developer who does not follow a structured approach, automated tests constitute less than 8% of the number of assessments carried out, and the most main barriers to testing automation are the amount of time it takes to configure or adapt the toolkits, the costs of acquirement and integration, costs of implementation, and the moment required to develop this same unit tests[8].

A. K. Arumugam stated that the study to develop the software testing technique. A testing technique is a very difficult part of the SDLC. It is a slow rate and demanding procedure hence, improving the technique and advanced methodologies is necessary. This software testing technique is the automatic testing process and all testing processes are end to find the result and improve the present testing systems. The architecture used for software development and testing is still especially essential and is changing all the time. However, something as vital and significant as testing typically occurs rather late in the software development process. For deeper comprehension and early review, specification developers and testers should collaborate as much as necessary [9].

J. Ibrahim et al. illustrated the different growing trends of tools and methodologies of software testing available in the market. This software testing technique is the main focus of engineers, so any engineering project under development is a quality project. Testing is always a highly complex and time taking task. An automatic software testing technique is a time-saving and very high-speed process over manual testing [10].

R. Roshan et al. illustrated that the recent advances in the field of search-based software testing have taken on the name software evaluation methods. Improved dependability, as well as lessened software testing burden, are only two of the merits of search-based software testing. There are many ways to use search-based software evaluation methods, including WBT, GBT, and BBT. This paper also forecasts the considerable academic community's interest in this incredibly promising field of software testing, as seen by an increasing rise in the number of publications on search-based software testing [11].

S. Khan and R. Khan embellish that the software testing technique is a life cycle development in security testing. This analysis defined, procedures, implements, and techniques of system security and also include a lifetime phase for software safety. This technique is for security purposes. It also advises a mathematics method to calculate a test. This is used for security testing. The project development validates these protection mechanisms. The suggested work could assist testers in more effectively and efficiently understanding and carrying out conducted tests. [12].

P. Ajibade and S. Mutula discussed that the business records management is essential for allowing companies to uphold accountability, reduce corruption, and avoid mismanagement while encouraging effective decision-making and the proper functioning of everyday operations. Efficient leadership of both digitally and physically manufactured recorded business information is essential for large corporations. Records are produced as a result of contractual relationships in the global economy, in the digital and networked world. Although a large number of organizations handle network-generated corporate data and digital records, a large number of organizations have not been able to manage their business processes by using information technology infrastructure. In attempt to assist businesses, particularly small and medium enterprises (SMEs), manage their business analytics in a networked

environment, this chapter provided computer simulations employing service oriented architecture. Integrating and encouraging effective corporate records management while also taking into account the modernization of information technology. The usage of service-oriented design methods, if adopted by SMEs, would enhance company document management and sustainability, as well as compliance with records management laws, regulations, policies, and regulations, in accordance with the conclusion [13].

I. Arab et al. stated that the most important phase in the software development life cycle is software design, thus it must be managed carefully. To get at a good design that will enable for a subsequent development process to happen smoothly, software designers must go through several'd modernization. UML are the two primary modelling techniques that designers often must select between for this. Although both approaches are frequently utilized, they both have their own benefits and disadvantages. In order to formulate a methodology that would allow software designers to use both techniques to their full ability, this study incorporates both strategies and weighs their strengths. Although it may be especially applied to database design, this connection mostly concentrates on the entire software design process. It outlines the disadvantages and advantages of each of the two database modelling and design method UML. The conclusion of which of the two is appropriate at each stage of the simulation analysis is made based on a comparison of the UML diagrams that is shown later on in this essay [14].

A. Hamdy and M. Elsayed illustrated that the certain contexts, a design pattern is a sustainable response to a recurrent design issue. Software development that incorporates use of design patterns produces higher-quality, more comfortably products. However, choosing the appropriate design pattern to handle a design issue is a complex process for unskilled developers. The technique for autonomously selecting the most appropriate model structure from a set of patterns is presented in the paper. The suggested methodology is based on a text retrieval methodology, in which situations for engineering problems are expressed in common parlance. For the collection of design patterns, a vector space model (VSM) was constructed. For the specified design difficulty scenario, a word embedding and bigram-based feature vector is produced. The suggested design phase is the one that approaches the closest to the actual issue. The usefulness of the proposed methodology was shown by the experimental data after the possible mechanism was assessed using the Committee of Four design patterns [15].

DISCUSSION

Architectural styles are guiding concepts than mound a program. In terms of how a system is constructed, it is more like an abstract framework. The six main categories of architectural style are as follows:

i. Dataflow Architecture

All software systems can be categorized in this as lists of shifts on a collection of input data that is ordered historically, with data and operations being independent of one another. Data enters this structure and moves through each module individually until it is designated to a specific location. It is suited to applications requiring a sequence of integrated information calculations on clearly defined input and output and strives to attain the traits of reuse and unwavering quality. The data flow architecture employs three execution sequences between modules: Batch sequential, Pipe and filter or non-sequential pipeline mode, and Process control.

ii. Data Cantered Architecture

In this kind of organization, data is strategically placed and regularly accessible by other modules that alter data. The achievement of informational integrality is its fundamental goal. It consists of several parts that talk with one another through common data stores. The component are independent, meaning they only communicate via the data store, and each of them access a same data structure. The warehouse and the blackboard architectural styles are now the two sorts of data-centered architecture that are encapsulated by the flow of information. Most information methods incorporate this kind of design.

iii. Hierarchical Architecture

In this way, the whole computer is seen as a hierarchy where software systems are divided into components at various levels. It is generally used while creating system software, especially operating platforms and network protocols.

iv. Interaction Oriented Architecture

Separating customer interaction from post processing and business data processing is the prime purpose of interaction-oriented architecture. The system is divided into three main components the Data module, which provides data abstraction and all business logic; the Control module, which helps to identify the flow of the system's control and configuration actions; and the View introduction module, which is in charge of proffering data output visually or audibly. Model-View-Controller (MVC) and The Presentation-Abstraction-Control (PAC) are its two main styles.

v. Component Based Architecture

It is an organization that breaks down software designs in and out of useful units that each include their own methods, events, and variables. These elements become reusable and tenuously affiliated, resulting in modular applications that may be designed to meet any need.

vi. Distributed Architecture

This kind of architecture regulates or supports many more dispersed system components by being in the center of the system. Transparency, trustworthiness, and availability are its missions. It conceals how materials are accessed, variations in data platforms, resource placement, user-specific capabilities, failures, resource recovery, and a profusion of other things. Due apart from its compatibility with 21st-century advancing technology, it is the most frequently utilized kind of architecture. The emergence of the internet has dramatically helped software development. Today, software is networked, components are repurposed, concurrency is incorporated, and simultaneous data modifying changes are made. These are the distributed architecture's key advantages, which have contributed to its widespread use throughout the past.

vii. Broker Architecture

Primarily used as a technology bus to coordinate and facilitate communication between identified servers and clients.

viii. Client Server Architecture

Is commonly used by search engines, web servers, mail servers, it is mostly based on the functionality of the clients that is, requesting services of other components. The Service

Oriented Architecture is a major subdivision of this. It supports business-driven Information Technology (IT) approach in which an application consists of software services and software service consumers. It has the ability to develop new functions rapidly which makes it mostly used along with its basic features which would be explained in the next section.

CONCLUSION

An essential part of software engineering is software architecture and design. Both the infrastructure and design of software must always be taken into account for success in the area of software engineering. As a result, a number of fundamental issues relating to software design and structure have been examined. This essay's goal was to evaluate difficult subjects in software architecture and design. The collection of available papers and articles on the subject covered in the study was used as the analytic technique, and the percentage of each core subject were determined. Number of writers have discussed various aspects of software architecture and design, yet there aren't any set guidelines to adhere to that deal with all the problems in widespread software development. As has already been previously said, some components of software development should have been non-negotiable in order to maintain consistency and lessen the frequent system failures that were observed in the early stages of software development. This study emphasized the importance and meticulous labor that go into developing software and identified the key components that need to be thought about. It is crucial to remember that a critical and detailed study of software design and construction is necessary to prevent software from failing or crashing altogether throughout the software development process and to uncover any pertinent flaws in such strategies. Further research is required to investigate and assess ideas as they change in the ever-expanding sector of software architecture and design. The results of this study will be valuable to other academics who've been trying to learn more about software development and also who need to do more research on the largely unexplored subjects of software management, software evolution, and software development life cycle (SDLC).

REFERENCES

[1] J. Wen, H. M. Wang, S. Ying, Y. C. Ni, en T. Wang, “Toward a software architectural design approach for trusted software based on monitoring”, *Jisuanji Xuebao/Chinese J. Comput.*, 2010, doi: 10.3724/SP.J.1016.2010.02321.

[2] X. Xu, Q. Lu, Y. Liu, L. Zhu, H. Yao, en A. V. Vasilakos, “Designing blockchain-based applications a case study for imported product traceability”, *Futur. Gener. Comput. Syst.*, 2019, doi: 10.1016/j.future.2018.10.010.

[3] L. J. Osterweil, “Experience with an Approach to Comparing Software Design Methodologies”, *IEEE Trans. Softw. Eng.*, 1994, doi: 10.1109/32.286419.

[4] H. Sun, R. Houssin, M. Gardoni, en F. de Bauvrond, “Integration of user behaviour and product behaviour during the design phase: Software for behavioural design approach”, *Int. J. Ind. Ergon.*, vol 43, no 1, bll 100–114, Jan 2013, doi: 10.1016/j.ergon.2012.11.009.

[5] C. Hofmeister, P. Kruchten, R. L. Nord, H. Obbink, A. Ran, en P. America, “A general model of software architecture design derived from five industrial approaches”, *J. Syst. Softw.*, vol 80, no 1, bll 106–126, Jan 2007, doi: 10.1016/j.jss.2006.05.024.

[6] V. Garousi, A. Rainer, P. Lauvås, en A. Arcuri, “Software-testing education: A systematic literature mapping”, *J. Syst. Softw.*, vol 165, bl 110570, Jul 2020, doi: 10.1016/j.jss.2020.110570.

[7] M. Dadkhah, S. Araban, en S. Paydar, “A systematic literature review on semantic web enabled software testing”, *J. Syst. Softw.*, vol 162, bl 110485, Apr 2020, doi: 10.1016/j.jss.2019.110485.

[8] S. De en V. Vijayakumaran, “A Brief Study on Enhancing Quality of Enterprise Applications using Design Thinking”, *Int. J. Educ. Manag. Eng.*, 2019, doi: 10.5815/ijeme.2019.05.04.

[9] Arun Kumar Arumugam, “Software Testing Techniques New Trends”, *Int. J. Eng. Res.*, vol V8, no 12, Jan 2020, doi: 10.17577/IJERTV8IS120318.

[10] J. Ibrahim, S. Hanif, S. Shafiq, en S. Faroom, “Emerging Trends in Software Testing Tools & Methodologies: A Review Article in”, *Int. J. Comput. Sci. Inf. Secur.*, no December, 2019.

[11] R. Roshan, R. Porwal, en C. Mani Sharma, “Review of Search based Techniques in Software Testing”, *Int. J. Comput. Appl.*, vol 51, no 6, bll 42–45, Aug 2012, doi: 10.5120/8050-1387.

[12] I. Journal, M. Arif, A. Ahmad, en A. K. Arumugam, “IJERT-So ware Testing Techniques & New Trends Related papers Software Testing Techniques & New Trends”.

[13] P. Ajibade en S. M. Mutula, “Integrated Records Management: Using Software Design Approach to Support Business Process Management and Compliance in the Networked Environment”, *New Rev. Inf. Netw.*, vol 24, no 2, bll 178–192, Jul 2019, doi: 10.1080/13614576.2019.1618197.

[14] I. Arab, S. Bourhnane, en F. Kafou, “Unifying Modeling Language-Merise Integration Approach for Software Design”, *Int. J. Adv. Comput. Sci. Appl.*, vol 9, no 4, 2018, doi: 10.14569/IJACSA.2018.090402.

[15] A. Hamdy en M. Elsayed, “Automatic Recommendation of Software Design Patterns: Text Retrieval Approach”, *J. Softw.*, vol 13, no 4, bll 260–268, Apr 2018, doi: 10.17706/jsw.13.4.260-268.

CHAPTER 5

SOFTWARE ARCHITECTURE SYSTEM

Dr. Nagaraj S R, Associate Professor
Department of Computer Science and Engineering, Presidency University, Bangalore, India
Email Id-nagarajasr@presidencyuniversity.in

ABSTRACT:

The software architecture of a system represents the behavior and overall system architecture of the system. The design of the building of a system may be better understood and scrutinized by stakeholders by considering where key features like maintainability, dependability, and security will be handled. Software architecture is essentially the anticipated behavior of a system. All features, their interconnections, the context in which they operate, and the design principles that inspired the program are included in this hierarchy. In many cases, it may also mention the name of how the program will expand in the future.

KEYWORDS:

Computer Science, Software Engineering, Software Design, Software Architecture, Waterfall Model.

INTRODUCTION

It is becoming more and more obvious that the design of a software system strongly influences many of its key characteristics. Both functional and non-functional features are included in this group. Without an appropriate design, introducing such qualities into a system is at best very challenging and at worst impossible [1]. This implies that the architecture must be carefully designed to meet the needs of the system. There are several tools available to assist the software architect in this process, including the SAAM. But there are still a number of issues. One has to be able to analyse and debate many potential software architectures in order to make a sensible decision. Little more than rudimentary diagrams of boxes, lines, and notes comprise the conventional methods for describing architecture. These descriptions mainly depend on the participants' intuition and experience [2]. Although terminology like "client-server" or "layered" architectures might be used to describe architectural approaches, they still depend on human perception. The idea of architectural description languages is a step towards formalising the description of architectures. Many architectural description languages offer both textual and graphical notations. These languages may be textual or graphical. A language used to describe architecture has the following characteristics:

i. Composition

A system should be able to be characterized as a collection of unrelated parts and connections.

ii. Abstraction

A software architecture's elements and interactions should be able to be openly and concisely described in terms of their abstract functions in a system.

iii. Reusability

Even if they were created beyond the scope of the architectural system, components, interconnections, and architectural patterns should have been reusable in other architectural descriptions.

iv. Configuration

Independent of the items being regulated, architectural descriptions should specialize the description of system structure. Further, they must to accommodate dynamic configuration.

v. Heterogeneity

Different architectural principles must to be able to coexist in some kind of a single system.

vi. Analysis

It ought to be possible to do in-depth analysis of various kinds of architectural descriptions language (ADL). These are all good characteristics for an ADL, because they place more emphasis on the description than on the potential applications of the description. Analysis, the last point, is an exception. We offer three applications for interface definition languages that might be used to evaluate given architectural description against one or more system requirements on a machine-based or machine-supported basis [3]. An architecture should be assessed in light of the demands as early as practicable throughout development.

Building and maintaining the ability to add implementing components to the architectural description in order to create an implementation from the architecture. The characterization might serve as a reference document for the duration of the system's implementation and upkeep, providing a somewhat more precise structural characterization than the implementations source code. A description like that ought to make it virtually impossible for maintenance programmers to unwittingly violate the design. An ADL most likely has to be connected with the development environment for it to be helpful for building or maintenance. The potential for ornamental degradation is a challenging issue during upkeep. Maintenance programmers regularly make design choices that are at odds with the initial architecture and design when the rules change, features are added, and flaws are fixed. ADLs may assist in preventing erosion.

Common concepts of ADLs

This description should convey the architecture and its style while suppressing any low-level design decisions. Although the exact terms vary between ADLs, there does seem to be a core of concepts present in most:

i. Components:

Abstractions of subsystems within the architecture, such as filters in a pipes-and-filters architecture or layers in a layered architecture. Note that components are not necessarily atomic: most ADLs allow components to be composed of other components. Note also that since the architectural description is concerned with structure rather than functionality, only the externally visible properties of a component are present in the architectural description. If, for example, a component was implemented using a C++ class, only the public part of its interface would be present in the architectural description.

ii. Connectors:

Abstractions of the mechanisms for communication between components. Such connectors may describe ordinary function call, message passing, or communication using shared memory. They may also be used to describe concepts of architectural styles, such as pipes in a pipes-and-filters architecture. Although components can often be reasonably well described in implementation languages such as C++, Ada, or Java, these languages do not treat connectors as first class entities, but rather as properties of the components.

iii. Interfaces:

Abstractions of services provided by components or communicated by a connector. A component may implement an arbitrary number of interfaces and use an arbitrary number of interfaces of other components. To exemplify: it is not unreasonable to compare this set of concepts to traditional consumer electronics. If TV sets and VCRs are viewed as components, the cables can be considered to be connectors and the sockets to be interfaces. The power cable to the TV can be plugged into any power socket. Similarly, a component requiring a particular service can connect to any component that provides that service.

Fundamentals of Software Architecture

In the world of technology, starting from small children to young people and starting from young to old people everyone using their Smartphones, Laptops, Computers, PDAs etc to solve any simpler or complex task online by using some software programs, there everything looks very simple to user. Also that's the purpose of a good software to provide good quality of services in a user-friendly environment. There the overall abstraction of any software product makes it looks like simple and very easier for user to use. But in back if we will see building a complex software application includes complex processes which comprises of a number of elements of which coding is a small part [4].

After gathering of business requirement by a business analyst then developer team starts working on the Software Requirement Specification (SRS), sequentially it undergoes various steps like testing, acceptance, deployment, maintenance etc. Every software development process is carried out by following some sequential steps which comes under this Software Development Life Cycle (SDLC).

In the design phase of Software Development Life Cycle the software architecture is defined and documented. So in this article we will clearly discuss about one of significant element of Software Development Life Cycle (SDLC) i.e the Software Architecture.

Software Architecture

Software Architecture defines fundamental organization of a system and more simply defines a structured solution. It defines how components of a software system are assembled, their relationship and communication between them. It serves as a blueprint for software application and development basis for developer team. Software architecture defines a list of things which results in making many things easier in the software development process.

A. A software architecture defines structure of a system.

B. A software architecture defines behavior of a system.

C. A software architecture defines component relationship.

D. A software architecture defines communication structure.

E. A software architecture balances stakeholder's needs.

F. A software architecture influences team structure.

G. A software architecture focuses on significant elements.

H. A software architecture captures early design decisions.

Characteristics of Software Architecture:

Architects' separate architecture characteristics into broad categories depending upon operation, rarely appearing requirements, structure etc. Below some important characteristics which are commonly considered are explained.

Operational Architecture Characteristics

A. Availability

B. Performance

C. Reliability

D. Low fault tolerance

E. Scalability

Structural Architecture Characteristics

A. Configurability

B. Extensibility

C. Supportability

D. Portability

E. Maintainability

Cross-Cutting Architecture Characteristics

A. Accessibility

B. Security

C. Usability

D. Privacy

E. Feasibility

SOLID principles of Software Architecture:

Each character of the word SOLID defines one principle of software architecture. This SOLID principle is followed to avoid product strategy mistakes. A software architecture must adhere to SOLID principle to avoid any architectural or developmental failure. In Figure 1 shows the SOLID Principles of Software Architecture.

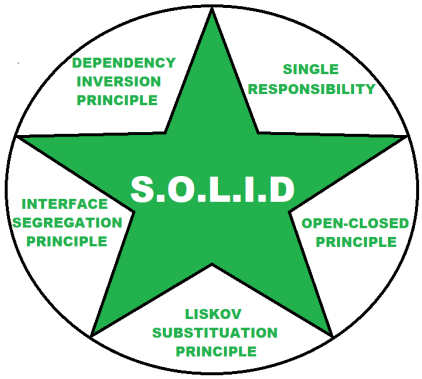

Figure 1: Represented that the SOLID Principles of Software Architecture.

Importance of Software Architecture:

Software architecture comes under design phase of software development life cycle. It is one of initial step of whole software development process. Without software architecture proceeding to software development is like building a house without designing architecture of house.

So, software architecture is one of important part of software application development. In technical and developmental aspects point of view below are reasons software architecture are important.

A. Selects quality attributes to be optimized for a system.

B. Facilitates early prototyping.

C. Allows to be built a system in component wise.

D. Helps in managing the changes in System.

Besides all these software architectures is also important for many other factors like quality of software, reliability of software, maintainability of software, Supportability of software and performance of software and so on.

Advantages of Software Architecture:

A. Provides a solid foundation for software project.

B. Helps in providing increased performance.

C. Reduces development cost.

Disadvantages of Software Architecture:

A. Sometimes getting good tools and standardization becomes a problem for software architecture.

B. Initial prediction of success of project based on architecture is not always possible.

From above it's clear how much important a software architecture for the development of a software application. So, a good software architecture is also responsible for delivering a good quality software product.

LITERATURE REVIEW

C. Venters et al. illustrated that the context Modern societies are heavily reliant on intricate, massive, software-intensive systems that function in a paradigm of continuous availability and are difficult to maintain and change to as stakeholder objectives and network needs change over time. Software architectures serve as the building blocks of all software systems as well as a method for deliberating on essential factors for software quality. For software architecture study and practice, sustainability or the ability to persist in changing environments is a crucial topic. Problem As a natural and progressive feature of the entire software design and development process, accidental software complexity increases with time. From the perspective of software architecture, the latter permits a number of issues to overlap, including, but not limited to: the accumulation of technical debt design decisions of individual components and systems resulting in coupling and cohesion issues; the application of tacit architectural knowledge leading to unsystematic and unrecognized design decisions; architectural knowledge vaporization of design choices and the continued ability of the organization to understand the architecture. To ensure efficient and effective maintenance and affecting the environment, sustainable software architectures must develop over the whole lifespan of the system, from the beginning of the original design to the end of the life. Method This article provides a context and vocabulary for framing the discussion on software applications and sustainability by outlining broad ideas and viewpoints on sustainability with regards to software systems. It presents some of the most present study trends and techniques with respect to explicitly addressing sustainability in the area of software architectures, with a focus on the ability of software applications and architectural design decisions to persist through time [5].

H. van Vliet et al. illustrated that the traditionally, software architecture is seen as the result of the software architecture design process, the solution, usually represented by a set of components and connectors. Recently, the why of the solution, the set of design decisions made by the software architect, is complementing or even replacing the solution-oriented definition of software architecture. This in turn leads to the study of the process of making these decisions. We outline some research directions that may help us understand and improve the software architecture design process [6].

N. Gavrilović and A. Mishra et al. illustrated that the Internet of things (IoT) enables organizations to automate the process and improves service delivery through Internet technology and transferring the data at the cloud level. IoT does not allow the use of a universal software architecture for different fields in which it is used, but needs to be adjusted according to the requirements of users. This paper presents an analysis of currently available types of software architectures of the IoT systems in the field of smart cities, healthcare, and agriculture. It provides a proposal for solutions and improvements of different software architecture types, interactions between identified software architecture elements that will provide better performance and simplicity. The novelty of the study is the analysis of different types of IoT software architecture such as: layered, service-oriented and cloud-based software architecture application in these areas of IoT. Based on the analysis, the study proposed the type of software architecture of the IoT system for the relevant area of application (smart city, healthcare, and agriculture). Specific points of research are: analysis of different types of software architecture applied in IoT systems, identification of functionalities available in IoT systems through different types of software architecture, the proposal for enhancement of the above functionalities, and proposal of software architecture that is most relevant to the IoT system of a particular area[7].

M. Razavian et al. illustrated that the Despite past empirical research in software architecture decision making, we have not yet systematically studied how to perform such empirical research. Software architecture decision making involves humans, their behavioral issues and practice. As such, research on decision making needs to involve not only engineering but also social science research methods. This paper studies empirical research on software architecture decision making. We want to understand what research methods have been used to study human decision making in software architecture. Further, we want to provide guidance for future studies. Method: We analyzed research papers on software architecture decision making. We classified the papers according to different sub-dimensions of empirical research design like research logic, research purpose, research methodology and process. We introduce the study focus matrix and the research cycle to capture the focus and the goals of a software architecture decision making study. We identify gaps in current software architecture decision making research according to the classification and discuss open research issues inspired by social science research. We show the variety of research designs and identify gaps with respect to focus and goals. Few papers study decision making behavior in software architecture design. Also these researchers study mostly the process and much less the outcome and the factors influencing decision making. Furthermore, there is a lack of improvements for software architecture decision making and in particular insights into behavior have not led to new practices. The study focus matrix and the research cycle are two new instruments for researchers to position their research clearly. This paper provides a retrospective for the community and an entry point for new researchers to design empirical studies that embrace the human role in software architecture decision making [8].

D. Sobhy et al. stated that the Evaluating software architectures in uncertain environments raises new challenges, which require continuous approaches. We define continuous evaluation as multiple evaluations of the software architecture that begins at the early stages of the development and is periodically and repeatedly performed throughout the lifetime of the software system. Numerous approaches have been developed for continuous evaluation; to handle dynamics and uncertainties at run-time, over the past years, these approaches are still very few, limited, and lack maturity. This review surveys efforts on architecture evaluation and provides a unified terminology and perspective on the subject. We examined each approach and provided a classification framework for this field. We present an analysis of the results and provide insights regarding open challenges. Major results and conclusions: The survey reveals that most of the existing architecture evaluation approaches typically lack an explicit linkage between design-time and run-time. Additionally, there is a general lack of systematic approaches on how continuous architecture evaluation can be realized or conducted. To remedy this lack, we present a set of necessary requirements for continuous evaluation and describe some examples[9].

I. Ungurean and N. Gaitanet al. illustrated that the Internet of Things (IoT) is an emerging concept that has revolutionized the use of new technologies in everyday life. The economic impact of IoT becoming very important, and it began to be used in the industrial environment under the name of the Industrial Internet of Things (IIoT) concept, which is a sub-domain of IoT. The IIoT changes the way industrial processes are controlled and monitored, increasing operating efficiency. This article proposes a software architecture for IIoT that has a low degree of abstraction compared to the reference architectures presented in the literature. The architecture is organized on four-layer and it integrates the latest concepts related to fog and edge computing. These concepts are activated through the use of fog/edge/gateway nodes, where the processing of data acquired from things is performed and it is the place where things interact with each other in the virtual environment. The main contributions of this paper are the proposal and description of a complete IIoT software architecture, the use of a

unified address space, and the use of the computing platform based on SoC (System on Chip) with specialized co-processors in order to be able to execute in real-time certain time-critical operations specific to the industrial environment [10].

T. Yang et al. illustrated that the Software architecture is the heart of web-based software systems determining its components and their connections. These days, fast release and quick delivery of next-generation software, which is the primary goal of the software industry, triggers an occurring error in the software development process. Therefore, recovery and metric measurement techniques are essential tools to assess the quality and soundness of web-based software architecture and return the system to the earlier or original stable state. Reusability techniques could be used to decrease the time, effort, and cost of software development as well. Clustering is a commonly used data mining technique employed to achieve these goals. Therefore, this paper as a first survey presents a literature review for web-based software architecture clustering models that are categorized into software architecture recovery, software architecture metric measurement, and software architecture reusability[11].

T. Gu et al. discussed that the Current research on software vulnerability analysis mostly focus on source codes or executable programs. But these methods can only be applied after software is completely developed when source codes are available. This may lead to high costs and tremendous difficulties in software revision. On the other hand, as an important product of software design phase, architecture can depict not only the static structure of software, but also the information flow due to interaction of components. Architecture is crucial in determining the quality of software. As a result, by locating the architecture-level information flow that violates security policies, vulnerabilities can be found and fixed in the early phase of software development cycle when revision is easier with lower cost. In this paper, an approach for analyzing information flow vulnerability in software architecture is proposed. First, the concept of information flow vulnerability in software architecture is elaborated. Corresponding security policies are proposed. Then, a method for constructing service invocation diagrams based on graph theory is proposed, which can depict information flow in software architecture. Moreover, an algorithm for vulnerability determination is designed to locate architecture-level vulnerabilities. Finally, a case study is provided, which verifies the effectiveness and feasibility of the proposed methods [12].

M. Ozkaya and F. Erata discussed that the Software architecture views separate up the software architectures into so many categories, each of which deals with a separate issue. For modelling software architectures from plenty of angles, practitioners depend greatly on the Unified Modeling Language (UML). In this work, we sought to understand the many perspectives through which practitioners used UML to describe software architectures. In order to understand how stakeholders use UML from six distinct perspectives functional, informational, concurrency, development, and operational 109 practitioners with a variety of backgrounds have been polled. Each perspective has been taken into account in the context of the range of software models that may be constructed in that perspective [13].

T. Glatard et al. stated that the Science gateways often rely on workflow engines to execute applications on distributed infrastructures. We investigate six software architectures commonly used to integrate workflow engines into science gateways. In tight integration, the workflow engine shares software components with the science gateway. In service invocation, the engine is isolated and invoked through a specific software interface. In task encapsulation, the engine is wrapped as a computing task executed on the infrastructure. In the pool model, the engine is bundled in an agent that connects to a central pool to fetch and execute workflows. In nested workflows, the engine is integrated as a child process of

another engine. In workflow conversion, the engine is integrated through workflow language conversion. We describe and evaluate these architectures with metrics for assessment of integration complexity, robustness, extensibility, scalability and functionality. Tight integration and task encapsulation are the easiest to integrate and the most robust. Extensibility is equivalent in most architectures. The pool model is the most scalable one and meta-workflows are only available in nested workflows and workflow conversion. These results provide insights for science gateway architects and developers [14].

B. Williams and J. Carver discussed that the increasing demands on software, software developers must produce software that can be changed without the risk of degrading the software architecture. One way to address software changes is to characterize their causes and effects. A software change characterization mechanism allows developers to characterize the effects of a change using different criteria, e.g. the cause of the change, the type of change that needs to be made, and the part of the system where the change must take place. This information then can be used to illustrate the potential impact of the change. This paper presents a systematic literature review of software architecture change characteristics. The results of this systematic review were used to create the Software Architecture Change Characterization Scheme (SACCS). This report addresses key areas involved in making changes to software architecture. SACCS's purpose is to identify the characteristics of a software change that will have an impact on the high-level software architecture [15].

DISCUSSION

Asking oneself why one wishes to describe the architecture in the first place is necessary before selecting what to say in an architectural description language. We have proposed three potential applications for ADLs in this paper: analysis, building, and debate. The Intercol module interconnection language, the Rapide language, the MetaH language, and the ArTek language are the four ADLs that we have succinctly introduced.

Additionally, we have spoken about the VHDL hardware description language, which has some of the characteristics we would want to see in a software architecture description language, as well as the BETA programming language (from an architectural point of view). Note that the current ADLs provide only mediocre support for stakeholder talks, with the exception of ArTek and MetaH, both of which are focused on a particular domain. We want to utilize regular programming languages for architectural description perhaps with extra capabilities for architecture description. The integration of such an approach with current tools would be advantageous for building. However, in the absence of further restrictions, such integration can potentially have a detrimental effect on system upkeep.

It may be simpler to modify an architecture during maintenance if it is described in the implementation's language, which raises the danger of architecture erosion. Additionally, care must be made to keep the low-level design distinct from the architectural description. Programming languages of today are relatively unsuitable for more advanced analyses since the architecture is not an explicit concept in the description. Nor are they useful for stakeholder discussion, since they describe structure and functionality rather than requirements.

The VHDL language has a number of properties one would like to see in an architecture description language. Although software architecture probably cannot be described using VHDL, the language concepts are relatively close to the ones commonly used to describe software architecture.

CONCLUSION

This is the software architecture level of design. There is a considerable body of work on this topic, including module interconnection languages, templates and frameworks for systems that serve the needs of specific domains, and formal models of component integration mechanisms. In addition, an implicit body of work exists in the form of descriptive terms used informally to describe systems. And while there is not currently a well-defined terminology or notation to characterize architectural structures, good software engineers make common use of architectural principles when designing complex software. Many of the principles represent rules of thumb or idiomatic patterns that have emerged informally over time. Others are more carefully documented as industry and scientific standards. It is increasingly clear that effective software engineering requires facility in architectural software design. First, it is important to be able to recognize common paradigms so that high-level relationships among systems can be understood and so that new systems can be built as variations on old systems. Second, getting the right architecture is often crucial to the success of a software system design; the wrong one can lead to disastrous results. Third, detailed understanding of software architectures allows the engineer to make principled choices among design alternatives. Fourth, an architectural system representation is often essential to the analysis and description of the high-level properties of a complex system.

REFERENCES

[1] C. Hofmeister, P. Kruchten, R. L. Nord, H. Obbink, A. Ran, en P. America, "A general model of software architecture design derived from five industrial approaches", *J. Syst. Softw.*, vol 80, no 1, bll 106–126, Jan 2007, doi: 10.1016/j.jss.2006.05.024.

[2] L. De Silva en D. Balasubramaniam, "Controlling software architecture erosion: A survey", *J. Syst. Softw.*, 2012, doi: 10.1016/j.jss.2011.07.036.

[3] N. Medvidovic en G. Edwards, "Software architecture and mobility: A roadmap", *J. Syst. Softw.*, 2010, doi: 10.1016/j.jss.2009.11.004.

[4] J. Bishung *et al.*, "A critical analysis of topics in software architecture and design", *Adv. Sci. Technol. Eng. Syst.*, 2019, doi: 10.25046/aj040228.

[5] C. C. Venters *et al.*, "Software sustainability: Research and practice from a software architecture viewpoint", *J. Syst. Softw.*, vol 138, bll 174–188, Apr 2018, doi: 10.1016/j.jss.2017.12.026.

[6] H. van Vliet en A. Tang, "Decision making in software architecture", *J. Syst. Softw.*, vol 117, bll 638–644, Jul 2016, doi: 10.1016/j.jss.2016.01.017.

[7] L. Garcés, F. Oquendo, en E. Y. Nakagawa, "Software mediators as first-class entities of systems-of-systems software architectures", *J. Brazilian Comput. Soc.*, 2019, doi: 10.1186/s13173-019-0089-3.

[8] M. Razavian, B. Paech, en A. Tang, "Empirical research for software architecture decision making: An analysis", *J. Syst. Softw.*, vol 149, bll 360–381, Mrt 2019, doi: 10.1016/j.jss.2018.12.003.

[9] N. Thanachawengsakul, P. Wannapiroon, en P. Nilsook, "The knowledge repository management system architecture of digital knowledge engineering using machine learning to promote software engineering competencies", *Int. J. Emerg. Technol. Learn.*, 2019, doi: 10.3991/ijet.v14i12.10444.

[10] I. Ungurean en N. C. Gaitan, "A Software Architecture for the Industrial Internet of Things—A Conceptual Model", *Sensors*, vol 20, no 19, bl 5603, Sep 2020, doi: 10.3390/s20195603.

[11] X. Xu, Q. Lu, Y. Liu, L. Zhu, H. Yao, en A. V. Vasilakos, "Designing blockchain-based applications a case study for imported product traceability", *Futur. Gener. Comput. Syst.*, 2019, doi: 10.1016/j.future.2018.10.010.

[12] T. Gu, M. Lu, L. Li, en Q. Li, "An Approach to Analyze Vulnerability of Information Flow in Software Architecture", *Appl. Sci.*, vol 10, no 1, bl 393, Jan 2020, doi: 10.3390/app10010393.

[13] M. Ozkaya en F. Erata, "A survey on the practical use of UML for different software architecture viewpoints", *Inf. Softw. Technol.*, vol 121, bl 106275, Mei 2020, doi: 10.1016/j.infsof.2020.106275.

[14] T. Glatard *et al.*, "Software architectures to integrate workflow engines in science gateways", *Futur. Gener. Comput. Syst.*, vol 75, bll 239–255, Okt 2017, doi: 10.1016/j.future.2017.01.005.

[15] B. J. Williams en J. C. Carver, "Characterizing software architecture changes: A systematic review", *Inf. Softw. Technol.*, vol 52, no 1, bll 31–51, Jan 2010, doi: 10.1016/j.infsof.2009.07.002.

CHAPTER 6

FUNDAMENTAL SOFTWARE ARCHITECTURES TECHNIQUES

Ms. Sudha Y, Assistant Professor
Department of Computer Science and Engineering, Presidency University, Bangalore, India
Email Id- sudha.y@presidencyuniversity.in

ABSTRACT:

We currently lack quantitative methods to evaluate software architectures. As a result of the continent's youth. There are, meanwhile, certain qualitative methods that seem to provide reasonable outcomes. However, even using the most elementary methodologies expressed concern about their maturity and usefulness. In the field of assessing software architecture, this article is state-of-the-art. In the future this chapter will help to evaluate the key information about software architecture and to improve the new methods in the future. On the other hand it will help students and researchers in their research.

KEYWORDS:

Computer Science, Information Technology, Software Engineering, Software Design.

INTRODUCTION

A systems lifetime is measured in 10-20 years. During this time period the system will most likely be modified several times. Even though the system is stressed by this factor and probably many others, the system is supposed to be prepared for all this. It is very important that the software architecture is evaluated as early as possible. It is a well-known fact that it costs a lot more to fix a bug late in the development process than in an early stage. This is of course also true for software architectures. Formal reviews, described in, are one way to achieve the hunt for errors in the software architectures. Although the formal review in is focused on design and code reviews, they should be just as applicable for reviewing software architectures. The most effective and useful evaluation is a quantitative one [1]. Unfortunately, I have to make a conclusion already in the introduction, because of one single fact. There are no quantitative evaluation techniques available today.

Measuring Techniques

With this approach to evaluate software architectures we are actually able to set a (some) figure(s) on how good this particular software architecture really is. There are, according to, two basic techniques within the measuring area.

i. Metrics

This is a rather well explored area within the design and coding phase. It could also be used in the software architecture phase as well. In order make use of metrics we must have a good number of details specified in the software architecture. This is basically because otherwise we do not have anything to measure on. The success of metrics in both the design phase and in particular the coding phase is that these phases are so formalized that tools could, more or less, automatically calculate all the metrics, only given, for instance, the source code of the system. The metrics could be both general as well as domain specific. One kind of domain specific metric is a tool for calculating the rate-monotonic scheduling of a concurrent real time system. There are also several algorithms to calculate the maximum waiting time for

acquiring a lock that protect some global data within the concurrent real time system domain as found in [2].

ii. Simulations, Prototypes, and Experiments:

This is things that really costs a lot. To build a simulation tool or prototypes for one single software architecture is probably too expensive. However, often some kind of prototypes are built for demonstrating the systems capabilities for the customer. Most certainly this kind of prototype could be used to evaluate the software architecture. This synergy effect could be possible if at least the software architecture designer could participate in building the prototype [3].

Fundamental Evaluation Technique Discussion

When choosing the technique to evaluate a software architecture there are several questions that must be answered. Some are: How mature is our development process? Are we going to use the evaluation technique for a whole product family or not? In which stage of the development process of the software architecture do we wish to conduct the evaluation?

These are some of the questions that are important. If we look a little ahead in this paper, we will discover in the next few sections that all techniques proposed uses slightly different versions of the scenario questioning technique. None of the methods described supports the quantitative approach described earlier. This could be seen as an immaturity in the software architecture domain, and particularly in the maturity of evaluation of the software architecture.

Software Architecture Analysis Method (SAAM)

Software Architecture Analysis Method (SAAM) was developed by Kazman, Bass, Abowd, and Webb in 1994. The main issue with this method is to compare different software architectures in order to conclude which one is the best to proceed with and to become the base for the software system to be built. SAAM discusses three perspectives of a software architecture.

i. Functionality - This perspective is how the system should work from a functional point of view. Traditional techniques, such as, object orientated modeling techniques, could be used. The scoop of this perspective is to break down the functionality and coarsely describe the behaviour of the system.
ii. Structure - This is the perspective that mostly correspond to the traditional view of a software architecture. The structure describes which components the system is built from and the different connectors that connects the components.

A simple notation, later to be used in the example, can be found in Figure 1. The “Process” in the figure is a traditional process or a single thread-of-control within a concurrent system. “Computational Component” is what is used to call a module, procedure, function, method, or such a thing. The “Passive Data Repository” is about persistent storage of data, such as in files, etc. On the other hand “Active Data Repository” is about an active persistent data storage that is able to inform others whenever a value is changed. There are also two types of connections, “Data Flow” which is a connection that is transferring data, both unidirectional as well as Bi-directional. The connection type called “Control Flow” is dealing with modules that call each other, it could be both unidirectional as well as Bi-directional as shown in Figure 1.

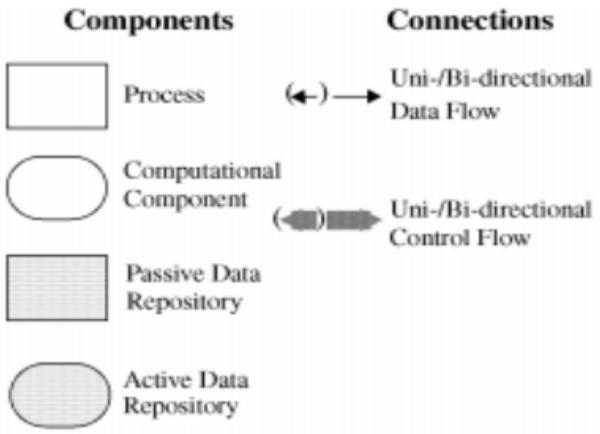

Figure 1: Represented that the A simple notation of the Structure perspective in SAAM [4]**.**

Allocation

This perspective maps the Functionality on Figure 1. A simple notation of the Structure perspective in SAAM. 26 the Structure in order to find matches and, more important, where the Structure does not map the Functionality.

Step 1: The Canonical Function Partitioning

The first step in SAAM is to find a canonical function partitioning for the system or domain. Hopefully there are such a partitioning to be find. Otherwise it may be invented by specialists within the domain.

Step 2: Mapping the Canonical Function onto the Software Architecture

Given to this step is the software architecture and Based on the Structure and the Functionality this step maps the functional partitioning onto the architecture's structural decomposition.

Step 3: Choosing Quality Attributes

This third step is dealing with the quality attributes. Here we select which quality attributes to stress the software architecture with. Exactly how this selection of attributes is performed is however quite unclear. Probably it is based on the requirement specification, but there are many more aspects to take in count than those found in the requirement specification. One example is, for instance, whether the system will become the first system in a whole product family, and thus quality attributes about adaptability may become more interesting.

Step 4: Choosing the Scenarios

The next step in SAAM is to create a number of scenarios that will stress the software architecture on those quality attributes, which is mention above. How these scenarios are created and by who is, however, not defined in SAAM. Preferably all stakeholders should participate in this step in order to get as realistic scenarios as possible. Perhaps the requirement specification could be useful in this step to find suitable scenarios.

Step 5: The Evaluation

The last step in SAAM is about the evaluation itself. Now SAAM iterates through all the scenarios for each software architecture and to put it simple, the software architecture which

got the most points wins. If no software architecture fulfills the scenarios then new software architectures must be found and examined.

It is an introductory guide for software architects. Nowadays, the title of "architect" is very hot around the world, but there is no real guide to becoming a software architect.

i. The general concept of software architecture

ii. Architectural styles

iii. Technology

iv. soft skills (making architectural decisions, risk analysis techniques, presentation skills, management team relations, negotiation, architect career planning)

v. Design principles

In the real world, there is no one fixed solution for one problem. When we are doing or learning architecture knowledge, we must understand that many decisions made by architects are based on their actual situation. **The architectural styles** refer to the type of architectural style that the system implements such as micro services, hierarchical, or microkernel. There is a few examples we can look at below in Figure 2:

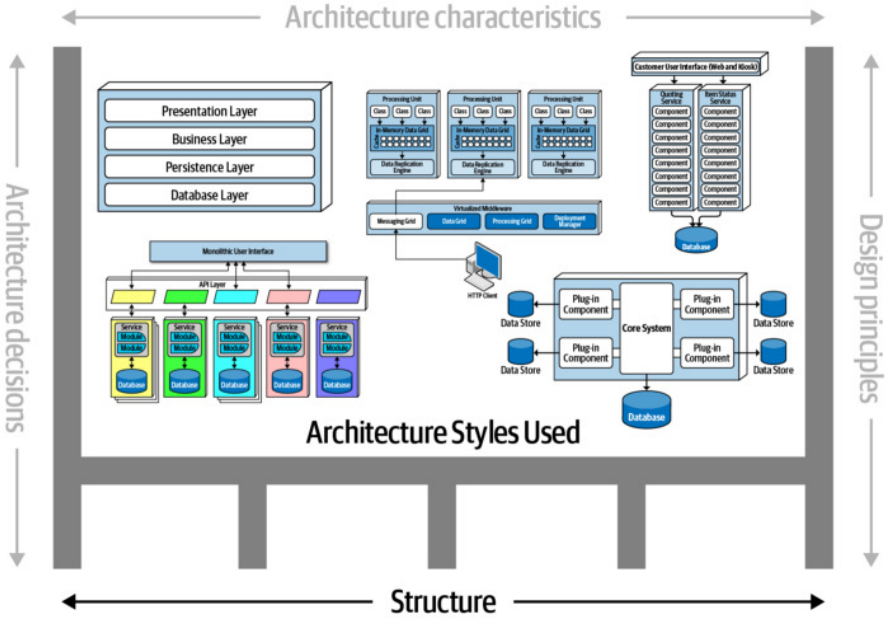

Figure 2: Illustrated that the Architecture Style [5]**.**

Architectural decisions define the rules for how the system should be constructed. For instance, the architect can decide if only the business layer and the service layer can access the database and restrict the presentation layer from directly calling the database. Architectural decisions form the constraints of the system and guide the development team on what is allowed and what is not allowed the SOLID principle which is mention in Table 1.

Table 1: Illustrated that the SOLID Principles.

S	Single Responsibility Principle	Every object should have a single responsibility and that should be encapsulated by the class.
O	Open Close Principle	Software should be open for extension, but closed

		for modification.
L	Liskov's Substitution Principle	Any subclass should always be usable instead of its parent class.
I	Interface Segregation Principle	Many client specific interfaces are better than one general purpose interface.
D	Dependency Inversion Principle	Abstractions should not depend upon details and details should depend upon abstraction.

Architecture Characteristics

A company decides to solve a particular problem using software, so it gathers a list of requirements for that system. A wide variety of techniques exist for the exercise of requirements gathering, generally defined by the software development process used by the team. But the architect must consider many other factors in designing a software solution, as illustrated in Figure 3.

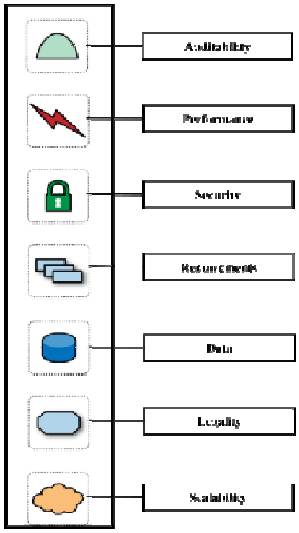

Figure 3: Represented That the both Domain Requirements and Architectural Characteristics

Architects may collaborate on defining the domain or business requirements, but one key responsibility entails defining, discovering, and otherwise analyzing all the things the software must do that isn't directly related to the domain functionality:

i. Architectural Characteristics:

Many things, including the role that architects have in defining architectural characteristics, the important aspects of the system independent of the problem domain. Many organizations describe these features of software with a variety of terms, including nonfunctional requirements, but we dislike that term because it is self-denigrating. Architects created that term to distinguish architecture characteristics from functional requirements, but naming something nonfunctional has a negative impact from a language standpoint: how can teams be convinced to pay enough attention to something "nonfunctional"? Another popular term is quality attributes, which we dislike because it implies after-the-fact quality assessment rather than design. We prefer architecture characteristics because it describes concerns critical to the

success of the architecture, and therefore the system as a whole, without discounting its importance. An architecture characteristic meets three criteria:

A. Specifies a no domain design consideration,
B. Influences some structural aspect of the design,
C. Is critical or important to application success

LITERATURE REVIEW

M. Glaser et al. illustrated that the DNA nanotechnology first appeared almost 40 years ago, the field has advanced from its initial conceptualizations of rather simple DNA structures having a branched, multi-strand architecture to beautifully complicated systems composed of hundreds or thousands of unique strands, with both the ability to precisely control the leadership roles down to the molecular level. While the earliest building approaches, like straightforward Holliday interchanges or tiles, could theoretically be designed on paper and in a short period of time, the advent of complex methods, like DNA origami or DNA bricks, requires software to minimize the amount of time needed and the odds of human error during the design process. When it is available, easily accessible design software accelerates our capacity to share methodologies with researchers from a broad range of sectors, and it has aided in hastening the adoption of techniques like DNA origami into a wide range of applications ranging from biomedicine to photo detectors. Here, we examine the creation and state-of-the-art of CAD software to allow a range of essential techniques for exploiting structural DNA technology. We trace the evolution and relevance of several software packages to the present state-of-the-art, starting with both the earliest tools for predicting sequential secondary structure of nucleotides and laying a special emphasis on open source programs[6].

O. Asuna et al. stated that the Computational models of emotion (CMEs) are software systems designed to imitate particular aspects of human emotions. The main purpose of this type of computational model is to capture the complexity of the human emotion process in a software system that is incorporated into a cognitive agent architecture. However, creating a CME that closely imitates the actual functioning of emotions demands to address some challenges such as (i) sharing information among independently developed cognitive and affective components, and (ii) interconnecting complex cognitive and affective components that must interact with one another in order to generate realistic emotions, which may even affect agents' decision making. This paper proposes an architectural pattern aimed at cataloging and describing fundamental components of CMEs and their interrelationships with cognitive components. In this architectural pattern, external cognitive components and internal affective components of CMEs do not interact directly but are extended by including message exchange methods in order to use a publish-subscribe channel, which enables their intercommunication, thus attenuating issues such as software heterogeneity. This structural approach centralizes communication management and separates the inherent complexity of the cognitive and affective processes from the complexity of their interaction mechanisms. In so doing, it enables the design of CMEs' architectures composed of independently developed affective and cognitive components. The proposed architectural pattern attempts to make progress in capturing the complex process of human emotions in a software system that adheres to software engineering best practices and that incorporates quality attributes such as flexibility and interoperability[7].

G. Ma et al. illustrated that the adoption of Building Information Modeling (BIM) will definitely improve the efficiency and quality of the AEC (architecture, engineering and construction) industry. However, many factors need to be improved before BIM adoption.

Based on the interaction between institutions and technology, factors affecting BIM adoption in AEC organizations, within the context of China, are identified and analyzed. Firstly, there is 21 factors are identified by literature research. Then, an interpretive structural model (ISM) technique is used to establish a hierarchical structure, and matrix impacts cross-reference multiplication applied to a classification (MICMAC) is used for factor classification. The results indicate that corporate/project leadership and software functionality are the two fundamental factors. What's more, the dynamic mechanism has gradually changed from top-down to a combination of top-down and bottom-up [8].

E. García-Martín et al. illustrated that the Energy consumption has been widely studied in the computer architecture field for decades. While the adoption of energy as a metric in machine learning is emerging, the majority of research is still primarily focused on obtaining high levels of accuracy without any computational constraint. We believe that one of the reasons for this lack of interest is due to their lack of familiarity with approaches to evaluate energy consumption. To address this challenge, we present a review of the different approaches to estimate energy consumption in general and machine learning applications in particular. Our goal is to provide useful guidelines to the machine learning community giving them the fundamental knowledge to use and build specific energy estimation methods for machine learning algorithms. We also present the latest software tools that give energy estimation values, together with two use cases that enhance the study of energy consumption in machine learning [9].

J. Zalewski discussed the fundamentals of software architectures for real-time systems are covered in this article. A real-time architecture's core notion is derived from the control engineering concept of feedback. For each of the three main real-time system types, a generic architecture is created. Then a basic design pattern that applies to all significant architectures is revealed. The main architectural changes for distributed systems and safety-related systems are then discussed. The discussion of a case study and tool assistance for architectural design concludes [10].

S. Hassan et al. stated that the Microservices are an emerging architectural style for autonomous, scalable, and more dependable computing that has achieved considerable recognition and support in the software industry. The requirement for improved alignment of technical design choices with increasing value potentials of architectures has been a major driving force while behind shift to microservices. Despite the widespread use of microservices, there is still a lack of a systematic comprehension of the shift and of agreement on the ideas and actions that underlie it. In this article, we provide the results of a comprehensive mapping research that compiles a range of opinions, strategies, and initiatives often used to facilitate the switch to microservices. In addition to offering a working description of the transition and the technological operations that underlie it, the research seeks to improve understanding of the change. Microservitization is the word we use to describe the process and technological actions leading to microservice structures. We then shed light on a key issue with microservitization: the granularity of microservices and the justification for recognizing them as first-class entities. This paper evaluates the state-of-the-art and current practise in reasoning about microservice refinement; it reviews modelling methodologies, factors taken into account, rules, and procedures utilised to do so. This study provides insights areas that might be explored further in the context of thinking about the granularity of microservices [11].

T. Chen gives the overview of the present state of software-defined mobile networks is given in this article (SDMNs). A potential method to control the complexity of communication networks is software defined networking. The complexity of network management in 5G

mobile networks and beyond, driven by rising mobile traffic demand, varied wireless environments, and various service needs, is the reason for the necessity for SDMN. In order to handle complexity and provide flexibility in 5G networks, it is essential to implement new radio network architecture by using software oriented design, the separation of the data and control planes, and network virtualization. Because mobile networks deal with the wireless access issue in complicated radio settings, while the Internet primarily deals with the packet forwarding problem, it is obvious that software oriented design for mobile networks will be fundamentally different from SDN for the Internet. SDMN's evolution is influenced by specific needs for mobile networks. With a specific emphasis on the software-defined design for radio access networks, we explain the demands and requirements of SDMN in this paper. We examine the core issues with radio access networks that need SDN architecture and provide an SDMN idea. We provide a quick review of the SDMN and standardization initiatives' existing solutions. Although SDN design is now concentrated on mobile core networks, we contend that expanding SDN to radio access networks would be the logical next step. We outline a number of topics for future research on SDN for radio access networks and anticipate further basic investigations to fully realize the promise of software-defined 5G networks [12].

A. Adnan et al. illustrated that the Software-Defined Vehicular Network (SDVN) has gained a lot of attention recently in the academic and research community due to the fast growth of Vehicular Ad Hoc Networks (VANETs) and the progress of Software Defined Networking (SDN) as an emerging technology. The network is scalable and simple because to SDN's distinctive qualities and capabilities, including its flexibility, programmability, and centralised control. Currently, the primary study areas for VANETs researchers to take into account while creating an effective and safe VANETs architecture are traffic management and secure exchange of vehicle information utilizing the public network. In order to provide a unique and safe hierarchical architecture for SDVN, this article emphasises the potential attack vectors that have been found and effectively addresses network weaknesses. For effective and secure communication between vehicles, we presented a Public Key Infrastructure-based digital signature paradigm to address the aforementioned issue. Additionally, we employed the three-way handshake mechanism for secure session establishment and secure data transfer in the SDN controller together with the public key authority infrastructure for Vehicle to Infrastructure. The well-known simulation programme AVISPA is used to verify the suggested security. Additionally, a formal security model is used to efficiently and favorably evaluate the design hierarchic architecture's core security features for SDVN. In a comparative study, we demonstrate that, in contrast to previous state-of-the-art systems, our suggested approach meets all the necessary security features[13].

M. Anderrson et al. illustrated that the number of scholars and industry professionals have claimed that there has been a ‘software-biased shift’ in the nature and direction of innovation, in that software development is a core part of innovation activities in firms across a wide array of industries. Empirical firm-level evidence of such a shift is still scant. In this paper, we employ new and unique firm-level survey data on the frequency and nature of software development among firms in Sweden, matched with the Community Innovation Survey (CIS). We find robust evidence supporting a software bias in innovation, in that software development is associated with a higher likelihood of introducing innovations, as well as higher innovation sales among firms in both manufacturing and service industries. Furthermore, this positive relationship is stronger for firms that employ in-house software developers than for those that only use external developers, suggesting that there is a hierarchy but possibly also a complementarity between in-house and external software development. We also find support for complementarity between software-based technology

and human capital; the estimated marginal effect of software development on innovation is particularly strong for firms that combine in-house software development with a highly educated workforce in both STEM and other disciplines[14].

M. Mishra et al. illustrated that the Quality pressure is one of the factors affecting processes for software development in its various stages. DevOps is one of the proposed solutions to such pressure. The primary focus of DevOps is to increase the deployment speed, frequency and quality. DevOps is a mixture of different developments and operations to its multitudinous ramifications in software development industries, DevOps have attracted the interest of many researchers. There are considerable literature surveys on this critical innovation in software development, yet, little attention has been given to DevOps impact on software quality. This research is aimed at analyzing the implications of DevOps features on software quality. DevOps can also be referred to a change in organization cultures aimed at removal of gaps between the development and operations of an organization. The adoption of DevOps in an organization provides many benefits including quality but also brings challenges to an organization. This study presents systematic mapping of the impact of DevOps on software quality. The results of this study provide a better understanding of DevOps on software quality for both professionals and researchers working in this area. The study shows research was mainly focused in automation, culture, continuous delivery, fast feedback of DevOps [15].

DISCUSSION

We discuss a number of architectural traits that have various, complex meanings. Performance is a good illustration. Many projects examine overall performance, such as how lengthy a web application's request and response cycles are. However, a lot of effort has been put into creating performance budgets, with particular budgets for certain components of the application, by architects and DevOps engineers. For instance, many companies have studied user behavior and found that the ideal first-page render time, or the first visible indication that a webpage is loading in a browser or on a mobile device, is half a second. Most applications, however, fall in the double-digit range for this metric. However, this is a crucial indicator to monitor for contemporary websites that want to attract as many people as possible, and the companies that support them have developed very precise metrics. Some of these indicators have further effects on how apps are created. Many progressive companies set K-weight budgets for page downloads, which is the maximum amount of libraries and frameworks that can fit on one page. Physics limitations which limit the number of bytes that can go through a network at once, particularly for mobile devices in high-latency areas are the basis for their justification for this structure. High-level teams build their criteria on statistical analysis rather than merely establishing concrete performance metrics. Consider the case of a video streaming business that wishes to keep an eye on scalability. Engineers assess the scale over time, create statistical models, and trigger warnings if the real-time measurements deviate from the prediction models rather than setting an arbitrary figure as the target. The types of traits that teams may currently assess are increasing fast, along with tools and sophisticated knowledge. A failure might signify one of two things: the model is erroneous or something is awry. For instance, a lot of teams have lately concentrated on performance budgets for metrics like first CPU idle and first contently paint, both of which speak volumes about performance difficulties for mobile website visitors. Teams will develop new tools and methods for measuring when devices, targets, capabilities, and a variety of other factors change.

CONCLUSION

This chapter introduced the fundamental concepts of object orientation, open systems, and object-oriented architectures. It also discussed object orientation in terms of isolating changes in software systems by combining the data and processing into modules called objects. Object technology is a capability that is already present and entering the mainstream of software development. Object technology is broadly supported by commercial industry through software vending and by many mainstream end-user organizations in their application development. As discussed, the only sustainable commercial advances are through open systems forms of commercial technology. With proprietary technologies, the obsolescence of capabilities conflicts with the need to build stable application environments that support the extension of application functionality. Additionally, stovepipe systems are the pervasive form of application architecture but can be reformed into more effective component object architectures. In the next chapter, object technologies and various reference models that make these technologies understandable will be described also, the different architectural layers, including two-tier, three-tier, N-Tier, and peer-to-peer approaches, were examined. The advantages of using different layering techniques were covered and provide essential guidance in deciding when a particular project merit one of the more complex architectural alternatives. In conclusion, a wide range of open systems client-server technologies support object orientation. These technologies enable the construction of a wide array of distributed systems based upon objects and components.

REFERENCES

[1] V. P. Castro-Rivera, R. A. Herrera-Acuña, en M. A. Villalobos-Abarca, "Development of a web software to generate management plans of software risks", *Inf. Tecnol.*, 2020, doi: 10.4067/S0718-07642020000300135.

[2] M. Singh, A. Mittal, en S. Kumar, "Survey on Impact of Software Metrics on Software Quality", *Int. J. Adv. Comput. Sci. Appl.*, 2012, doi: 10.14569/ijacsa.2012.030121.

[3] W. Ren, H. Isler, M. Wolf, J. Ripoll, en M. Rudin, "Smart Toolkit for Fluorescence Tomography: Simulation, Reconstruction, and Validation", *IEEE Trans. Biomed. Eng.*, 2020, doi: 10.1109/TBME.2019.2907460.

[4] L. Dobrica en E. Niemelá, "A survey on software architecture analysis methods", *IEEE Transactions on Software Engineering*. 2002. doi: 10.1109/TSE.2002.1019479.

[5] A. Athar, R. Muzamal Liaqat, en F. Azam, "A Comparative Analysis of Software Architecture Evaluation Methods", *J. Softw.*, 2016, doi: 10.17706/jsw.11.9.934-942.

[6] M. V. Debole *et al.*, "TrueNorth: Accelerating From Zero to 64 Million Neurons in 10 Years", *Computer (Long. Beach. Calif).*, 2019, doi: 10.1109/MC.2019.2903009.

[7] K. Redmond, L. Luo, en Q. Zeng, "A Cross-Architecture Instruction Embedding Model for Natural Language Processing-Inspired Binary Code Analysis", 2019. doi: 10.14722/bar.2019.23057.

[8] G. Ma, J. Jia, J. Ding, S. Shang, en S. Jiang, "Interpretive Structural Model Based Factor Analysis of BIM Adoption in Chinese Construction Organizations", *Sustainability*, vol 11, no 7, bl 1982, Apr 2019, doi: 10.3390/su11071982.

[9] E. García-Martín, C. F. Rodrigues, G. Riley, en H. Grahn, "Estimation of energy consumption in machine learning", *J. Parallel Distrib. Comput.*, vol 134, bll 75–88, Des 2019, doi: 10.1016/j.jpdc.2019.07.007.

[10] J. Zalewski, "Real-Time Software Architectures and Design Patterns: Fundamental Concepts and Their Consequences", *IFAC Proc. Vol.*, vol 32, no 1, bll 1–13, Mei 1999, doi: 10.1016/S1474-6670(17)39958-5.

[11] S. Hassan, R. Bahsoon, en R. Kazman, "Microservice transition and its granularity problem: A systematic mapping study", *Softw. Pract. Exp.*, vol 50, no 9, bll 1651–1681, Sep 2020, doi: 10.1002/spe.2869.

[12] T. Chen, M. Matinmikko, X. Chen, X. Zhou, en P. Ahokangas, "Software defined mobile networks: concept, survey, and research directions", *IEEE Commun. Mag.*, vol 53, no 11, bll 126–133, Nov 2015, doi: 10.1109/MCOM.2015.7321981.

[13] T. Bauer, P. Oliveira Antonino, en T. Kuhn, "Towards Architecting Digital Twin-Pervaded Systems", 2019. doi: 10.1109/SESoS/WDES.2019.00018.

[14] K. Reizenbuk, T. Sarapulova, S. Shchedrin, en I. Shchedrina, "Application of distributed computing in developing architecture of intelligent information system for automated stock exchange trading", *J. Adv. Res. Dyn. Control Syst.*, 2019.

[15] A. Mishra en Z. Otaiwi, "DevOps and software quality: A systematic mapping", *Comput. Sci. Rev.*, vol 38, bl 100308, Nov 2020, doi: 10.1016/j.cosrev.2020.100308.

CHAPTER 7

SOFTWARE ARCHITECTURE QUALITY ATTRIBUTES EVALUATION

Dr. Prabagar. S, Associate Professor
Department of Computer Science and Engineering, Presidency University, Bangalore, India
Email Id- prabagar.s@presidencyuniversity.in

ABSTRACT:

Software architecture is still a rapidly evolving field with many of key challenges, despite the software engineering community's heightened focus and extensive effort in recent years. How can we determine that a software architecture will fulfil its criteria, if one exists? Primarily, its standards for quality. In this chapter, we will provide an explanation of evaluation approaches and demonstrate how a software architecture may be reviewed against its quality characteristics. In future this paper will be helps to another student and researcher for their work and d provide the accurate and sufficient data for study.

KEYWORDS:

Information Technology, Software Engineering, Software Design, Software quality, Software Analysis.

INTRODUCTION

Only 2% of the programmer could be utilized as provided, according to a 1982 General Accounting Office (GAO) analysis of nine software contracts. We think that one of the biggest influencing elements to this enormous failure was the way these contractors managed the characteristics. Some of the failures may be attributed to shifting client demands, for which a process-oriented approach or evolutionary delivery may have mitigated the disaster. However, we think that a large portion of the unsuccessful enterprises failed as a result of not meeting the quality standards. There may be a range of reasons for this, including the possibility that the quality standards were just not sufficiently stated, tested enough, or that if a test revealed that a criterion was not satisfied, the decision was either too late or expensive to make the necessary corrections [1].The quality standards must be taken into account from the inception and cannot be resolved towards the conclusion of the project and sprinkled on top of the system. It is pretty apparent that it will be less costly to take corrective steps early in a project if there is a good attribute requirement definition and a strategy for how to verify the requirements. As a result, we would want to evaluate the system's quality characteristics as soon as feasible, ideally at the architecture level. We cannot use a universal assessment approach to all quality characteristics when assessing them. Due to their unique character, different methodologies are not equally as effective in exposing the deficiencies of the various grades qualities. It should be emphasized that the implemented system may still fail on this point even if a structure may offer quality characteristics that will satisfy the quality standards[2].Many things that occur during development and performance may not be represented in the architecture and are thus not assessed. The rest of this essay is structured as follows: We endeavor to summaries a few basic themes and how they relate in the following section. Then, we provide a few different classification schemes for quality characteristics. Then, we evaluate several fundamental organizational design elements and how they are anticipated to influence the system's quality characteristics. Finally, we examine a few methods for evaluating an architecture.

Quality Attributes in Software Architecture

Software quality attributes are attributes of a software system that may be measured or tested and are utilised by quality architects. These attributes aid in determining if the following examples the needs and requirements of the stakeholders. The list of the most essential components of software architecture is provided below in Figure 1. You may order them according to the necessities and standards of your software project

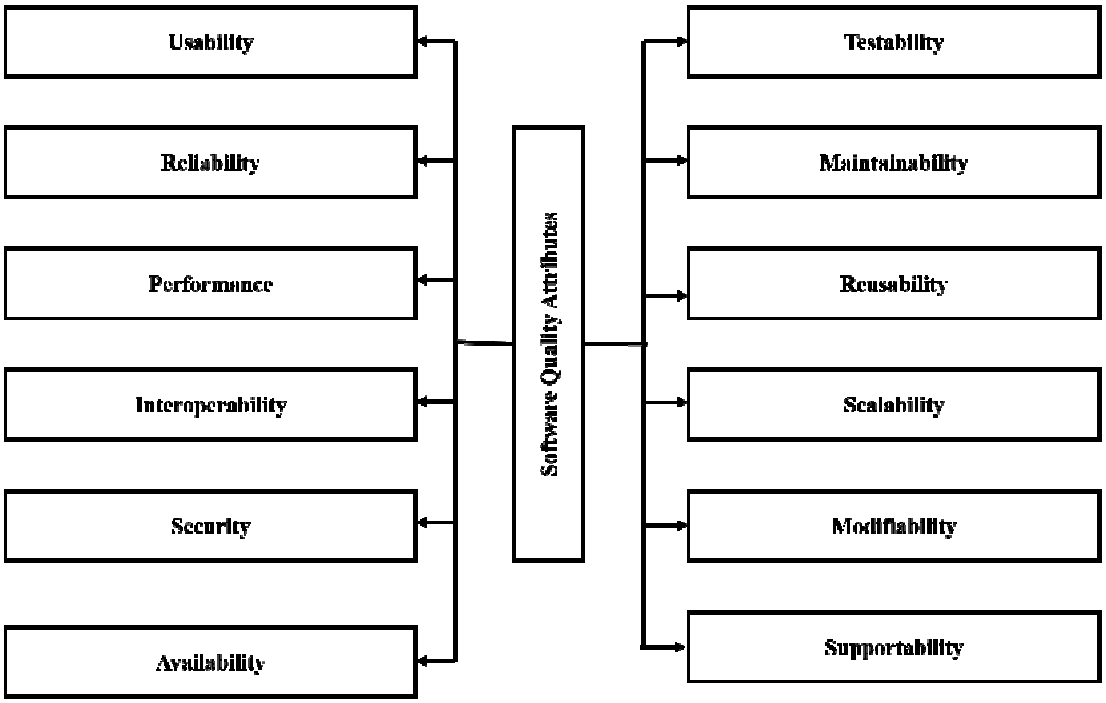

Figure 1: Represented that the All Software Architecture Attributes [3]**.**

i. Usability

This software quality attribute helps define the ease with which users can perform a specific task on the system (registering an account or adding an item to the shopping cart). What issues can be seen as usability problems? These may include inconsistency, too complicated signup process, poor error handling, or unclear navigation, to name a few. This software characteristic consists of the following sub-characteristics:

A. Operability denotes the degree to which a software system or a digital product has qualities that simplify its operation and control.

B. User error protection is the degree to which a software system or its components protect users from committing errors.

C. User interface aesthetics means the degree to which users get pleasing and satisfying interaction with a software solution or digital product.

ii. Reliability

The term reliability refers to the degree to which a software system or its components performs specific functions under predefined conditions for a certain period of time.

The main sub-characteristics of this software quality are as follows:

A. Maturity is the degree to which a software system meets the standards for reliability during regular operation.

B. Availability means the degree to which a software system can be accessed by the users when it is required.

C. Fault tolerance it is the degree to which a software system performs normally despite hardware or software issues.

D. Recoverability this sub-characteristic describes the degree to which if an interruption or failure takes place, a software system can recover the data directly affected by this failure.

iii. Compatibility

The term compatibility refers to the degree to which a software system or its component can exchange data with other systems or perform its main functions while sharing the same hardware or software environment. This software quality attribute includes two important sub-characteristics:

A. **Interoperability** defines the degree to which two or more software systems can exchange and reuse the data.

B. **Co-existence** defines the degree to which a software solution can perform its main functions while sharing a common software or hardware environment with other digital products without causing a negative impact on other applications or systems.

iv. Portability

Software portability is a quality attribute that refers to the degree to which a system or its components can be transferred from one hardware, software, or other environments to another. Here we should take into account the following sub-characteristics:

A. **Adaptability** is the degree to which a software system can be adapted to different hardware, software, or another environment.

B. **Replace ability** means the degree of efficiency to which one software product can replace another software solution designed for the same purpose.

C. **Install ability** defines the degree of efficiency to which a software solution can be installed or uninstalled in a particular environment.

v. Testability

In simple terms, testability determines how easy a software solution is to test to find bugs or ensure that it meets all predefined criteria. Testability can result from efficient collaboration between the development, product, and testing teams. The development team should consider the testing ability when implementing a new feature. Thus, the input of testers is required to ensure efficient testing.

vi. Scalability

This quality attribute refers to the ability of a software system to handle the increased load without decreasing its performance. You can improve the scalability of your application in two different ways. First, you can add more resources, namely memory, discs, or processors. That will be vertical scalability. Alternately, you can add more computing units and divide

the load between them. In this case, we talk about horizontal scalability. The following indicators will help you measure this quality attribute:

A. The system's possibility to be scaled horizontally.

B. Scaling limitation, namely the maximum number of servers or the network capacity.

C. The growing number of transactions or amount of content, in other words, the possibility to scale.

vii. Flexibility

Flexibility is another attribute of good software products that can easily adapt to future changes. To be more specific, we say that the application is flexible when it can run smoothly on any device, platform, or operating system. Besides, it can be easily integrated with any other third-party software solution. A clear advantage of flexible software is that it provides new business opportunities. By enhancing your existing software with new features and integrations, you can gain a competitive advantage in your niche and, thus, drive more customers.

viii. Functional suitability

This quality attribute means the degree to which a software solution or a digital product offers functions that satisfy the predefined needs when utilized under certain conditions. This characteristic includes the following sub characteristics:

A. **Functional completeness** implies the degree to which the system's functionality covers all outlined tasks and user's goals.

B. **Functional correctness** defines the degree to which a software solution offers the correct results with the required precision degree.

C. **Functional appropriateness** defines the degree to which the product's features allow for the accomplishment of certain tasks or the achievement of particular goals.

ix. Maintainability

This quality attribute defines the degree of efficiency to which a software solution can be modified for its improvement or adaptation to the evolving requirements or changes in the environment. Take a look at the following sub-characteristics of this quality attribute:

A. **Reusability** defines the degree to which a system component or an asset can be utilized on several systems or in building other components or assets.

B. **Modifiability** means the degree to which a software system can be effectively modified without causing defects or bugs or decreasing the quality of the existing system.

C. **Testability** shows the degree to which test criteria can be used for the software system, and tests can be performed to determine whether these criteria have been satisfied.

x. Interoperability

When we speak about the software system's interoperability, we mean its ability to communicate or exchange data seamlessly between different operating systems, databases, and protocol conditions. The most common interoperability issues are as follows:

A. Legacy internal systems.

B. Different formats of data belonging to similar external systems.

C. Different API versions in external systems.

D. Poor quality or lack of standards for external systems.

E. You can improve interoperability by creating well-designed external interfaces and system standardization.

xi. Performance efficiency

This software quality attribute shows a software product's performance relative to the number of resources applied under predefined conditions. When checking the performance efficiency, consider the following sub-characteristics:

A. **Time behaviour** means the degree to which an operating software solution's response and processing time meet the predefined requirements.

B. **Capacity** is the degree to which the maximum limit of a software product or parameters satisfies the established requirements.

C. **Resource utilization** defines the degree to which the amount of resources utilized by a working software system satisfies the main requirements.

xii. Security

This attribute of product quality refers to the degree to which a software system safeguards the information or data so that users or other systems have the degree of access to these data based on the authorization level. The main sub-characteristics of this software quality attribute are as follows:

A. **Confidentiality** is the degree to which a software system guarantees that only authorized users can access the data.

B. **Integrity** is the degree to which a software system prevents unauthorized access to a program or data.

C. **Authenticity** is the degree to which the identity of the user or resource can be verified when required.

D. **Accountability** means the degree to which the actions of a particular entity can be monitored and tracked uniquely to this entity.

LITERATURE REVIEW

A. Sedaghatbaf et al. illustrated that the challenging to create a software architecture that complies with all quality standards. It is required to statistically analyses quality characteristics on the architectural model to see if the criteria have been met. How to modify the design based on the assessment findings, how to examine calculation uncertainties, and how to address disputes that may arise between the qualities preferences of stakeholders are all problems that a competent evaluation process should have adequate solutions for. We present SQME as a platform for automatically evaluating software architecture models in this study. The framework employs evidence theory to handle uncertainty, evolutionary algorithms to develop the design, and EV/TOPSIS to decide which trade-offs to make. A case

study is conducted to verify the framework's applicability, and a software tool is created to aid in the assessment process [4].

P. Bachmann et al. illustrated that the several parts of the methodology must be in place for a software architecture design process to meet quantitative performance standards. Prior to determining if the intended architecture can meet them, there must be a method for expressing the criteria for quality attribute requirements. Second, there has to be a mechanism for integrating different the information based on quality characteristics so that the design approach won't have to be aware of every single variety of quality qualities that exist. The interconnections here between quality qualities must also be managed in some manner so that either the criteria may be met and the ones which cannot be met can be detected. A "reasoning framework" is a structure that the authors define as a prefabrication of quality attribute information. Concrete quantitative performance scenarios are used to specify the constraints that the architectural must meet. Every reasoning framework offers tools that will change the architecture in light of a certain understanding of quality attributes. The authors differentiate between an architectural model and a quality attribute model inside a reasoning frameworks and describe the acts a reasoning framework makes as essential architectural transformations. In order to handle competing needs, the process of discovering interconnections across reasoning frameworks is finally started. In order to maintain a functioning technique while the unresolved challenges of designing to meet quality attribute constraints are being addressed, the deployment of reasoning frameworks is integrated inside an existing design and architecture methodology [5].

M. Svahnberg et al. illustrated that the essential to have a consistent software architecture that has been known by all developers and according to which all modifications to the system comply in order to maintain a software system's qualities throughout evolution and alter the performance parameters as the needs change. This software architecture may be made in advance, but it must also updated to take into account changes in the domain and, therefore, the necessities of the programmed. The selection of which data structures to use is often made informally. To the highest of our knowledge, there isn't much information available on the specific qualities that various architectural techniques encourage or hinder. The approach sometimes used quantify the perceived support that various software architectures provide for various quality qualities being presented in this article as an empirical research. This thus makes it possible to determine with knowledge which architecture and design option best matches the combination of admirable characteristics demanded by the system being created [6].

M. Alenezi discussed that in recent years, software architecture has emerged as a critical area within software engineering. The development of new advanced technologies to begin addressing architectural design as an engineering subject has advanced significantly. Any engineering field examines the level as a fundamental component. The software architecture's quality characteristics may be quantified to provide useful information about the architecture. Additionally, it will assist professionals including architects in deciding which alternative architecture best suits their requirements. This discovery provides the path for academics to begin looking into techniques to gauge the qualities of software architecture. For this area of software engineering, these attributes must be measured. This study examines the stability and comprehensibility of software architecture, as well as a number of metrics that have a consequence on them and a survey the related literature [7].

J. Lourenço et al. illustrated that the over forty years, relational databases have been the leading model for data storage, retrieval and management. However, due to increasing needs for scalability and performance, alternative systems have emerged, namely NoSQL

technology. The rising interest in NoSQL technology, as well as the growth in the number of use case scenarios, over the last few years resulted in an increasing number of evaluations and comparisons among competing NoSQL technologies. While most research work mostly focuses on performance evaluation using standard benchmarks, it is important to notice that the architecture of real world systems is not only driven by performance requirements, but has to comprehensively include many other quality attribute requirements. Software quality attributes form the basis from which software engineers and architects develop software and make design decisions. Yet, there has been no quality attribute focused survey or classification of NoSQL databases where databases are compared with regards to their suitability for quality attributes common on the design of enterprise systems. To fill this gap, and aid software engineers and architects, in this chapter, we survey and create a concise and up-to-date comparison of NoSQL engines, identifying their most beneficial use case scenarios from the software engineer point of view and the quality attributes that each of them is most suited [8].

R. Mzyk and S. Paszkiel illustrated that the technology advances, programmers can now handle a broad variety of corporate needs and solve increasingly complex difficulties. Larger demands place more pressure on software developers to provide scalable, reusable, and adaptable systems. Conscious trade-offs made during the software development process may highlight some of its excellent characteristics, but it is also extremely simple to create a piece of software that is essentially impractical to be, for example, scalable. Even if there isn't yet a perfect architecture, it's still worthwhile to search for answers that will ultimately help us more than they harm us[9].

F. Ghasemi et al. illustrated that the Increasing level of personal and social health and life expectancy has resulted in the growth of aged people population. Elderly care is an essential and costly issue in any society that should be addressed in researches. Elderly care has faced some problems such as elderly solitary, trained caregiver's insufficiency, and increasing cost of late diagnosis of diseases and accident. Regarding to these problems, smart home technology can be used as an efficient solution. It provides an expertise, long-term, and low-cost care that empowers elderly to have an independent life. In this paper, we propose an architecture for a health-care system in a smart home. In this architecture, rapid and timely diagnosis of environmental incidents and health risks causes reduction in costs of health care and relief. Given to the vital aspect of health-care systems, the proposed architecture components and solutions are presented to meet quality attributes such as availability, performance, security, and interoperability. The proposed architecture evaluation is based on the Architecture Tradeoff Analysis Method (ATAM) scenario-based approach. ATAM is a method for analysis and evaluation of software systems architecture. And quality attributes scenarios are examined to meet quality requirement [10].

E. Folmer and J. Bosch Over the importance of software architecture in meeting a system's quality needs has come to the attention of the software engineering community throughout the years. The software architecture of a software system greatly influences its quality characteristics. The software engineering community has created several tools and methods recently that enable the design of quality characteristics, such performance or maintainability, at the level of software architecture. This design strategy, in our opinion, may be used to improve usability as well as "conventional" quality factors like performance and maintainability. This survey investigates if such a design strategy is feasible. The state of the art is examined from the viewpoint of a software architect. Exist any design techniques that permit usability design at the architectural level? Exist any tools for evaluating designs in terms of how well they promote usability? Describe usefulness. These three study topics are

shown in a framework. All design phases should be driven by usability, however existing usability engineering practices fall short of meeting this objective. Our investigation reveals that there are no design methodologies or evaluation tools that enable architectural design for usability [11].

T. Wamg and B. Li illustrated that the cost of software maintenance goes up, software quality goes down, software performance goes down, etc. as software architecture deteriorates. Therefore, it is very important to develop a workable method for assessing software architecture in order to quickly identify and stop the erosion of software design. We discover via empirical research that one of the major factors contributing to the degradation of software architecture is its capacity to evolve. In order to address the aforementioned issues with software architecture evolution, we offer a method in this work for analyzing SA evolvability based on numerous architectural characteristic assessments. Our method entails the following steps: first, we suggest four corresponding architectural attributes based on the evolutionary process; second, these attributes are measured based on fundamental data and dependency data; third, SA evolvability is measured based on multiple architectural attribute measurements. To test the efficacy of our strategy, we run trials on thirteen open-source Java projects. According to the experimental findings, our method can accurately portray the SA evolvability from the following two perspectives: The mix of characteristics may indicate the composite SA evolvability, whereas a single attribute can reflect a particular facet of the SA evolvability. By integrating data and evolutionary activities, we can also identify the reasons for SA erosion. We also put forward evolutionary suggestions to increase SA evolvability [12].

M. Svahnberg et al. illustrated that the software architecture that is acknowledged by all developers and to which all modifications to the system follow is required to maintain the qualities of a software platform throughout development and to alter the quality characteristics as the needs vary. This software architecture may be designed in advance, but it must also be modified because when program's domain and, by extension, its needs change. It may be challenging to design a software architecture for a system or a component of a system that satisfies the appropriate quality standards. We provide a decision-support technique in this research to make it easier to comprehend potential software system architectures. To aid with this effort, we provide a technique that is flexible in terms of both the pool of prospective architectural choices and the quality qualities pertinent to the system's domain. Using a multi-criteria decision method, the method develops a support framework that enables comparison of various software architecture candidates for a given software quality attribute and vice versa. It then makes use of this support framework to come to an agreement on the advantages and disadvantages of the various software architecture candidates and to boost confidence in the final architectural style decision [13].

J. Horcas et al. illustrated that the structure elicitation phase, quality factors are important. Software energy efficiency and sustainability are extremely vital attributes that may be utilized as qualifying criteria to choose between various design or implementation options. Usually, other non-functional demands, including performance, are in opposition with energy efficiency. Using a technique described in this article, engineers may automatically create functional quality attribute settings that are as effective and effective as possible. The behavioral characteristics that must be introduced into a software architecture to satisfy a certain quality attribute are known as the functional quality attributes. Method: To determine the design and execution variations of quality characteristics and how the various configurations affect both energy efficiency and performance, quality attributes are characterized. Each specified quality parameter has a specific use model. With the intention

of explicitly thinking about it, the variability of quality characteristics, together with the energy efficiency and effectiveness experiment outcomes, are expressed as a constraint fulfilment issue. The chosen functional quality criteria are then manually combined to provide an optimal configuration for the chosen goal function. Results: By utilizing our method to conduct a comprehensive study of the energy consumption and performance of several options for functional quality criteria, software companies may enhance the energy efficiency and/or effectiveness of their applications. We quantify the advantages of using our method and talk about the validity risks. The methodology described in this study will assist software developers in creating more energy-efficient software while being conscious of how their choices impact other quality factors, such as performance [14].

A. Banijamali et al. illustrated that the interest in the confluence of cloud computing and the Internet of Things has grown over the last several years (IoT). There is no systematic study of this understanding, despite the fact that software systems in this field have attracted academics who have created a considerable body of knowledge on software architectural designs. The goal of this research is to discover and incorporate cutting-edge architectural components, such as design patterns, styles, viewpoints, and quality aspects, in the convergence of cloud computing and IoT. The style that is most commonly used in this situation is service-oriented architecture. Scalability, timeliness, and security were the quality qualities that garnered the most research out of all those that were relevant. Nine cross studies were also used together explore how various quality criteria and various applications relate to architectural patterns, styles, perspectives, and assessment methods. Our results suggest that this field is seeing an increase in research on software applications. Despite the fact that there were few studies that gave industrial appraisals, the industry needs additional research-based and empirically verified design frameworks to guide software engineering in this area. This report highlights topics for further study while giving the broad overview of the subject [15].

DISCUSSION

There are several methods for assessing a software architecture. The use of interrogation tactics is common in qualitative assessments. Scenario, questionnaire, and checklist are the three fundamental types of inquiry. Numerous scenarios may be built for each quality need. The various situations are designed to showcase the quality characteristic as much as possible while highlighting any potential shortcomings in the system. A questionnaire is a list of generic inquiries that may be used to evaluate the effectiveness of the system. Older projects' typical circumstances might be compiled into a questionnaire. It may be necessary to employ various surveys or situations to be able to respond to these questions. In the assessment process, scenarios are created, and prior to the project's launch, questionnaires and checklists should be in place. These might be based on knowledge of typical general inquiries from earlier studies. It is challenging to evaluate a software architecture quantitatively since additional system knowledge is often needed. There are two fundamental methods: metrics and simulation. Experiments and prototypes belong to the same category as simulation. Metrics are being used more and more to assess a system's design or execution, including whether the system meets its requirements, is ready for delivery, adheres to budgets, etc. The issue with utilizing metrics to evaluate software architecture is that they demand a lot of information, maybe so much detail that we are really assessing a design rather than an architecture. We think that in order to apply metrics for the essential quality aspects, it will sometimes be required to define the affected components in great detail. Typically, a skeleton of the building is constructed for simulations and prototypes. The elements and their connections are put into practice. Which quality qualities are to be evaluated determines a

great deal of the implementation's specifics and which components need to be emphasized. In Table 1 shown the Properties of the Evaluation Approaches.

Table 1: Represented that the Properties of the Evaluation Approaches.

Review Method	Generality	Level of Detail	Phase	What is evaluated
Scenarios	system-specific	medium	middle	artifact
Questionnaire	general	coarse	early	artifact, process
Checklist	domain-specific	varies	middle	artifact, process
Metrics	general or domain-specific	fine	middle	artifact
Simulation, Prototype, Experiment	domain-specific	varies	early	artifact

CONCLUSION

Software engineering is a branch of science that deals with the creation of software. As a field, engineering is focused on ensuring things function properly by using theories, approaches, and tools as needed. We provide a brief introduction to software engineering in order to help you comprehend the idea of software architecture and help you realise where it fits in the software development process. The application of software architecture is then described in this context. In a development project, software development is often carried out. The project determines process requires from a client or target market. After that, it evaluates, creates, and implements a software system to address these needs. The individuals working on the development project do a variety of activities that result in the production of various information, such as requirements and specifications, prototypes, and implementations. The individuals working on the development project often adhere to a software development process to make the jobs simpler and more organized. A process for developing software outlines the tasks that must be completed and the appropriate times throughout the development process. The functions that should be present in the development organization may also be characterized by a process. Engineers of requirements, programmers, software architects, testers, and managers are a few examples of jobs. The procedure outlines the responsibilities and obligations of each function. The number of employees in each function may depend greatly the extent to which the development has come.

REFERENCES

[1] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, en I. Meedeniya, “Software Architecture Optimization Methods: A Systematic Literature Review”, *IEEE Trans. Softw. Eng.*, vol 39, no 5, bll 658–683, Mei 2013, doi: 10.1109/TSE.2012.64.

[2] D. Tiwari *et al.*, “Metrics driven architectural analysis using dependency graphs for C language projects”, 2019. doi: 10.1109/COMPSAC.2019.00025.

[3] C. Stoermer, L. O’Brien, en C. Verhoef, “Moving towards quality attribute driven software architecture reconstruction”, 2003. doi: 10.1109/WCRE.2003.1287236.

[4] A. Sedaghatbaf en M. A. Azgomi, “SQME: a framework for modeling and evaluation of software architecture quality attributes”, *Softw. Syst. Model.*, vol 18, no 4, bll 2609–2632, Aug 2019, doi: 10.1007/s10270-018-0684-3.

[5] F. Bachmann, L. Bass, M. Klein, en C. Shelton, "Designing software architectures to achieve quality attribute requirements", *IEE Proc. - Softw.*, vol 152, no 4, bl 153, 2005, doi: 10.1049/ip-sen:20045037.

[6] M. Svahnberg en C. Wohlin, "An Investigation of a Method for Identifying a Software Architecture Candidate with Respect to Quality Attributes", *Empir. Softw. Eng.*, vol 10, no 2, bll 149–181, Apr 2005, doi: 10.1007/s10664-004-6190-y.

[7] M. Alenezi, "Software Architecture Quality Measurement Stability and Understandability", *Int. J. Adv. Comput. Sci. Appl.*, vol 7, no 7, 2016, doi: 10.14569/IJACSA.2016.070775.

[8] J. R. Lourenço, B. Cabral, P. Carreiro, M. Vieira, en J. Bernardino, "Choosing the right NoSQL database for the job: a quality attribute evaluation", *J. Big Data*, vol 2, no 1, bl 18, Des 2015, doi: 10.1186/s40537-015-0025-0.

[9] M. Höhnerbach en P. Bientinesi, "Accelerating AIREBO: Navigating the Journey from Legacy to High-Performance Code", *J. Comput. Chem.*, 2019, doi: 10.1002/jcc.25796.

[10] F. Ghasemi, A. Rezaee, en A. M. Rahmani, "Structural and behavioral reference model for IoT-based elderly health-care systems in smart home", *Int. J. Commun. Syst.*, vol 32, no 12, bl e4002, Aug 2019, doi: 10.1002/dac.4002.

[11] E. Folmer en J. Bosch, "Architecting for usability: a survey", *J. Syst. Softw.*, vol 70, no 1–2, bll 61–78, Feb 2004, doi: 10.1016/S0164-1212(02)00159-0.

[12] T. Wang en B. Li, "Analyzing Software Architecture Evolvability Based on Multiple Architectural Attributes Measurements", in *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*, Jul 2019, bll 204–215. doi: 10.1109/QRS.2019.00037.

[13] M. SVAHNBERG, C. WOHLIN, L. LUNDBERG, en M. MATTSSON, "A QUALITY-DRIVEN DECISION-SUPPORT METHOD FOR IDENTIFYING SOFTWARE ARCHITECTURE CANDIDATES", *Int. J. Softw. Eng. Knowl. Eng.*, vol 13, no 05, bll 547–573, Okt 2003, doi: 10.1142/S0218194003001421.

[14] J. M. Horcas, M. Pinto, en L. Fuentes, "Variability models for generating efficient configurations of functional quality attributes", *Inf. Softw. Technol.*, 2018, doi: 10.1016/j.infsof.2017.10.018.

[15] A. Banijamali, O.-P. Pakanen, P. Kuvaja, en M. Oivo, "Software architectures of the convergence of cloud computing and the Internet of Things: A systematic literature review", *Inf. Softw. Technol.*, vol 122, bl 106271, Jun 2020, doi: 10.1016/j.infsof.2020.106271.

CHAPTER 8

FUNCTIONAL ASPECTS OF SOFTWARE ARCHITECTURE

Ramya Vathsala C.V, Assistant Professor
Department of Computer Science and Engineering, Presidency University, Bangalore, India
Email Id- ramya.v@presidencyuniversity.in

ABSTRACT:

The ideas of styles and patterns came forth as a result of some organizational strategies for software pieces demonstrating desirable traits, such as providing comprehensive descriptions of software at various levels of abstraction and serving as a response to numerous system needs. This fact is the outcome of software engineers often and use these software element groups when they create systems that are more or less independent from each other. The ability to choose appropriate methods of defining software architectures has become more crucial as the magnitude and complexity of software systems continue to grow. The function of patterns and patterns in the field of software architecture will be covered in this essay. Most of the talks will Centre on differing opinions on the resources already available. To accommodate style and pattern-based architectural descriptions, problem statements will also be addressed via a hypothetical automated system, among other approaches.

KEYWORDS:

Computer Science, Software Engineering, Software Design, Software Architecture, SDLC.

INTRODUCTION

In light of the well-known features these components imply, seasoned software designers and engineers often duplicate the same design of elements. The selected architecture has really been described using more or less informal terminology, such as a Client and Server structure or a Pipes and Filter structure. Styles and patterns are often regarded as certain approaches of arranging software components. Styles and patterns have increasingly affected software engineering and development during the 1990s. This may be attributable to the fact that software is consistently expanding in both size and complexity, necessitating the requirements for appropriate software design. However, the establishment of communities and conventions, as well as the publication of books and catalogues, on the theories and their impact, have represented the major breakthroughs. The formalization of the hitherto informal descriptions, which has huge repercussions, is one contribution of such work. Examples have included the usage of standard terminology, the ability to clarify a structure's meaning, and the expansion of specific design alters the composition[1].

Reusability is a particular sort of desirable quality, and certain structures could well be uniquely related to it. One other advantage that applies to all designs is that they are greater likely to be reused if they are more clearly presented. For a real system, this is also true. The likelihood of reusing network elements is further increased by a well-documented system, particularly in terms of a well-describing architecture. The topic of styles and patterns has important because they provide perspective on software design that address issues like statement clarity and programmed characteristics. However, the function is not absolutely obvious. This is due, in part, to the ambiguous definition of "software architecture." A system's general description may be found in the software architecture. The following

definition of software architecture is more formal: The structure of the system's structures, which are made up of software applications, their externally observable attributes, and their interactions make up the software architecture of a programming or computer system [2].

This pretty thorough explanation still leaves certain things up for debate, including what degrees of transparency an architecture should provide. Another topic is whether an infrastructure is always tied to a system, as well as the correlation between architecture and design and system components. This essay aims to engage in a debate on the usefulness of styles and patterns in software architecture. Naturally, such conversations will be informed by the rather vague definitions. How thoroughly styles and patterns could support architectural requirements is one example of this. On the other hand, another illustration is the level of detail that should be presented by patterns and styles. It appears that nobody agrees that architectural descriptions shouldn't at the very least take system implementation characteristics into account. The talks will mostly be conducted in light from already information released. Still, issues will be listed, perhaps in the context of a fictitious interface definition system that supports style and pattern. Such a technology ought to serve as a laboratory where various qualities may be paired to satisfy various objectives [3].

Comparisons between Styles and Patterns

This section will examine the definition, similarities, and differences of styles and patterns.

i. Similarities

Both styles and patterns have their roots in well-known, useful structures from the world of software engineering. Both styles and patterns have been progressively formalized in the 1990s due to the creative and advantageous perspective on software design they imply. They all deal with a limitation on how a set of software components may be organized or behave when the components are participants in an organization. They are connected to software design in this way. This chapter will discuss the historical use of abstraction approaches in software system characterization. They draw attention to the differences between modern software system designers and earlier approaches that made data structures the main instrument. It also covers the usage of patterns to describe the design of a graphical editor. In the past, classes like "Drawing," "Figure," or "Handle" should have been used to represent the architecture, but the introduction of patterns has given these terms new significance. As a result, styles and patterns focus on how these components are arranged rather than taking into account program elements like data structures, classes, or higher level components. Their interactions are defined by the structures and behaviors they engage in.

ii. Trade-off

The contrasts between this viewpoint and the one discussed before will mostly concern scale. Architectural patterns, design patterns, and idioms are the three primary divisions of patterns. Idioms often refer to implementation issues that are particular to programming languages, however this is not directly relevant to the article. Both architectural and design patterns often focus on two distinct levels. Architectural patterns apply to organizational frameworks pertaining to the whole system, while design patterns focus on specific subsystems or parts of a system. Additionally, contrast architectural designs and stylistic trends. This comparison, for instance, highlights level inequalities. Styles only represent the highest level, but patterns often describe a variety of scales. Smaller sized patterns are the building blocks of patterns. A discussion on how architectures are transformed to meet various types of needs is included. Here, changes are done at several levels of architecture, where those levels are classified according to styles, architectural patterns, and design patterns. The criteria used for

categorization depend on how much changes have an impact on an architecture. In this case, a style correlates to an architecture's top level; changes based on styles have an impact on the whole architecture. On the other hand, changes to architectural patterns only affect certain parts of the architecture rather than the whole structure. Architectural patterns are not predominate, despite claims to the contrary. Design patterns are minor components of an architecture, and changes made to them have an equivalent small-scale influence. Design patterns are also low-level and object-oriented, which is related to them. This has the connotation that method invocation is the main means of interaction. Contrary to architectural styles or patterns, where contact may take any form, for as via files or events.

Software Attributes:

Developers of critical systems are responsible for identifying the requirements of the application, developing software that implements the requirements, and for allocating appropriate resources (processors and communication networks). It is not enough to merely satisfy functional requirements. Critical systems in general must satisfy security, safety, dependability, performance, and other, similar requirements as well. Software quality is the degree to which software possesses a desired combination of attributes such that reliability, interoperability.

Addressed Quality Attributes:

There are different schools/opinions/traditions concerning the properties of critical systems and the best methods to develop them:

i. Performance from the tradition of hard real-time systems and capacity planning.
ii. Dependability from the tradition of ultra-reliable, fault-tolerant systems.
iii. Security from the traditions of the government, banking and academic communities.
iv. Safety from the tradition of hazard analysis and system safety engineering.

Systems often fail to meet user needs that is lack quality when designers narrowly focus on meeting some requirements without considering the impact on other requirements or by taking them into account too late in the development process.

Software Quality Attribute Trade-offs:

Designers need to analyse trade-offs between multiple conflicting attributes to satisfy user requirements. The ultimate goal is the ability to quantitatively evaluate and trade off multiple quality attributes to arrive at a better overall system. We should not look for a single, universal metric, but rather for quantification of individual attributes and for trade-off between these different metrics, starting with a description of the software architecture.

Software Architecture Design Non-Functional Requirements:

Introduction(Functional Requirements):

Non-functional requirements are different from functional requirements in many ways.

Functional Requirements are:

A. Describe what a system should do.

B. Mostly come from the customer.

C. Can be described by a Use Case model and set of formal "shall" statements.

Non-Functional Requirements:

A. Are not related to individual use cases, but rather to system-wide attributes like performance.

B. Can be complete show-stoppers if not met. # Often conflict with each other, requiring trade-offs.

C. Are more architecture-dependent than functional requirements.

D. Are often determined by the architect and stakeholders within the organisation.

E. Can be described in terms of standard Quality Attributes.

Quality Attributes:

A. Usually, business considerations determine the qualities that must be accommodated in a system architecture.

B. Too often, functionality overrides maintainability, portability, scalability, and other factors determining the long-term success of a project.

C. Functionality and quality attributes are orthogonal, since a given functionality can be achieved by many different architectures.

D. Quality requirements depend on the system architecture more than on the functional requirements.

There are three main categories of quality attributes:

A. System Qualities: availability, modifiability, performance, security, testability, usability, others.

B. Business Qualities: time to market, cost and benefit, product lifetime, target market, roll-out schedule, integration, others.

C. Architectural Qualities: conceptual integrity, correctness and completeness.

A quality attribute scenario has six parts, shown in the Figure 1:

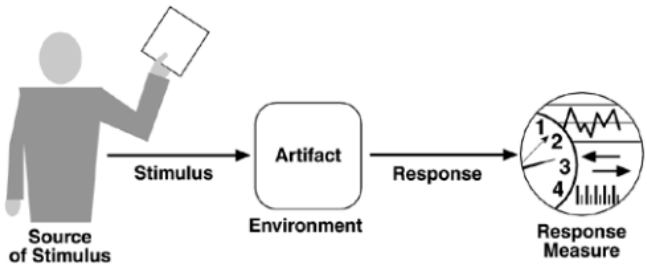

Figure 1: Represented that the Six Part of Software Attributes.

A. Source of Stimulus the entity generating the stimulus. Could be an actor, an actuator, a sensor, and so on.

B. Stimulus: a condition arriving at a system. Includes faults, stated intentions by actors, and so on.

C. Environment the conditions surrounding the stimulus. Might be normal operation, degraded operation, overload, and so on.

D. Artifact the part or parts of the system stimulated.

E. Response the response the system takes to the stimulus.

F. Response Measure how the response can be measured and test.

System Quality Attributes:

i. Availability:

The availability attribute is concerned with system failures. Faults are problems that are corrected or masked by the system. Failures are uncorrected errors that are user-visible.

Availability = [mean time to failure] / ([mean time to failure] + [mean time to repair])

ii. Modifiability:

The modifiability quality is concerned with what can change, when changes are made, and who makes the changes.

iii. Performance:

The performance quality is concerned with response times and similar measures for various events.

iv. Security:

A. Non-repudiation

B. Confidentiality

C. Integrity

D. Assurance or authenticity

E. Availability (no denial of service)

F. Auditing

v. Testability:

The testability attribute is concerned with detecting failure modes. Typically, 40% of the cost of a large project is spent on testing. This means architectural support for testing that reduces test cost is time well spent. We need to control the internal state of and inputs to each unit, then observe the corresponding output of that unit.

vi. Usability:

A. How easy it is to learn the features of the system

B. How efficiently the user can use the system

C. How well the system handles user errors

D. How well the system adapts to user needs

E. To what degree the system gives the user confidence in the correctness of its actions.

vii. Business Quality Attributes:

A. Time to Market: architectural reuse affects development time.

B. Cost and Benefit: in-house architectural expertise is cheaper than outside expertise.

C. Projected Lifetime of the System: long-lived systems require architectures that are modifiable and scalable.

D. Targeted Market: architecture affects what platforms will be compatible and incompatible with the system.

E. Roll-out Schedule: if functionality is planned to increase over time, the architecture needs to be customisable and flexible.

F. Integration with Legacy Systems: the architecture of the legacy system being integrated will influence the overall system's architecture.

viii. Architectural Quality Attributes:

A. Conceptual Integrity is the underlying vision or theme unifying the components and their interactions. The architecture should do similar things in similar ways.

B. Correctness and Completeness is concerned with checking the architecture for errors and omissions.

C. Build ability is concerned with the organization's capabilities to actually construct the architecture in question.

LITERATURE REVIEW

H. Tao et al. illustrated that the Software engineering's measurement of software trustworthiness is a significant area of study that is tremendously helpful for evaluating software quality. In this article, we provide a theoretical programming method for methodologies the trustworthy amount to each sub-attribute of a software character and then maximizing the trustworthy degree of this property given certain constraint circumstances. We look at a few requisite or sufficient criteria for analyzing this mathematical programming issue. Likewise, a polynomial allocation methodology is provided for generating this mathematical programming's optimal solution. A last illustration of the usefulness of this effort is provided. By modifying the dependable level of each sub-attribute at the same cost, the information gained here may be used to optimize the productivity and quality [4].

B. Belinda et al. illustrated that the use of quality software is of importance to stakeholders and its demand is on the increase. This work focuses on meeting software quality from the user and developer's perspective. After a review of some existing software-quality models, twenty-four software quality attributes addressed by ten models such as the McCall's, Boehm's, ISO/IEC, FURPS, Dromey's, Kitchenham's, Ghezzi's, Georgiadou's, Jamwal's and Glibb's models were identified. We further categorized the twenty-four attributes into a group of eleven main attributes and another group of thirteen sub-attributes. Thereafter,

questionnaires were administered to twenty experts from fields including Cybersecurity, Programming, Software Development and Software Engineering. Analytic Hierarchy Process (AHP) was applied to perform a multi-criteria decision-making assessment on the responses from the questionnaires to select the suitable software quality attribute for the development of the proposed quality model to meet both users and developer's software quality requirements. The results obtained from the assessment showed Maintainability to be the most important quality attribute followed by Security, Testability, Reliability, Efficiency, Usability, Portability, Reusability, Functionality, Availability and finally, Cost[5].

B. Jereb et al. discussed that this chapter represents a step towards defining attributes of software and the metrics of these attributes by creating a framework of them. By standardization both, the attributes and the metrics, the description of software makes useful in its life cycle processes. Some of the proposed attributes and their metrics have been in use for decades, some are under development and some are proposed in this article for the first time. However, the proposed system is unique and it classifies attributes into four groups according to their properties such that task system, software, hardware and data surrounding and time limits and history of execution. Which of the attribute groups and which individual attribute within a group we need to consider depend on the nature of the software and related demands. The attribute framework should be tailored to the needs of any software project [6].

K. Elish et al. represented his study the process of improving the design of existing code by changing its internal structure without affecting its external behavior. Refactoring tends to improve software quality by improving design, improving readability and reducing 'bugs'. There are many different refactoring methods, each having a particular purpose and effect. Consequently, the effect of refactoring methods on software quality attributes may vary. Moreover, it is often unclear to software designers how to use refactoring methods to improve specific quality attributes. In this paper, we propose a classification of refactoring methods based on their measurable effect on software quality attributes. This, in turn, helps software designers choose appropriate refactoring methods that will improve the quality of their designs, based on the design objectives. It also enables them to predict the quality drift caused by using particular refactoring methods [7].

L. Lavazza et al. illustrated that the technical debt is currently receiving increasing attention from practitioners and researchers. Several metaphors, concepts, and indications concerning technical debt have been introduced, but no agreement exists about a solid definition of technical debt. We aim at providing a solid basis to the definition of technical debt and the way it should be quantified. We view technical debt as a software quality attribute and therefore we use Measurement Theory, the general reference framework for the quantification of attributes, to define technical debt and its characteristics in a rigorous way. We show that technical debt should be defined as an external software quality attribute. Therefore, it should be quantified via statistical and machine-learning models whose independent variables are internal software quality attributes. Different models may exist, depending on the specific needs and goals of the software product and development environment. Also, technical debt is a multifaceted concept, so different kinds of technical debt exist, related to different quality attributes, such as performance, usability, and maintainability. These different kinds of technical debt should be evaluated individually, so one can better focus on the specific quality issues that need to be addressed. We show that, to provide it with a rigorous basis, technical debt should be considered and measured as an external software attribute [8].

A. Ampatzoglou et al. illustrated that the Design patterns are used in software development to provide reusable and documented solutions to common design problems. Although many

studies have explored various aspects of design patterns, no research summarizing the state of research related to design patterns existed up to now. The research questions of this study deal with if design pattern research can be further categorized in research subtopics, which of the above subtopics are the most active ones and what is the reported effect of GoF patterns on software quality attributes. The results suggest that design pattern research can be further categorized to research on GoF patterns formalization, detection and application and on the effect of GoF patterns on software quality attributes. Concerning the intensity of research activity of the abovementioned subtopics, research on pattern detection and on the effect of GoF patterns on software quality attributes appear to be the most active ones. Finally, the reported research to date on the effect of GoF patterns on software quality attributes are controversial; because some studies identify one pattern's effect as beneficial whereas others report the same pattern's effect as harmful [9].

H. Henif et al. illustrated that the Software development aims to produce software systems that satisfy two requirement categories: functional and quality. One aspect of software quality is nonfunctional attributes (NFAs), such as security, performance, and availability. Software engineers can meet NFA requirements by applying suitable guidelines during software development. However, this process is complicated by the different effects of different guidelines on NFA quality and the relationships among the guidelines themselves. Thus, finding a suitable set of guidelines is not straightforward. This article introduces a step-by-step approach that gives software engineers a suitable guideline set to apply to improve NFA quality during the software development life cycle. The approach manages the effects different guidelines have on both the attributes and the relationships among the guidelines [10].

J. Moses et al. illustrated that the most external software quality attributes are conceptually subjective. For example, maintainability is an external software quality attribute, and it is subjective because interpersonally agreed definitions for the attribute include the phrase 'the ease with which maintenance tasks can be performed'. Subjectivity clearly makes measurement of the attributes and validation of prediction systems for the attributes problematic. In fact, in spite of the definitions, few statistically valid attempts at determining the predictive capability of prediction systems for external quality attributes have been published. When validations have been attempted, one approach used is to ask experts to indicate if the values provided by the prediction system informally agree with the experts' intuition. These attempts are undertaken without determining, independently of the prediction system, whether the experts are capable of direct consistent measurement of the attribute. Hence, a statistically valid and unbiased estimate of the predictive capability of the prediction system cannot be obtained (because the experts' measurement process is not independent of the prediction system's values). In this paper, it is argued that the problem of subjective measurement of quality attributes should not be ignored if quality is to be introduced into software in a controlled way. Further, it is argued that direct measurement of quality attributes should be encouraged and that in fact such measurement can be quantified to establish consistency using an existing approach. However, the approach needs to be made more accessible to promote its use. In so doing, it would be possible to decide whether consistent independent estimates of the true values of software quality attributes can be assigned and prediction systems for quality attributes developed [11].

A. Aydin illustrated that the developing software systems to meet user-demanded functionality is critical. Achieving the design goals by providing the needed functionality is a necessary task, and it is about figuring out a proper set of quality attributes and implementing each one by reflecting a complete set of quality attributes. This study presents popular quality

attributes of crisis software systems by conducting a literature review. Each crisis software system has been studied by concentrating on crisis management phases where the system is used, design purposes, and the data processing style. The findings of this research shed light on the crisis software development process by presenting a quality attribute-oriented perspective, addressing design challenges, and recommending to developers remedies to handle challenges [12].

A. Kaur stated that the Code smells indicate problems in design or code which makes software hard to change and maintain. It has become a sign of software systems that cause complications in maintaining software quality. The detection of harmful code smells which deteriorate the software quality has resulted in a favorable shift in interest among researchers. Therefore, a significant research towards analyzing the impact of code smells on software quality has been conducted over the last few years. This study aims at reporting a systematic literature review of such existing empirical studies investigate the impact of code smells on software quality attributes. The results indicate that the impact of code smells on software quality is not uniform as different code smells have the opposite effect on different software quality attributes. The findings of this review will provide the awareness to the researchers and a practitioner regarding the impact of code smells on software quality. It would be more advantageous to conduct further studies that consider less explored code smells, least or not investigated quality attributes, involve industry researchers and use large commercial software systems [13].

A. Anand and J. Banshal discussed that the quality of any product or service defines the agility of the product and its life cycle in dynamic environment. The demand of high quality becomes an imperative concern, when software is acting as a product or a service. Since the nature of the software is intangible and more complex, therefore the assurance of providing accurate results is anxiety for companies. The overall quality of the software is based upon many individual factors that makes software reliable, inclined and a long-lasting product in the marketplace. But how these factors can influence each other is significant to identify. Therefore, the purpose of this paper is to study the quality aspect of the software and analyses the interrelationship of impactful attributes. The analysis has been done through responses sought from software development teams in India. The questionnaire related to the software quality was administered to the sample population. Interconnection among impactful characteristics has been analyzed by using a qualitative technique called interpretive structural modelling. The procedure of applying ISM method has been automated and provided it as package ISM using software. Though ISM provides an organized modelling framework yet its results are considered less statistically significant. Therefore, it would be interesting to concatenate the present findings with the findings of any analytical methodology; which gives statistically significant results. The present proposal deals with the interpretation of the software quality attributes and their contextual relationship but with more effective and efficient manner. It can help management to understand the complexity of relationship amongst attributes more accurately and precisely. Since today is an era of automation, the manual part is being substituted so as to reduce the labor cost, improve safety, security and product quality to increase production. This study is, therefore, an effort and a helping hand in making the hassle free calculations for obtaining intermediate matrices and doing eventual calculations. The package created here can save precious time of users. Provides well-formatted and readable excel output files that make interpretation easier. Software is one such product which plays a significant role in this high-technological world, where each and every firm try their best to be on the top of the list of consumer's preference [14].

[15] The search for techniques to improve software quality and achieve robust, reliable, and maintainable software is ongoing. Refactoring, an approach that improves the internal structure of software without affecting its external behavior, is one method that aims to achieve better quality. Refactoring to patterns is another. The goal of this paper is to investigate whether refactoring to patterns improves software quality. This is done empirically by examining the metric values of external quality attributes for different software systems before and after refactoring to patterns is applied. We found no consistent improvement trends in the software quality attributes. This is because each refactoring to patterns technique has a particular purpose and effect, and hence affects software quality attributes differently [15].

Kahtan discussed that the existing software applications become increasingly distributed as their continuity and lifetimes are lengthened; consequently, the users' dependence on these applications is increased. The security of these applications has become a primary concern in their design, construction and evolution. Thus, these applications give rise to major concerns on the capability of the current development approach to develop secure systems. Component-Based Software Development (CBSD) is a software engineering approach. CBSD has been successfully applied in many domains. However, the CBSD capability to develop secure software applications is lacking to date. This study is an extension of the previous study on the challenges of the security features in CBSD models. Therefore, this study proposes a solution to the lack of security in CBSD models by highlighting the attributes that must be embedded into the CBSD process. A thorough analysis of existing studies is conducted to investigate the related software security attributes. The outcome analysis is beneficial for industries, such as software development companies, as well as for academic institutions. The analysis also serves as a baseline reference for companies that develop component-based software [16].

K. Elish et al. illustrated that the refactoring to patterns allows software designers to safely move their designs towards specific design patterns by applying multiple low-level refactoring's. There are many different refactoring to pattern techniques, each with a particular purpose and a varying effect on software quality attributes. Thus far, software designers do not have a clear means to choose refactoring to pattern techniques to improve certain quality attributes. This paper takes the first step towards a classification of refactoring to pattern techniques based on their measurable effect on software quality attributes. This classification helps software designers in selecting the appropriate refactoring to pattern techniques that will improve the quality of their design based on their design objectives. It also enables them to predict the quality drift caused by using specific refactoring to pattern techniques [17].

DISCUSSION

When teams are making choices about the system, rather than after development, integration, or deployment, software architecture facilitates analysis of system attributes. This timely analysis helps teams to decide if the methods they've selected will provide an acceptable solution, whether they're creating a new system, improving a current system, or updating a legacy system. Every step of the project is conceptually held together for all of its stakeholders by an effective architecture, which also promotes agility, time and money savings, and early design risk detection.

It might be difficult to create an effective architecture that supports both long-term objectives and quick product delivery for today's demands. Project delays, expensive rework, or worse might result from failing to properly identify, prioritise, and manage trade-offs among

architecturally relevant attributes. Effective continuous system evolution is made possible by an effective software architecture backed by agile architectural methods. Practices like these include analysing the deployed system for architecture conformance and documenting the architectural elements and relationships intended to achieve key qualities.

These practises are repeated to determine whether the architecture is fit for an organization's business and mission goals. These procedures, when carried out properly, provide for predictable product quality, fewer difficulties later on, and time and money savings during integration and testing, and cost-effective system development.

This is another reason why software architects still need to be developers without creating and testing anything, they can't comprehend or foresee the dynamics at play in a system. Software Architect needs to be more than just an honorarium for developers who have stopped working on new projects but yet have information that the company deems beneficial. The process of architecting requires a thorough understanding of a system in order to formulate valid hypotheses regarding quality qualities, as well as the skills necessary to create code and design tests.

CONCLUSION

To sum up this section, there are intuitive similarities between styles and patterns regardless of the notion that is usually used. These are based on limitations related to certain organizational structures of architectural elements. The issue of size is one of this paper's key points of interest. Styles are top-level structures, while design patterns are structures with a finer granularity. The top-level ideas of style and architectural patterns, respectively, are another point of divergence. Architectural patterns focus more on the challenges they solve than architectural styles do on the fundamental composition of a building. In the part after, this will be covered in more detail. A style guide or catalogue acts as a knowledge base that anybody with an interest in software design may use. They are particularly useful for describing architecture. Architecture descriptions are often made during a system's design or documentation phases. Specifically, objectives focus on topics like as organization and quality criteria. To explain an architecture's main components and behaviors, to design an architecture that satisfies the demands placed on it, or to assess how effectively an architecture relates to the demands. These collections must be supported by categories placed atop them in order to make design choices based on them more straightforward. This not only expedites the search for a good look, but also makes it simpler to distinguish between related options' similarities and differences.

REFERENCES

[1] L. Garcés, F. Oquendo, en E. Y. Nakagawa, “Software mediators as first-class entities of systems-of-systems software architectures”, *J. Brazilian Comput. Soc.*, 2019, doi: 10.1186/s13173-019-0089-3.

[2] M. El Bajta en A. Idri, “Identifying software cost attributes of software project management in global software development: An integrative framework”, 2020. doi: 10.1145/3419604.3419780.

[3] X. Zhang, W. Li, Z. Zheng, en B. Guo, “Optimized statistical analysis of software trustworthiness attributes”, *Sci. China Inf. Sci.*, vol 55, no 11, bll 2508–2520, Nov 2012, doi: 10.1007/s11432-012-4646-z.

[4] H. Tao, H. Wu, en Y. Chen, “An Approach of Trustworthy Measurement Allocation Based on Sub-Attributes of Software”, *Mathematics*, vol 7, no 3, bl 237, Mrt 2019, doi: 10.3390/math7030237.

[5] S. van Engelenburg, M. Janssen, en B. Klievink, "Design of a software architecture supporting business-to-government information sharing to improve public safety and security", *J. Intell. Inf. Syst.*, vol 52, no 3, bll 595–618, Jun 2019, doi: 10.1007/s10844-017-0478-z.

[6] B. Jereb, "Software describing attributes", *Comput. Stand. Interfaces*, vol 31, no 4, bll 653–660, Jun 2009, doi: 10.1016/j.csi.2008.06.012.

[7] K. O. Elish en M. Alshayeb, "A Classification of Refactoring Methods Based on Software Quality Attributes", *Arab. J. Sci. Eng.*, vol 36, no 7, bll 1253–1267, Nov 2011, doi: 10.1007/s13369-011-0117-x.

[8] L. Lavazza, S. Morasca, en D. Tosi, "Technical debt as an external software attribute", in *Proceedings of the 2018 International Conference on Technical Debt*, Mei 2018, bll 21–30. doi: 10.1145/3194164.3194168.

[9] A. Ampatzoglou, S. Charalampidou, en I. Stamelos, "Research state of the art on GoF design patterns: A mapping study", *J. Syst. Softw.*, 2013, doi: 10.1016/j.jss.2013.03.063.

[10] M. Hneif en Sai Peck Lee, "Using Guidelines to Improve Quality in Software Nonfunctional Attributes", *IEEE Softw.*, vol 28, no 6, bll 72–77, Nov 2011, doi: 10.1109/MS.2010.157.

[11] J. Moses, "Should we try to measure software quality attributes directly?", *Softw. Qual. J.*, vol 17, no 2, bll 203–213, Jun 2009, doi: 10.1007/s11219-008-9071-6.

[12] A. A. AYDIN, "Prominent quality attributes of crisis software systems: a literature review", *TURKISH J. Electr. Eng. Comput. Sci.*, vol 28, no 5, bll 2507–2522, Sep 2020, doi: 10.3906/elk-1911-5.

[13] A. Kaur, "A Systematic Literature Review on Empirical Analysis of the Relationship Between Code Smells and Software Quality Attributes", *Arch. Comput. Methods Eng.*, vol 27, no 4, bll 1267–1296, Sep 2020, doi: 10.1007/s11831-019-09348-6.

[14] A. Anand en G. Bansal, "Interpretive structural modelling for attributes of software quality", *J. Adv. Manag. Res.*, vol 14, no 3, bll 256–269, Aug 2017, doi: 10.1108/JAMR-11-2016-0097.

[15] M. Alshayeb, "The Impact of Refactoring to Patterns on Software Quality Attributes", *Arab. J. Sci. Eng.*, vol 36, no 7, bll 1241–1251, Nov 2011, doi: 10.1007/s13369-011-0111-3.

[16] Kahtan, "DEPENDABILITY ATTRIBUTES FOR INCREASED SECURITY IN COMPONENT-BASED SOFTWARE DEVELOPMENT", *J. Comput. Sci.*, vol 10, no 7, bll 1298–1306, Jul 2014, doi: 10.3844/jcssp.2014.1298.1306.

[17] K. O. Elish en M. Alshayeb, "Using Software Quality Attributes to Classify Refactoring to Patterns", *J. Softw.*, vol 7, no 2, Feb 2012, doi: 10.4304/jsw.7.2.408-419.

CHAPTER 9

SOFTWARE DESIGN METHODS

Dr. Lokesh Kumar, Assistant Professor
Department of Computer Science and Engineering, Sanskriti University, Mathura, Uttar Pradesh, India
Email Id- lokesh@sanskriti.edu.in

ABSTRACT:

Software design refers to the method by which an agent develops a specification of a software artefact that aims to accomplish objectives, utilizing a collection of simple building blocks and subject to boundaries. In this work, architectural design methodologies that describe the state of the art in software are provided. First, we provide a brief review of the subject of design methodologies and discuss why conventional object-oriented methodologies are insufficient for the design of software architecture. Then, we succinctly outline a variety of suggested approaches and strategies for software design process. We conclude the study by outlining the likely new research. It is essential for the growth of the software architecture research field and the profession of software engineering.

KEYWORDS:

Computer Science, Software Engineering, Software Design, SDLC, Waterfall Model.

INTRODUCTION

Software system design is the process of establishing a solution to one or more problems in a manner that balances the requirements imposed on the solution. A software architecture design technique implies the defining of two things. First, a process or methodology for completing the activities specified. An explanation of the results, or the sort of outcomes that will be attained when the strategy is applied, follows in second. These are the procedures for defining the architectural abstractions, the components and their interfaces, interactions between components, and design choices when alternative options are available from the viewpoint of software architecture. These choices are then put in writing so that an application may be designed and implemented accurately[1].

Software architecture refers to the most abstract level at which we actually create and plan software systems. The software design places limitations on the quality levels that the resulting systems can achieve. Thus, software architecture is the best solution to fulfil requirements for software quality, such as reusability, performance, safety, and reliability. When we consider what software architectural design is, we realize how important it is to balance needs, particularly those that are connected to software quality. The design process must include a step that determines if the design product, in this case the software architecture, meets the requirements or whether more iterations of the approach are required. If this operation is missing from a method, we cannot consider it to have finished [2].

The enabling technology for the so-called design phase is neither technological nor physical; rather, it is human creativity. It is up to the human mind to choose the proper abstractions, build relations, etc. to make sure the solution fits the purpose. Even if some aspects of these tasks may be aided by exact methodology, every design approach will depend on the creative

genius of the designer, or the competence of the unique human mind. Methodological differences will show up as more or less effective input and output processing or more or less suitable metaphors for defining input and output. This does not preclude design methods from outlining specific instances when they succeed or fall short. It is critical to remember that a technique can never fully make up for a lack of knowledge and that its results strongly depend on the skill of the participants. Software development is the name given to a group of computer science tasks involved in developing, deploying, and maintaining software[3].

The collection of instructions or programmer that a computer follows are known as software. It makes computers programmable and is independent of hardware. There are three fundamental kinds:

A. System software for fundamental operations such operating systems, disc management, utilities, hardware management, and other requirements for operation

B. Developing software that gives programmers access to coding tools including text editors, compilers, linkers, and debuggers.

C. Application software also known as apps that assists users in carrying out tasks. Examples include office productivity suites, data management applications, media players, and security software. Web and mobile apps, such as those used to purchase on Amazon.com, interact with Facebook, or upload photos to Instagram, are also referred to as applications.

The fourth category may be embedded software. Software for embedded systems is used to operate a variety of machinery and gadgets, including automobiles, industrial robots, telecommunications networks, and more. The Internet of Things allows for the connection of these devices and associated software (IoT).

Programmers, software engineers, and developers are largely responsible for software development. The relationships between these professions vary significantly between development departments and communities, and they interact and overlap.

A. Computer programmers, also known as coders or programmers, create source code to instruct computers to do certain activities, such as combining databases, processing online orders, directing communications, running searches, or displaying text and images. Programmers often translate instructions from engineers and software developers into actions using programming languages like C++ or Java.

B. Software engineers create software and systems to address issues using engineering concepts. Instead than only providing a solution for a single instance or customer, they often create solutions that can be applied to issues generally using modelling language and other tools. Software engineering solutions follow the scientific process and have to function in the actual world, like elevators or bridges. With the advent of microprocessors, sensors, and software, goods have become more intelligent, increasing their responsibility. Software development must be integrated with the product's mechanical and electrical development work since more products are depending on it to differentiate themselves in the market.

C. Software developers may be directly engaged with certain project areas, including creating code, and have a less formal function than engineers. They also manage development teams and procedures, carry out software testing and maintenance, collaborate across functional teams to translate requirements into features, and drive the whole software development lifecycle.

Software development involves more than just coding and development teams. Even though they aren't mainly software engineers, experts like scientists, gadget builders, and hardware producers nonetheless write software code. Additionally, it is not only limited to conventional information technology sectors like the semiconductor or software industry. In actuality, such corporations "account for fewer than half of the organisations conducting software development," according to the Brookings Institute. Custom software development as opposed to commercial software development is a crucial difference. The process of designing, building, deploying, and supporting software specifically for a group of users, tasks, or organisations is known as custom software development. Contrarily, commercial off-the-shelf software (COTS) may be packaged, commercially promoted, and distributed because it is made to meet a wide range of criteria. In Figure 1 shown the Solution Design in Systems Integration.

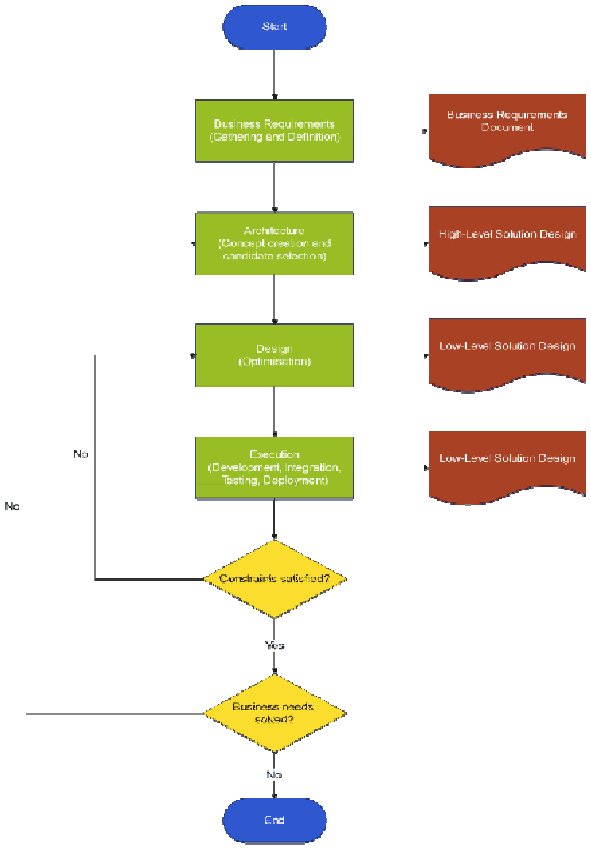

Figure 1: Illustrated that the Solution Design in Systems Integration

At first, these needs are broadly defined; for example, designing and implementing an algorithm to predict the demand for umbrellas in the rainy season. During requirement gathering, typically conducted by Business Analysts, requirements are articulated and refined, and the results are documented in a BRD. Requirement definition results in a clear set of objectives, some of which are functional, while others can be non-functional or project constraints. Non-functional requirements are fundamental to large software systems and will be discussed later in detail. The stage is now set, and architects can construct the high-level design. Potential solutions are explored at a conceptual level during this stage, and a solution candidate is selected.

After the high-level design is approved, tech leads and senior developers produce a low-level solution design in a series of iterations and revisions involving Business Analysts. Low-

level technical decisions about the product and its use cases are made, and parameters are tweaked to satisfy the product's usability constraints. Using the umbrella demand forecast application as an example, we can think of the high-level design decisions as follows:

i. The choice of the prediction engine (regression models, neural networks)

ii. The data processing pipeline (sanitization, generation of training sets)

iii. Model update and deployment

iv. Technology stack

v. Training algorithms

vi. Non-functional requirements such as redundancy and high availability.

Low-level technical design decisions might involve:

i. The User Interface (UI)

ii. The deployment models

iii. Backup and recovery

iv. Edge cases

v. Non-functional requirements such as compliance and performance.

It is crucial to involve the customers in both design stages, and this involvement is especially vital in large projects where feedback loops can be very long. It may be too late and expensive to correct design problems after they have been heavily integrated into the more comprehensive solution and their effects propagated and mitigated in downstream applications. This process involving one high and another low-level design phase with many iterations in between them is the generic approach for solution design in system integration projects.

LITERATURE REVIEW

M. Mekni et al. illustrated that this chapter, we propose a novel methodology to guide and assist practitioners supporting software architecture and design activities in agile environments. Software architecture and design is the skeleton of a system. It defines how the system has to behave in terms of different functional and non-functional requirements. Currently, a clear specification of software architectural design activities and processes in agile environments does not exist. Our methodology describes in detail the phases in the agile software design process and proposes techniques and tools to implement these phases [4].

C. Altin Gumussoy stated that the diversity of services in the financial market increases, it is critical to design usable banking software in order to overcome the complex structure of the system. The current study presents a usability guideline based on heuristics and their corresponding criteria that could be used during the early stages of banking software design process. In the design of a usability guideline, the heuristics and their criteria are categorized in terms of their effectiveness in solving usability problems grouped and ranging from usability catastrophe to cosmetic problems. The current study comprises of three main steps: First, actual usability problems from three banking software development projects are categorized according to their severity level. Secondly, usability criteria are rated for how well they explain the usability problems encountered. Finally, usability heuristics are categorized according to the severity level of usability problems through two analytical

models; corresponding and cluster analyses. As the result, designers and project managers may give more importance to the heuristics related with the following usability problem categories: Usability catastrophe and then major usability problems. Furthermore, the proposed guideline can be used to understand which usability criteria would be helpful in explaining usability problems as well as preventing banking system catastrophes, by highlighting the critical parts in system design of banking software [5].

D. Budgen illustrated that the much of the difficulty underlying the development of large software-based systems arises from the complex and abstract nature of software itself, and nowhere is this more evident than in the problems encountered in seeking to establish systematic procedures for designing software. This paper first examines the properties of software and the design practices that are involved in its development, considering in particular how software design methods seek to systemize these. We then introduce the use of what we have termed the D-matrix as a means of describing 'software design models', and employ this to explore the forms of the models that are developed by following the procedures of a number of well-established software design methods. We conclude by reviewing these models and considering the factors that limit the practices that can be used in such methods, as well as the extent to which the more recently developed design methods can minimize their effects [6].

P.Flores et al. illustrated that this study aims to discover what persistent ideas students have when designing software, and discusses possible relationships between them. The research was conducted through qualitative case study over an academic period with Master's degree students in a Software Design course. The ideas obtained as results were grouped in persistence levels: low, medium and high; additionally, some ideas have been identified, that could be potentially persistent. The main contribution of this paper is focused on two aspects: (a) Software design education, which allows teachers to identify and address problems related to Software Design course; and (b) Professional impact in the industry, by warning the software industry about the main problems that students carry out, despite of the instruction [7].

H. Washizaki et al. illustrated that the robot design contest, called the "Embedded Technology (ET) Software Design Robot Contest," which involves designing software to automatically control a line-trace robot, was held in Tokyo, in 2005. The contest was intended to provide a practical opportunity to educate young Japanese developers in the field of embedded software development. In this paper, we give the results of the contest from the viewpoint of software quality evaluation. We created a framework for evaluating software quality, which integrated the design model quality and the final system performance, and we conducted an analysis using this framework. As a result of the analysis, the quantitative measurement of the structural complexity of the design model was found to have a strong relationship to the qualitative evaluation of the design by the contest judges. On the other hand, no strong correlation between the design model quality evaluated by the judges and the final system performance was found. For embedded software development, it is particularly important to estimate and verify reliability and performance in the early stages, according to the design and analysis models. Based on the results, we consider possible remedies with respect to the models submitted, the evaluation methods used, and the contest specifications. To adequately measure several quality characteristics, including performance, in terms of a model, it is necessary to improve the approach to developing robot software and to reexamine the evaluation methods [8].

M. Yeh ststed that the change in software design strategies used by novice programmers over the course of one semester by using verbal protocol analysis. Our participants were nine first-year undergraduate students (novices), and two experts. Overall, we observed that two types of strategy were used by the novice programmers. The most common strategy observed in our participants, at the beginning of the semester, was a UI-based strategy that focused on system components from the user's perspective. This strategy is often overly simplified with little operational and technical details. Another type of strategy used by novices later in the study was a functional-centered strategy in which novices incorporated programming concepts into their design. Novices who used the latter strategy were able to provide more operational detail than when the UI-based strategy was used. We also found that, due to lack of experience, the designs were still very preliminary. In addition, the novices also exhibited opportunistic design behavior more often than systematic behavior (i.e., a top-down or bottom-up strategy) during the semester. We argue that teaching programming knowledge and skills alone will not develop students' software design knowledge effectively [9].

M. Aniche et al. illustrated that the extensive 50-year-old body of knowledge in object-oriented programming and design, good software designs are, among other characteristics, lowly coupled, highly cohesive, extensible, comprehensible, and not fragile. However, with the increased complexity and heterogeneity of contemporary software, this might not be enough. This paper discusses the practical challenges of object-oriented design in modern software development. We focus on three main challenges the first one is how technologies, frameworks, and architectures pressure developers to make design decisions that they would not take in an ideal scenario, the complexity of current real-world problems require developers to devise not only a single, but several models for the same problem that live and interact together, and how existing quality assessment techniques for object-oriented design should go beyond high-level metrics. Finally, we propose an agenda for future research that should be tackled by both scientists and practitioners soon. This paper is a call for arms for more reality-oriented research on the object-oriented software design field [10].

G. Sielis et al. stated that the work describes the design, development and evaluation of a software Prototype, an educational tool that employs two types of Context-aware Recommendations of Design Patterns, to support users who want to improve their design skills when it comes to training for High Level Software models. The tool’s underlying algorithms take advantage of Semantic Web technologies, and the usage of Content based analysis for the computation of non-personalized recommendations for Design Patterns. The recommendations’ objective is to support users in functions such as finding the most suitable Design Pattern to use according to the working context, learn the meaning, objectives and usages of each Design Pattern. The current work presents the Semantic Modeling of the Software Design process through the definition of the context that defines the Software Design process and in particular the representation of the Design Patterns as Ontology model, the implemented Context Aware Recommendation Algorithms and the evaluation results extracted from a user based testing for the prototype [11].

D. Jackson ststed that the Python for Software Design is a concise introduction to software design using the Python programming language. Intended for people with no programming experience, this book starts with the most basic concepts and gradually adds new material. Some of the ideas students find most challenging, like recursion and object-oriented programming, are divided into a sequence of smaller steps and introduced over the course of several chapters. The focus is on the programming process, with special emphasis on debugging. The book includes a wide range of exercises, from short examples to substantial projects, so that students have ample opportunity to practice each new concept. Exercise

solutions and code examples are available from thinkpython.com, along with Swampy, a suite of Python programs that is used in some of the exercises[12].

B. Adelson and E. Soloway discussed that the designer's expertise rests on the knowledge and skills which develop with experience in a domain. As a result, when a designer is designing an object in an unfamiliar domain he will not have the same knowledge and skills available to him as when he is designing an object in a familiar domain. In this paper we look at the software designer's underlying constellation of knowledge and skills, and at the way in which this constellation is dependent upon experience in a domain. What skills drop out, what skills, or interactions of skills come forward as experience with the domain changes? To answer the above question, we studied expert designers in experimentally created design contexts with which they were differentially familiar. In this paper we describe the knowledge and skills we found were central to each of the above contexts and discuss the functional utility of each. In addition to discussing the knowledge and skills we observed in expert designers, we will also compare novice and expert behavior [13].

P. Holtkamp et al. illustrated that the Global software development changes the requirements in terms of soft competency and increases the complexity of social interaction by including intercultural aspects. While soft competency is often seen as crucial for the success of global software development projects, the concrete competence requirements remain unknown. Internationalization competency represents one of the first attempts to structure and describe the soft competence requirements for global software developers. Based on the diversity of tasks, competence requirements will differ among the various phases of software development. By conducting a survey on the importance of internationalization competences for the different phases of global software development, we identified differences in terms of competence importance and requirements in the phases. Adaptability and Cultural Awareness were the main differences. Cultural Awareness distinguishes requirements engineering and software design from testing and implementation while "Adaptability" distinguishes implementation and software design from requirements engineering and testing [14].

R. Zhang et al. illustrated that the recently, the trending 5G technology encourages extensive applications of on-device machine learning, which collects user data for model training. This requires cost-effective techniques to preserve the privacy and the security of model training within the resource-constrained environment. Traditional learning methods rely on the trust among the system for privacy and security. However, with the increase of the learning scale, maintaining every edge device's trustworthiness could be expensive. To cost-effectively establish trust in a trustless environment, this paper proposes democratic learning, which makes the first step to explore hardware/software co-design for blockchain-secured decentralized on-device learning. By utilizing blockchain's decentralization and tamper-proofing, our design secures AI learning in a trustless environment. To tackle the extra overhead introduced by blockchain, we propose as a novel blockchain consensus mechanism, which first exploits cross-domain reuse AI learning and blockchain consensus in AI learning architecture. Evaluation results show our DemL can protect AI learning from privacy leakage and model pollution, and demonstrated that privacy and security come with trivial hardware overhead and power consumption (2%). We believe that our work will open the door of synergizing blockchain and on-device learning for security and privacy.

DISCUSSION

The software and hardware are established, the types were identified, and the links between classes are acknowledged during the object-oriented assessment phase of software development. Addressing the area of application and particular system specifications is the

goal of object oriented analysis. Specifying specifications and doing an early examination of the normative framework and viability of a platform are the deliverables of this step. Object programming, dynamic modelling, and operational modelling are the three analysis approaches that are coupled for object-oriented analysis.

Object Modelling

The static framework of the software system is designed by object modelling. It identifies the components as well as the classes through which they may be grouped and the interconnections between them. The essential characteristics and functions that determine each class are also discussed. The following stages could be employed to depict the object simulation process:

A. Identify things and classify them.

B. Define the connections between classes.

C. Draw a diagram of the user object model.

D. Define the characteristics of a user object.

E. Specify the actions that should be taken in relation to the classes.

Dynamic Modeling

Dynamic modelling supports the objective of investigating the system's behaviours in response to time and environmental forces after its static performance has been explored. Dynamic Modeling can be definite as “a way of describing how an individual object responds to events, either internal events triggered by other objects, or external events triggered by the outside world. The following stages could be utilized to depict the dynamic simulation process:

A. Define any object's states.

B. Identify events and determine how well actions apply.

C. Create a state transition exploded view dynamic model schematic.

D. Describe each state and used the properties of something like the item,

E. Verify the state-transition graphs that were developed.

Functional Modeling

Functional programming is the third and last element in an object-oriented analysis. The multifunctional model illustrates underlying processing of an object as well as the alterations that take place as data is transmitted throughout activities. It defines the goals of dynamic modelling operations as well as object modelling activities. The information flow diagram from a structural modeling analysis is identical to that same functional model. The operational modelling method may be summarized by the components below:

A. List every input and output.

B. Create data flow diagrams that illustrate the functional relationships

C. Describe each function's goal.

D. List the restrictions.

E. Identify optimization standards

Object-Oriented Design

Following the analytical stage, the proposed framework is further developed using object-oriented design to generate an object-oriented model. Object-Oriented Design creates a model for the computational domain by mapping fundamental technology-independent ideas out from analytical model onto implement classes, identifying requirements, and designing connections. The establishment of a system's extension of the traditional is the primary goal of object-oriented design. The following are the elements of object-oriented design:

A. Outlining the platform's context

B. Software architecture creation

C. Recognition of something like the system's objects

D. Building of model specifications

E. Interface definitions for objects

The first step of object-oriented design is project planning, while the second layer is detailed design.

Conceptual Design

At this level, all the modules required to develop the system are recognized. Moreover, each class is given a set of defined duties. Class diagrams are used to demonstrate how classes relate to one another, while interaction diagrams are used to show how events progress. Another name for it is high-level design.

Detailed Design

Features and actions are now allocated to each-class in accordance with their interface diagrams. State machine diagrams are created to explain the additional design features. Additionally called low-level design.

Design Principles

Following are the major design principles:

Principle of Decoupling

A system with a number of classes that are decidedly interconnected is challenging to preserve because changes to one class might need cascade adjustments to other classes. Tight coupling in an OO design may be removed by adding additional programs or using birth right.

Confirming Cohesion

A cohesive class carries out a group of closely connected tasks. A class that lacks coherence will carry out unrelated tasks, albeit this won't have an impact on the system as a whole. It makes it challenging to manage, develop, maintain, and modify the complete software architecture.

Open-closed Principle

This idea states that a system ought to be able to grow to accommodate new demands. A system extension should not result in changes to the system's code or current implementation. Additionally, the following rules must be adhered to while using the open-closed principle:

A. Separate interfaces and implementations must be kept up for each concrete class.

B. Keep the characteristics private while working in a multithreaded environment.

C. Use of class and global variables should be kept to a minimum.

CONCLUSION

The field of software architecture need a revamp. Its reputation is harmed by several outdated beliefs about the issues it must address and the best way to do so. The core of a continuous approach to software architecture is seeing it as a continuous activity focused on developing hypotheses about how the system will fulfil quality criteria and then using empiricism to demonstrate that the system achieves them. Taking control of software design away from groups of individuals who aren't developers and placing it in the hands of those who can make it real and usable, the developers, is another shift that has to be made. The robustness and sustainability we need from today's apps won't come about until then.

REFERENCES

[1] R. N. Thakur en U. S. Pandey, “The Role of Model-View Controller in Object Oriented Software Development”, *Nepal J. Multidiscip. Res.*, 2019, doi: 10.3126/njmr.v2i2.26279.

[2] T. D. La Toza, M. Chen, L. Jiang, M. Zhao, en A. Van Der Hoek, “Borrowing from the crowd: A study of recombination in software design competitions”, 2015. doi: 10.1109/ICSE.2015.72.

[3] A. Molnar, “SMARTRIQS: A Simple Method Allowing Real-Time Respondent Interaction in Qualtrics Surveys”, *J. Behav. Exp. Financ.*, 2019, doi: 10.1016/j.jbef.2019.03.005.

[4] M. Mekni, G. Buddhavarapu, S. Chinthapatla, en M. Gangula, “Software Architectural Design in Agile Environments”, *J. Comput. Commun.*, vol 06, no 01, bll 171–189, 2018, doi: 10.4236/jcc.2018.61018.

[5] C. Altin Gumussoy, “Usability guideline for banking software design”, *Comput. Human Behav.*, 2016, doi: 10.1016/j.chb.2016.04.001.

[6] D. Budgen, “‘Design models’ from software design methods”, *Des. Stud.*, 1995, doi: 10.1016/0142-694X(95)00001-8.

[7] P. Flores, N. Medinilla, en S. Pamplona, “Persistent Ideas in a Software Design Course: A Qualitative Case Study”, *Int. J. Eng. Educ.*, 2016.

[8] H. Washizaki *et al.*, “Quality evaluation of embedded software in robot software design contest”, *Prog. Informatics*, 2007, doi: 10.2201/NiiPi.2007.4.6.

[9] M. K. C. Yeh, “Examining novice programmers’ software design strategies through verbal protocol analysis”, *International Journal of Engineering Education*. 2018.

[10] M. Aniche, J. Yoder, en F. Kon, “Current Challenges in Practical Object-Oriented Software Design”, in *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, Mei 2019, bll 113–116. doi: 10.1109/ICSE-NIER.2019.00037.

[11] G. A. Sielis, A. Tzanavari, en G. A. Papadopoulos, "ArchReco: a software tool to assist software design based on context aware recommendations of design patterns", *J. Softw. Eng. Res. Dev.*, 2017, doi: 10.1186/s40411-017-0036-y.

[12] D. Jackson, "Alloy: A language and tool for exploring software designs", *Commun. ACM*, 2019, doi: 10.1145/3338843.

[13] B. Adelson en E. Soloway, "The Role of Domain Experience in Software Design", *IEEE Trans. Softw. Eng.*, 1985, doi: 10.1109/TSE.1985.231883.

[14] P. Holtkamp, J. P. P. Jokinen, en J. M. Pawlowski, "Soft competency requirements in requirements engineering, software design, implementation, and testing", *J. Syst. Softw.*, vol 101, bll 136–146, Mrt 2015, doi: 10.1016/j.jss.2014.12.010.

CHAPTER 10

SOFTWARE DESIGN AND THEIR OPTIMIZATION

Dr. Himanshu Singh, Assistant Professor
Department of Computer Science and Engineering, Sanskriti University, Mathura, Uttar Pradesh, India
Email Id- himanshu.singh@sanskriti.edu.in

ABSTRACT:

Engineering design approach known as design optimization uses a mathematical definition of a design issue to enable selecting the most appropriate design from several possibilities. Many engineering design issues may be mathematically described as single-objective optimization issues, where one objective function must be reduced maximized while being subject to a set of tolerance threshold from demands in areas like quality performance or physical sizes. In future this paper will be elaborated that the previous technologies and it helps to the implemented on the new software design process.

KEYWORDS:

Computer Science, Software Engineering, Software Design, SDLC, Waterfall Model.

INTRODUCTION

Executives at Toyota made it a requirement to consider a broad range of alternatives during the decision-making process. They even encouraged their engineers to look for feedback from specialists in different areas or departments for maximum diversification. In Figure 1 shows the Exploring Alternative Designs.

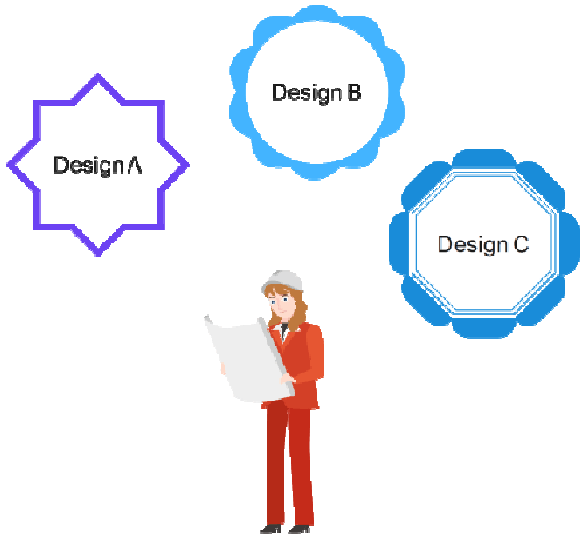

Figure 1: Illustrate that the Exploring Alternative Designs [1].

Toyota understood that although a full exploration of the design space can be costly and will inevitably delay product launches, this cost can be justified when the stakes are very high.Toyota executives stress the importance of efficient implementations to compensate for the time spent deliberating on design decisions. In software, alternatives often exist, and requirements for meeting deadlines may overshadow the viability of this exploration. This short-term tactical win is not efficient in the long term as it helps accumulate architectural and technical debt.

Solution Design Optimization

One of the outcomes of a solution architecture is a vector of design and operation parameters that can be combined and varied in different permutations. Each combination will produce a slightly different design that behaves similarly at its core level but with slight variations in more peripheral areas. One example of a design parameter is the number of TCP links between two components. One link is easy to manage, while multiple links make session and communication management slightly more complicated but increase throughput. An operational parameter can be the information displayed on the dashboard of an infrastructure monitoring application[2]. The amount of data can be increased for better sense-making of problems, albeit at the cost of additional data storage and processing.

The complexity of the optimization process is because:

A. Any change in one design parameter typically induces an adverse change in another. In most cases, better performance and more features can be attained by increasing complexity, cost, and project risk.

B. Another challenge lies in swiftly and effectively concluding a multi-actor, cross-departmental decision-making exercise if that happens to be the case.

A sweet spot must be located. You know you have hit the Pareto optimum when no change in any parameter can be made without negatively impacting another. There could exist not one but multiple Pareto optimums, and part of the challenge is to know which solution will work best shown in Figure 2.

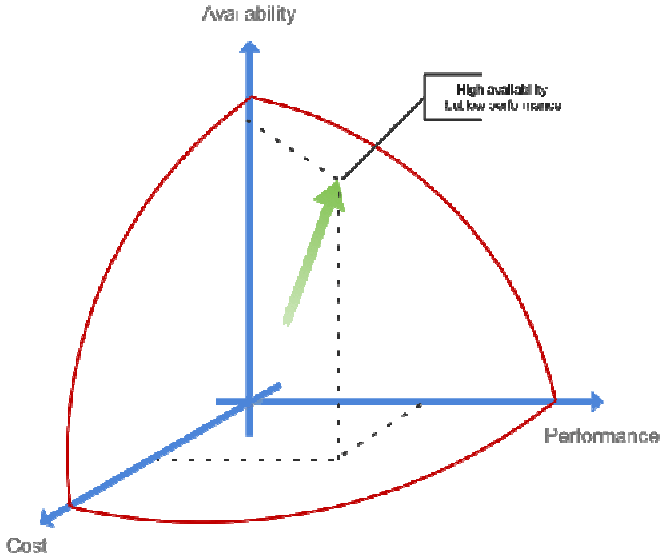

Figure 2: Illustrated that the Pareto Optimum Surface of System Cost, Availability, and Performance Criteria [3]**.**

Consider the above graph, where system availability and performance are plotted. The idea is as follows: increasing the throughput of a system also increases the risk of damaging it and putting it out of service. The 2D surface in the graph represents the set of Pareto optimums available. You can decide which criterion is more important, performance or availability an SLA placing system availability at 98.7%, for example, will determine the peak performance achievable. Naturally, if you want to increase performance and availability simultaneously, you must upgrade your infrastructure at a further cost.

One method of preventing a deadlock is ranking design criteria by order of importance to stakeholders. We discuss this in the next section in more detail. The optimization process, however, should produce a robust design vis-a-vis the criteria ranking. This robustness is vital as stakeholder influence and preferences change; you want to keep the same solution despite changes in opinions or choices.

Ranking Decision Criteria

In profit-seeking organizations, the top-priority measure is long-term profit. Technical superiority is only desirable if it serves the business through competitive advantage, advancing the value proposition, or lowering long-term costs. Beyond that point, it is typically frowned upon by whoever is in charge of financials[4].

These subtle and conflicting constraints make technical choices non-trivial and subject to intense negotiations between business, technology, and finance departments. Profit and utility aside, as long as the final product does the job, there is much room for manoeuvring when fine-tuning design parameters. Consider the following example of an online and batch system in an ideal scenario. Table 1 shown the Process of Decision Criteria between Two Systems.

Table 1: Illustrated that the Process of Decision Criteria between Two Systems.

Sr. No.	Attribute	Batch System	Online System
1.	Technology	Mature technology that is easy to learn is OK if it delivers the required functionality.	Best in breed it is more important to get optimal performance even at the cost of increased time-to-market
2.	Availability	Medium priority as batch systems is required to be online only when needed.	A top priority in mission-critical systems
3.	Profit	Non-negotiable	Non-negotiable
4.	Utility, functionality, business value	Non-negotiable	Non-negotiable
5.	Performance	Medium priority as long as the work is done within a reasonable time frame	High priority for the best user experience

Importance of Design Solution

The three reasons given by the union's Federal Secretary to explain the abandonment of the project will be, as well as others, the subject of this section's discussion. We will argue that a proper design process should help eliminate such risks.

i. Avoiding Costly Redesign

Most would agree that the sooner an error is detected, the less costly it is to fix it this means careful planning and design at the early stages of the project helps control uncertainty and project risk. In Figure 3 shown the Design Efforts try to mitigate this Issue Upfront.

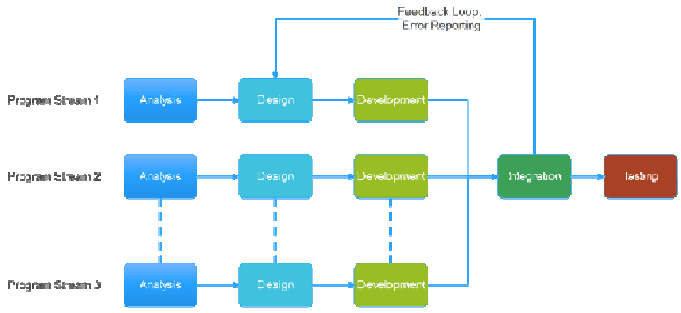

Figure 3: Represented that the Design Efforts try to mitigate this Issue Upfront [5]**.**

Rework is not an issue as long as it's cheap, but such cases are generally rare. The price of rework is measured by the *cost of change* and comes from the following:

A. *The Nature of the Project:*

In some cases, little can be done to avoid high expenditures when rework is required. The cost of change rises with scale as more people and effort are involved in redesigning a feature, and the longer the problem goes undetected, the harder it will be to fix it, especially if more features are built around it.

B. *The Project Delivery* Methodology*:*

Agile projects are more suited for earlier error detection because of shorter feedback loops. On the other hand, Agile is not always applicable, although a hybrid model can be adopted with no risk.When discussing rework or redesign, a distinction must be made between corrective measures and experimentation.In the latter case, you typically carry more than one design and try these out to test their feasibility and suitability. In the former, rework is created due to failure in the analysis or implementation stages, and this failure could have been avoided with more diligence. Diligence in designing solutions can reduce or even eliminate rework.

Solution Design Requirements

When designing a system, we look for an effective solution does the job that is also efficient does it well and cost-effective. The diagram below in Figure 4, articulates these ideas with a hierarchy where Operational Excellence in the traditional sense, slightly different from what we use in their website for Software Development is on the top.

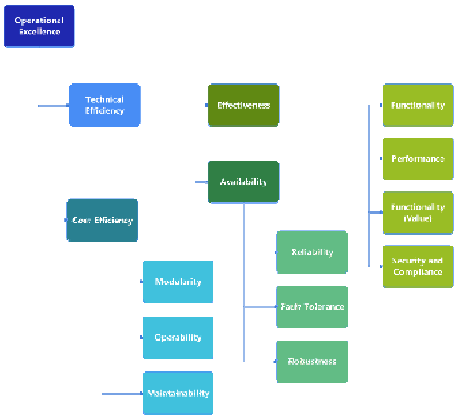

Figure 4: Represented that the Solution Design Functional and Non-Functional Requirements[6]**.**

A practical solution can only achieve some things, and once the optimal design has been found, changes from there on would mean a compromise of one aspect for another.

i. Functional Requirements

Functional requirements (also called business requirements) cover the following:

C. Services and features that have business value to the user. They typically solve a business need.

D. User experience, including user-friendliness, aesthetics, and running costs.

E. The system's behaviour in nominal and exceptional scenarios.

F. The system users are generally happy to pay for the services they receive. For example, a video game's functional requirement may cover gaming experience, which OS systems it can run on, and the minimum hardware required.

ii. Non-Functional Requirements

Non-functional requirements make the product viable and are equally interesting to the organisation and the end user. The end user does not directly pay for non-functional requirements; the supplier does through initial investments and running costs. Going back to our example of video game software, the typical gamer is not necessarily thrilled by the software's technology stack, the CI/CD pipeline on which it gets built, or its maintainability and modularity as long as they have a great gaming experience. Maintainability and modularity, however, are interesting for the vendor as they can keep the costs down and increase the value proposition.

Security

Software and cyber security requirements are essential to any IT project. Security can be broken down into five major categories:

A. **Confidentiality:** Protecting private data from unauthorized access

B. **Integrity:** Or how easy it is to identify messages that have been exchanged between two parties and were either modified or tampered with along the way

C. **Non-repudiation:** The degree to which actions or events can be proven to have occurred so that the events or activities cannot be repudiated later.

D. **Accountability:** This aspect deals with the firm's ability to audit sensitive activities on the system.

E. **Authenticity:** The degree to which internal or external entities accessing the system can be authenticated.

Security requirements must be observed in all designs involving sensitive information such as user and customer data.

LITERATURE REVIEW

D. Budgen illustrated that the much of the difficulty underlying the development of large software-based systems arises from the complex and abstract nature of software itself, and nowhere is this more evident than in the problems encountered in seeking to establish systematic procedures for designing software. This paper first examines the properties of software and the design practices that are involved in its development, considering in

particular how software design methods seek to systemize these. We then introduce the use of what we have termed the D-matrix as a means of describing 'software design models', and employ this to explore the forms of the models that are developed by following the procedures of a number of well-established software design methods. We conclude by reviewing these models and considering the factors that limit the practices that can be used in such methods, as well as the extent to which the more recently developed design methods can minimize their effects [7].

P. Holtkamp et al. illustrated that the Global software development changes the requirements in terms of soft competency and increases the complexity of social interaction by including intercultural aspects. While soft competency is often seen as crucial for the success of global software development projects, the concrete competence requirements remain unknown. Internationalization competency represents one of the first attempts to structure and describe the soft competence requirements for global software developers. Based on the diversity of tasks, competence requirements will differ among the various phases of software development. By conducting a survey on the importance of internationalization competences for the different phases of global software development, we identified differences in terms of competence importance and requirements in the phases. Adaptability and Cultural Awareness were the main differences. Cultural Awareness distinguishes requirements engineering and software design from testing and implementation while "Adaptability" distinguishes implementation and software design from requirements engineering and testing [8].

H. Washizaki et al. illustrated that the robot design contest, called the "Embedded Technology (ET) Software Design Robot Contest," which involves designing software to automatically control a line-trace robot, was held in Tokyo, in 2005. The contest was intended to provide a practical opportunity to educate young Japanese developers in the field of embedded software development. In this paper, we give the results of the contest from the viewpoint of software quality evaluation. We created a framework for evaluating software quality, which integrated the design model quality and the final system performance, and we conducted an analysis using this framework. As a result of the analysis, the quantitative measurement of the structural complexity of the design model was found to have a strong relationship to the qualitative evaluation of the design by the contest judges. On the other hand, no strong correlation between the design model quality evaluated by the judges and the final system performance was found. For embedded software development, it is particularly important to estimate and verify reliability and performance in the early stages, according to the design and analysis models. Based on the results, we consider possible remedies with respect to the models submitted, the evaluation methods used, and the contest specifications. To adequately measure several quality characteristics, including performance, in terms of a model, it is necessary to improve the approach to developing robot software and to reexamine the evaluation methods [9].

M. Mekni et al. illustrated that this chapter, we propose a novel methodology to guide and assist practitioners supporting software architecture and design activities in agile environments. Software architecture and design is the skeleton of a system. It defines how the system has to behave in terms of different functional and non-functional requirements. Currently, a clear specification of software architectural design activities and processes in agile environments does not exist. Our methodology describes in detail the phases in the agile software design process and proposes techniques and tools to implement these phases [10].

M. Yeh ststed that the change in software design strategies used by novice programmers over the course of one semester by using verbal protocol analysis. Our participants were nine first-year undergraduate students (novices), and two experts. Overall, we observed that two types

of strategy were used by the novice programmers. The most common strategy observed in our participants, at the beginning of the semester, was a UI-based strategy that focused on system components from the user's perspective. This strategy is often overly simplified with little operational and technical details. Another type of strategy used by novices later in the study was a functional-centered strategy in which novices incorporated programming concepts into their design. Novices who used the latter strategy were able to provide more operational detail than when the UI-based strategy was used. We also found that, due to lack of experience, the designs were still very preliminary. In addition, the novices also exhibited opportunistic design behavior more often than systematic behavior (i.e., a top-down or bottom-up strategy) during the semester. We argue that teaching programming knowledge and skills alone will not develop students' software design knowledge effectively [11].

C. Altin Gumussoy stated that the diversity of services in the financial market increases, it is critical to design usable banking software in order to overcome the complex structure of the system. The current study presents a usability guideline based on heuristics and their corresponding criteria that could be used during the early stages of banking software design process. In the design of a usability guideline, the heuristics and their criteria are categorized in terms of their effectiveness in solving usability problems grouped and ranging from usability catastrophe to cosmetic problems. The current study comprises of three main steps: First, actual usability problems from three banking software development projects are categorized according to their severity level. Secondly, usability criteria are rated for how well they explain the usability problems encountered. Finally, usability heuristics are categorized according to the severity level of usability problems through two analytical models; corresponding and cluster analyses. As the result, designers and project managers may give more importance to the heuristics related with the following usability problem categories: Usability catastrophe and then major usability problems. Furthermore, the proposed guideline can be used to understand which usability criteria would be helpful in explaining usability problems as well as preventing banking system catastrophes, by highlighting the critical parts in system design of banking software [12].

P.Flores et al. illustrated that this study aims to discover what persistent ideas students have when designing software, and discusses possible relationships between them. The research was conducted through qualitative case study over an academic period with Master's degree students in a Software Design course. The ideas obtained as results were grouped in persistence levels: low, medium and high; additionally some ideas have been identified, that could be potentially persistent. The main contribution of this paper is focused on two aspects: (a) Software design education, which allows teachers to identify and address problems related to Software Design course; and (b) Professional impact in the industry, by warning the software industry about the main problems that students carry out, despite of the instruction [13].

D. Jackson ststed that the Python for Software Design is a concise introduction to software design using the Python programming language. Intended for people with no programming experience, this book starts with the most basic concepts and gradually adds new material. Some of the ideas students find most challenging, like recursion and object-oriented programming, are divided into a sequence of smaller steps and introduced over the course of several chapters. The focus is on the programming process, with special emphasis on debugging. The book includes a wide range of exercises, from short examples to substantial projects, so that students have ample opportunity to practice each new concept. Exercise solutions and code examples are available from thinkpython.com, along with Swampy, a suite of Python programs that is used in some of the exercises[14].

R. Zhang et al. illustrated that the recently, the trending 5G technology encourages extensive applications of on-device machine learning, which collects user data for model training. This requires cost-effective techniques to preserve the privacy and the security of model training within the resource-constrained environment. Traditional learning methods rely on the trust among the system for privacy and security. However, with the increase of the learning scale, maintaining every edge device's trustworthiness could be expensive. To cost-effectively establish trust in a trustless environment, this paper proposes democratic learning, which makes the first step to explore hardware/software co-design for blockchain-secured decentralized on-device learning. By utilizing blockchain's decentralization and tamper-proofing, our design secures AI learning in a trustless environment. To tackle the extra overhead introduced by blockchain, we propose as a novel blockchain consensus mechanism, which first exploits cross-domain reuse AI learning and blockchain consensus in AI learning architecture. Evaluation results show our DemL can protect AI learning from privacy leakage and model pollution, and demonstrated that privacy and security come with trivial hardware overhead and power consumption (2%). We believe that our work will open the door of synergizing blockchain and on-device learning for security and privacy[15].

G. Sielis et al. stated that the work describes the design, development and evaluation of a software Prototype, an educational tool that employs two types of Context-aware Recommendations of Design Patterns, to support users who want to improve their design skills when it comes to training for High Level Software models. The tool's underlying algorithms take advantage of Semantic Web technologies, and the usage of Content based analysis for the computation of non-personalized recommendations for Design Patterns. The recommendations' objective is to support users in functions such as finding the most suitable Design Pattern to use according to the working context, learn the meaning, objectives and usages of each Design Pattern. The current work presents the Semantic Modeling of the Software Design process through the definition of the context that defines the Software Design process and in particular the representation of the Design Patterns as Ontology model, the implemented Context Aware Recommendation Algorithms and the evaluation results extracted from a user based testing for the prototype [16].

B. Adelson and E. Soloway discussed that the designer's expertise rests on the knowledge and skills which develop with experience in a domain. As a result, when a designer is designing an object in an unfamiliar domain he will not have the same knowledge and skills available to him as when he is designing an object in a familiar domain. In this paper we look at the software designer's underlying constellation of knowledge and skills, and at the way in which this constellation is dependent upon experience in a domain. What skills drop out, what skills, or interactions of skills come forward as experience with the domain changes? To answer the above question, we studied expert designers in experimentally created design contexts with which they were differentially familiar. In this paper we describe the knowledge and skills we found were central to each of the above contexts and discuss the functional utility of each. In addition to discussing the knowledge and skills we observed in expert designers, we will also compare novice and expert behavior [17].

M. Aniche et al. illustrated that the extensive 50-year-old body of knowledge in object-oriented programming and design, good software designs are, among other characteristics, lowly coupled, highly cohesive, extensible, comprehensible, and not fragile. However, with the increased complexity and heterogeneity of contemporary software, this might not be enough. This paper discusses the practical challenges of object-oriented design in modern software development. We focus on three main challenges the first one is how technologies, frameworks, and architectures pressure developers to make design decisions that they would

not take in an ideal scenario, the complexity of current real-world problems require developers to devise not only a single, but several models for the same problem that live and interact together, and how existing quality assessment techniques for object-oriented design should go beyond high-level metrics. Finally, we propose an agenda for future research that should be tackled by both scientists and practitioners soon. This paper is a call for arms for more reality-oriented research on the object-oriented software design field [18].

DISCUSSION

It is of course desirable to have simple relations between parameters of different aspects. As an example, again, an architecture support system would serve as a laboratory, where it should be made possible to experiment on architectures with respect to several aspects. Let's say that an architecture was supposed to be created to support a specific software product, with certain known requirements. The architecture support system should then be manipulated by feeding it with suitable quality attributes. As output the system should then produce a style to further build the architecture on. However, the result of the discussions of the previous subsection says that there generally are no simple relations between unit operations and quality attributes. Since components of styles and patterns also depend on the structures that results from the unit operations, it is also hard to tell anything generally about the relations between styles and quality attributes. At least as long as only pure structure is regarded that is there are no simple mappings between quality attributes and the structure of styles/patterns. Of course there are some specific effects that comes out of some styles. However, a result of the use of certain styles, generally have to be seen with respect to the semantic context in which they are used. This implies that the parts of this area also have to be discriminated. However, it is hard to see how could be done.

However, when regarding the role of styles and patterns in software architecture it's important to see that a description of an architecture, primarily based on styles and/or patterns, might not be equivalent to the architecture. Such descriptions are rather an aspect on the architecture. A style pattern description might not be sufficient and sometimes even misleading. E.g., the components of a system, represented by an architecture, cannot be equivalent to style components, or pattern classes/objects. This is because some system components might be acting in e.g., several behavioural patterns, i.e. they have several roles distributed over a system. This means that a one-to-one relationship between a system component and a style component, generally cannot hold. A programming system, to elaborate architectures and producing a system out of it, needs to consider the number of variations of heterogeneity. Such a system might also need a two-dimensional view, consisting of, on one hand the system components, and on the other hand styles and patterns. The two dimensions are associated to each other by connections between the system components and instances of the different types of styles and or patterns.

CONCLUSION

In Systems Engineering, you typically create an objective function representing the criteria your stakeholders care about. The objective function is parameterized by the system design and operation parameters, and you would usually use Multi-Disciplinary Optimization (MDO) techniques to try and optimize that function. Software engineering does not have the equivalent of an objective function. Instead, you have a set of requirements, some of which are mandatory while others are nice-to-have. This differentiation arises between the two disciplines and gives rise to two phenomena. The first phenomenon is that software and solution design are more of an art than hard science. Still, like art, they are governed by

heuristics laboriously gathered over the years and the second phenomenon is that every new project that adds features to the solution will have to leverage the existing design; it would not be cost-efficient to redesign a solution from the ground up with every new addition. This approach to solution building produces multi-layered architectural hierarchies that are the hallmark of complexity, and the latter always comes at a cost. For these two reasons, solution and software design offer additional challenges that architects and engineers must tackle, and it's never a good idea to sacrifice design efforts in an unsustainable manner.

REFERENCES

[1] I. Harel en S. Papert, "Software Design as a Learning Environment", *Interact. Learn. Environ.*, vol 1, no 1, bll 1–32, Mrt 1990, doi: 10.1080/1049482900010102.

[2] A. Wiberg, J. Persson, en J. Ölvander, "Design for additive manufacturing – a review of available design methods and software", *Rapid Prototyping Journal*. 2019. doi: 10.1108/RPJ-10-2018-0262.

[3] H. Rezayat, J. R. Bell, A. J. Plotkowski, en S. S. Babu, "Multi-solution nature of topology optimization and its application in design for additive manufacturing", *Rapid Prototyp. J.*, 2019, doi: 10.1108/RPJ-01-2018-0009.

[4] A. Ramírez, J. R. Romero, en S. Ventura, "A survey of many-objective optimisation in search-based software engineering", *Journal of Systems and Software*. 2019. doi: 10.1016/j.jss.2018.12.015.

[5] M. Zhou, J. Alexandersen, O. Sigmund, en C. B. Claus, "Industrial application of topology optimization for combined conductive and convective heat transfer problems", *Struct. Multidiscip. Optim.*, 2016, doi: 10.1007/s00158-016-1433-2.

[6] S. Liu en X. Ning, "A two-stage building information modeling based building design method to improve lighting environment and increase energy efficiency", *Appl. Sci.*, 2019, doi: 10.3390/app9194076.

[7] D. Budgen, "'Design models' from software design methods", *Des. Stud.*, 1995, doi: 10.1016/0142-694X(95)00001-8.

[8] P. Holtkamp, J. P. P. Jokinen, en J. M. Pawlowski, "Soft competency requirements in requirements engineering, software design, implementation, and testing", *J. Syst. Softw.*, vol 101, bll 136–146, Mrt 2015, doi: 10.1016/j.jss.2014.12.010.

[9] H. Washizaki *et al.*, "Quality evaluation of embedded software in robot software design contest", *Prog. Informatics*, 2007, doi: 10.2201/NiiPi.2007.4.6.

[10] M. Mekni, G. Buddhavarapu, S. Chinthapatla, en M. Gangula, "Software Architectural Design in Agile Environments", *J. Comput. Commun.*, vol 06, no 01, bll 171–189, 2018, doi: 10.4236/jcc.2018.61018.

[11] M. K. C. Yeh, "Examining novice programmers' software design strategies through verbal protocol analysis", *International Journal of Engineering Education*. 2018.

[12] C. Altin Gumussoy, "Usability guideline for banking software design", *Comput. Human Behav.*, 2016, doi: 10.1016/j.chb.2016.04.001.

[13] P. Flores, N. Medinilla, en S. Pamplona, "Persistent Ideas in a Software Design Course: A Qualitative Case Study", *Int. J. Eng. Educ.*, 2016.

[14] D. Jackson, "Alloy: A language and tool for exploring software designs", *Commun. ACM*, 2019, doi: 10.1145/3338843.

[15] S. Chopra en D. Kumar, "Characterization, optimization and kinetics study of acetaminophen degradation by Bacillus drentensis strain S1 and waste water degradation analysis", *Bioresour. Bioprocess.*, 2020, doi: 10.1186/s40643-020-0297-x.

[16] G. A. Sielis, A. Tzanavari, en G. A. Papadopoulos, "ArchReco: a software tool to assist software design based on context aware recommendations of design patterns", *J. Softw. Eng. Res. Dev.*, 2017, doi: 10.1186/s40411-017-0036-y.

[17] B. Adelson en E. Soloway, “The Role of Domain Experience in Software Design”, *IEEE Trans. Softw. Eng.*, 1985, doi: 10.1109/TSE.1985.231883.

[18] M. Aniche, J. Yoder, en F. Kon, “Current Challenges in Practical Object-Oriented Software Design”, in *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, Mei 2019, bll 113–116. doi: 10.1109/ICSE-NIER.2019.00037.

CHAPTER 11

DESIGN ARCHITECTURE AND OPTIMIZATION

Dr. Deepak Chauhan, Assistant Professor
Department of Computer Science and Engineering, Sanskriti University, Mathura, Uttar Pradesh, India
Email Id- deepak.chauhan@sanskriti.edu.in

ABSTRACT:

Quality criteria are just as fundamental for embedded systems as technical specifications, if not more so. Architecture design must lay the foundations for the accomplishment of these quality objectives. For software and system architects, however, it can be problematic to discover an acceptable architecture design. The ever-increasing complication of today's systems, stringent design requirements, and contradictory quality criteria, to name a few, are the causes. In order to make the job easier, this chapter introduces Arche-Opterix, an expandable Eclipse-based tool that offers a framework for generating evaluation methods and optimization algorithms for AADL specifications.

Evolutionary methodologies are currently being used to find ideal and substantially ideal deployment architectures for a diverse range of quality goals and design limitations. The program can successfully locate solution architectures with better caliber, according to tests with a subset of first deployment architectures.

KEYWORDS:

Computer Science, Software Engineering, Software Design, SDLC, Waterfall Model.

INTRODUCTION

The architectural design is necessary for the programme to represent the software design. "The process of identifying a collection of hardware and software components and their interfaces to provide the foundation for the creation of a computer system," according to IEEE, is what architectural design is. One of these numerous architectural styles can be seen in the software created for computer-based systems [1]. Each style will outline a group of systems that includes:

- **i.** A collection of parts, such as a database or computing modules, that together carry out a specific task for the system.
- **ii.** The connectors will promote cooperation, coordination, and communication among the parts.
- **iii.** Requirements for how a system's components can be combined.
- **iv.** Semantic models that aid the designer in comprehending the system's general characteristics.

The system's components are given a structure through the usage of architectural styles.

Taxonomy of Architectural styles:

Data Cantered Architectures

i. The central component of this design will be a data store, which is constantly used by the other components in order to update, add, delete, or modify the data already present in the store.
ii. The graphic shows a common cantered data format. A central repository is accessed by the client software. When client-related data or client-interested data change the alerts to client software, a variation of this strategy is utilised to turn the repository into a blackboard.
iii. The inerrability of this data-cantered architecture will be enhanced. This means that new client components can be introduced to the architecture and old ones can be altered without the consent or worry of other customers.
iv. The Blackboard technique can be used to transfer data between clients.

Advantage of Data Cantered Architecture (DCA)

According to the Figure 1, it display the all the important parts of the DCA, those are perform the major role in the architecture.

i. Repository of data is independent of clients
ii. Client work independent of each other
iii. It may be simple to add additional clients.
iv. Modification can be very easy

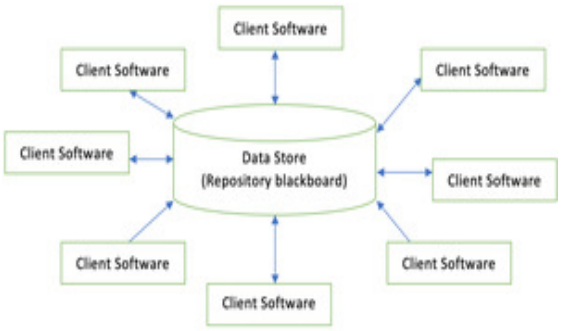

Figure 1: Represented that the Data Cantered Architecture [2]**.**

Data flow architectures:

i. This kind of architecture is used when input data to be transformed into output data through a series of computational manipulative components.
ii. The figure represents pipe-and-filter architecture since it uses both pipe and filter and it has a set of components called filters connected by pipes.
iii. Pipes are used to transmit data from one component to the next.
iv. Each filter will work independently and is designed to take data input of a certain form and produces data output to the next filter of a specified form. The filters don't require any knowledge of the working of neighbouring filters.

v. If the data flow degenerates into a single line of transforms, then it is termed as batch sequential. This structure accepts the batch of data and then applies a series of sequential components to transform it [3].

Advantage of Data Flow architecture

i. It encourages upkeep, repurposing, and modification.

ii. With this design, concurrent execution is supported.

Disadvantage of Data Flow Architecture

According to the Figure 2, it display the all the important parts of the DFA, those are perform the major role in this architecture.

i. It frequently degenerates to batch sequential system.

ii. Data flow architecture does not allow applications that require greater user engagement.

iii. It is not easy to coordinate two different but related streams.

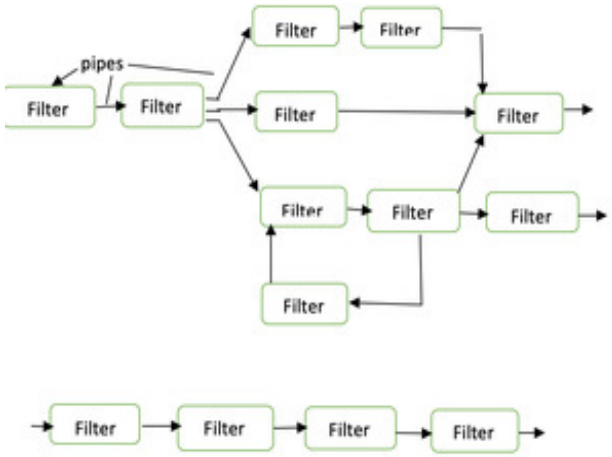

Figure 2: Illustrated that the Data Flow Architecture Model[4]**.**

Call and Return architectures

It is used to create a program that is easy to scale and modify. Many sub-styles exist within this category. Two of them are explained below.

i. Remote procedure call architecture

This components is used to present in a main program or sub program architecture distributed among multiple computers on a network.

ii. Main program or Subprogram architectures

The main program structure decomposes into number of subprograms or function into a control hierarchy. Main program contains number of subprograms that can invoke other components as mention in the Figure 3.

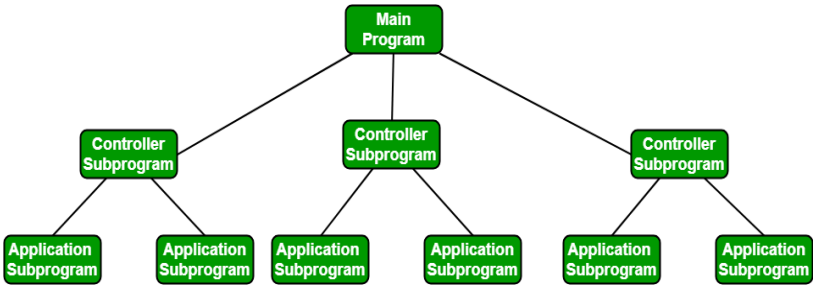

Figure 3: Represented that the Sub Programmes Architecture[5]**.**

Object Oriented Architecture

The components of a system encapsulate data and the operations that must be applied to manipulate the data. The coordination and communication between the components are established via the message passing.

Characteristics of Object Oriented architecture

i. Object protect the system's integrity.

ii. An object is unaware of the depiction of other items.

Advantage of Object Oriented architecture

i. It enables the designer to separate a challenge into a collection of autonomous objects.

ii. Other objects are aware of the implementation details of the object, allowing changes to be made without having an impact on other objects.

Layered architecture

i. A number of different layers are defined with each layer performing a well-defined set of operations. Each layer will do some operations that becomes closer to machine instruction set progressively as display in the Figure 4.

ii. At the outer layer, components will receive the user interface operations and at the inner layers, components will perform the operating system interfacing (communication and coordination with OS).

iii. Intermediate layers to utility services and application software functions.

iv. One common example of this architectural style is OSI-ISO (Open Systems Interconnection-International Organisation for Standardisation) communication system.

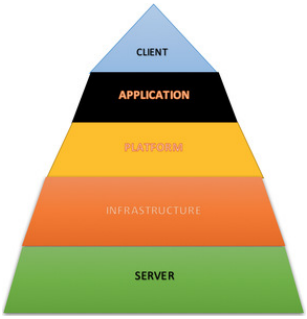

Figure 4: Represented that the Layered architecture.

The Architecture Design Process

We now dive into the process of architecture design: what it is, why it is important, how it works (at an abstract level) and which major concepts and activities it involves. We first discuss architectural drivers: the various factors that "drive" design decisions, some of which are documented as requirements, but many of which are not. In addition, we provide an overview of design concepts the major building blocks that you will select, combine, instantiate, analyse, and document as part of your design process.

Design in General

Design is both a verb and a noun. Design is a process, an activity, and hence a verb. The process results in the creation of a design a description of a desired end state. Thus the output of the design process is the thing, the noun, the artefact that you will eventually implement. Designing means making decisions to achieve goals and satisfy requirements and constraints. The outputs of the design process are a direct reflection of those goals, requirements, and constraints. Think about houses, for example.

The architectures of these styles of houses have evolved over the centuries to reflect their unique sets of goals, requirements, and constraints. Houses in China feature symmetric enclosures, sky wells to increase ventilation, south-facing courtyards to collect sunlight and provide protection from cold north winds, and so forth. A-frame houses have steep pitched roofs that extend to the ground, meaning minimal painting and protection from heavy snow loads which just slide off to the ground. Igloos are built of ice, reflecting the availability of ice, the relative poverty of other building materials, and the constraints of time.

In each case, the process of design involved the selection and adaptation of a number of solution approaches. Even igloo designs can vary. Some are small and meant for a temporary travel shelter. Others are large, often connecting several structures, meant for entire communities to meet. Some are simple unadorned snow huts. Others are lined with furs, with ice "windows", and doors made of animal skin. The process of design, in each case, balances the various "forces" facing the designer. Some designs require considerable skill to execute (such as carving designer. Some designs require considerable skill to execute such as carving and stacking snow blocks in such a way that they produce a self-supporting dome. Others require relatively little skill a lean-to can be constructed from branches and bark by almost anyone. But the qualities that these structures exhibit may also vary considerably.

Lean-tos provide little protection from the elements and are easily destroyed, whereas an igloo can withstand Arctic storms and support the weight of a person standing on the roof. Is design "hard"? Well, yes and no. Novel design is hard. It is pretty clear how to design a conventional bicycle, but the design for the Segway broke new ground. Fortunately, most design is not novel, because most of the time our requirements are not novel. Most people want a bicycle that will reliably convey them from place to place. The same holds true in every domain. Consider houses, for example. Most people living in Phoenix want a house that can be easily and economically kept cool, whereas most people in Edmonton are primarily concerned with a house that can be kept warm. In contrast, people living in Japan and Los Angeles are concerned with buildings that can withstand earthquakes. The good news for you, the architect, is that there are ample proven designs and design fragments, or building blocks that we call design concepts that can be reused and combined to reliably achieve these goals.

Design in Software Architecture

Architectural design for software systems is no different than design in general: It involves making decisions, working with available skills and materials, to satisfy requirements and constraints. In architectural design, we make decisions to transform our design purpose, requirements, constraints, and architectural concerns what we call the architectural drivers into structures, as shown in Figure 5. These structures are then used to guide the project. They guide analysis and construction, and serve as the foundation for educating a new project member. They also guide cost and schedule estimation, team formation, risk analysis and mitigation, and, of course, implementation.

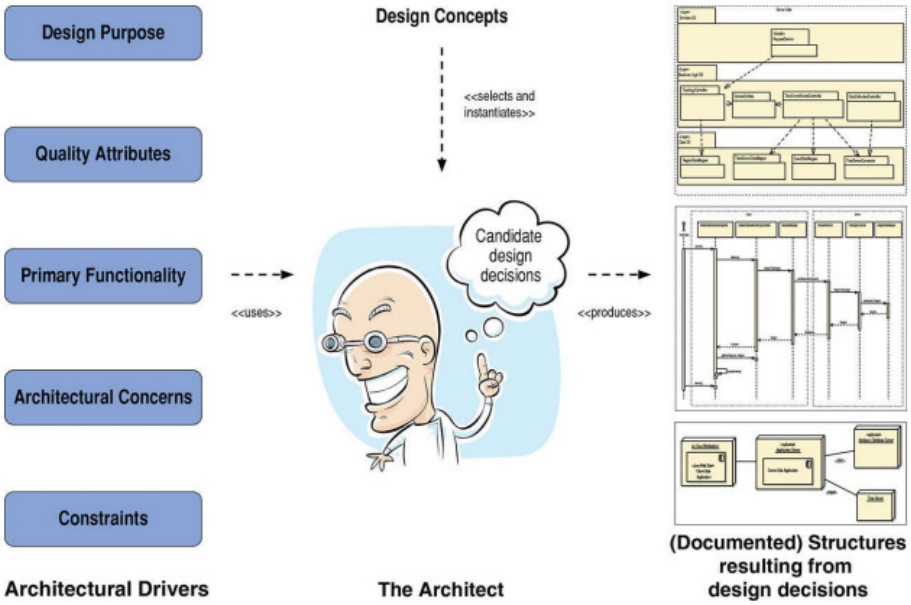

Figure 5: Represented that the Architecture Design Activity.

Architectural design is, therefore, a key step to achieving your product and project goals. Some of these goals are technical such that achieving low and predictable latency in a video game or an e-commerce website, and some are nontechnical like keeping the workforce employed, entering a new market, meeting a deadline. The decisions that you, as an architect, make will have implications for the achievement of these goals and may, in some cases, be in conflict. The choice of a particular reference architecture like the Rich Client Application may provide a good foundation for achieving your latency goals and will keep your workforce employed because they are already familiar with that reference architecture and its supporting technology stack. But this choice may not help you enter a new market mobile games, for example.

LITERATURE REVIEW

G. Guizzo et al. illustrated that the design of the product line architecture (PLA) is a difficult activity that can benefit from the application of design patterns and from the use of a search-based optimization approach, which is generally guided by different objectives related, for instance, to cohesion, coupling and PLA extensibility. The use of design patterns for PLAs is a recent research field, not completely explored yet. Some works apply the patterns manually and for a specific domain. Approaches to search-based PLA design do not consider the usage of these patterns. To allow such use, this paper introduces a mutation operator named "Pattern-Driven Mutation Operator" that includes methods to automatically identify suitable scopes and apply the patterns Strategy, Bridge and Mediator with the search-based approach

multi-objective optimization approach for PLA. A met model is proposed to represent and identify suitable scopes to receive each one of the patterns, avoiding the introduction of architectural anomalies. Empirical results are also presented, showing evidences that the use of the proposed operator produces a greater diversity of solutions and improves the quality of the PLAs obtained in the search-based optimization process, regarding the values of software metrics [6].

B. Shahriari et al. illustrated that the Big Data applications are typically associated with systems involving large numbers of users, massive complex software systems, and large-scale heterogeneous computing and storage architectures. The construction of such systems involves many distributed design choices. The end products such that the recommendation systems, medical analysis tools, real-time game engines, speech recognizers thus involve many tunable configuration parameters. These parameters are often specified and hard-coded into the software by various developers or teams. If optimized jointly, these parameters can result in significant improvements. Bayesian optimization is a powerful tool for the joint optimization of design choices that is gaining great popularity in recent years. It promises greater automation so as to increase both product quality and human productivity. This review paper introduces Bayesian optimization, highlights some of its methodological aspects, and showcases a wide range of applications [7].

J. Gray et al. ststed that the Multidisciplinary design optimization (MDO) is concerned with solving design problems involving coupled numerical models of complex engineering systems. While various MDO software frameworks exist, none of them take full advantage of state-of-the-art algorithms to solve coupled models efficiently. Furthermore, there is a need to facilitate the computation of the derivatives of these coupled models for use with gradient-based optimization algorithms to enable design with respect to large numbers of variables. In this paper, we present the theory and architecture of OpenMDAO, an open-source MDO framework that uses Newton-type algorithms to solve coupled systems and exploits problem structure through new hierarchical strategies to achieve high computational efficiency. OpenMDAO also provides a framework for computing coupled derivatives efficiently and in a way that exploits problem sparsely. We demonstrate the framework's efficiency by benchmarking scalable test problems. We also summarize a number of OpenMDAO applications previously reported in the literature, which include trajectory optimization, wing design, and structural topology optimization, demonstrating that the framework is effective in both coupling existing models and developing new multidisciplinary models from the ground up. Given the potential of the OpenMDAO framework, we expect the number of users and developers to continue growing, enabling even more diverse applications in engineering analysis and design [8].

T. Akiba et al. stated that the purpose of this study is to introduce new design-criteria for next-generation hyper parameter optimization software. The criteria we propose include the first one is define-by-run API that allows users to construct the parameter search space dynamically, second is efficient implementation of both searching and pruning strategies, and third one is easy-to-setup, versatile architecture that can be deployed for various purposes, ranging from scalable distributed computing to lightweight experiment conducted via interactive interface. In order to prove our point, we will introduce Optuna, an optimization software which is a culmination of our effort in the development of a next generation optimization software. As an optimization software designed with define-by-run principle, Optuna is particularly the first of its kind. We will present the design-techniques that became necessary in the development of the software that meets the above criteria, and demonstrate the power of our new design through experimental results and real world applications [9].

R. Clune et al. illustrated that the paper presents an object-oriented architecture for structural design software. The architecture's novel features are the representation of an artifact with distinct levels of idealization, a hierarchy of classification within each of these levels, and the appropriate separation of software components. These enable seamless integration of geometric modeling and structural analysis in an interactive environment, extensibility of modeling and analysis capabilities, and integration of interactive multi-objective optimization. The paper presents a design environment implemented on the basis of the architecture, and demonstrates the benefits of refocusing engineering software from analysis to design [10].

A. Krallish et al. illustrated that the polymer industry becomes more global and competitive pressures are intensifying, polymer manufacturers recognize the need for the development of advanced process simulators for polymer plants. The overall goal is to utilize powerful, flexible, adaptive design and predictive simulation tools that can follow and predict the behavior of polymer production processes in an accurate, prompt and comprehensive way. In response to the current needs, a new generation of software packages has been developed for the simulation, design, parameter and state estimation, optimization and control of specific polymerization processes aiming at increasing plant efficiency, improving product quality and reducing the impact to environment. The new software tools provide a user-friendly interface, including an object-oriented design environment that can be accessed from the engineer's windows-based desktop environment and provide full graphical interaction and expert system guidance on how to use the program or making engineering decisions such as selection of unit operation or physical property method [11].

A. Aleti ei al. ststed that due to significant industrial demands toward software systems with increasing complexity and challenging quality requirements, software architecture design has become an important development activity and the research domain is rapidly evolving. In the last decades, software architecture optimization methods, which aim to automate the search for an optimal architecture design with respect to a (set of) quality attribute(s), have proliferated. However, the reported results are fragmented over different research communities, multiple system domains, and multiple quality attributes. To integrate the existing research results, we have performed a systematic literature review and analyzed the results of 188 research papers from the different research communities. Based on this survey, a taxonomy has been created which is used to classify the existing research. Furthermore, the systematic analysis of the research literature provided in this review aims to help the research community in consolidating the existing research efforts and deriving a research agenda for future developments [12].

M. Karakush et al. illustrated that the Software-Defined Networking (SDN) architecture has emerged in response to limitations of traditional networking architectures in satisfying today's complex networking needs. In particular, SDN allows network administrators to manage network services through abstraction of lower-level functionality. However, SDN is a logically centralized technology. Therefore, scalability, and especially the control plane (i.e. controller) scalability in SDN is one of the problems that needs more attention. In this survey paper, we first discuss the scalability problems of controller(s) in an SDN architecture. We then comprehensively survey and summarize the characterizations and taxonomy of state-of-the-art studies in SDN control plane scalability. We organize the discussion on control plane scalability into two broad approaches: Topology-related approaches and Mechanisms-related approaches. In Topology-related approaches, we study the relation between topology of architectures and scalability issues. It has sub-categories of Centralized (Single) Controller Designs and Distributed approaches. Distributed approaches, in turn, have also sub-

categories: Distributed (Flat) Controller Designs, Hierarchical Controller Designs, and Hybrid Designs. In Mechanisms-related approaches, we review the relation between various mechanisms used to optimize controllers and scalability issues. It has sub-categories of Parallelism-based Optimization and Control Plane Routing Scheme-based Optimization. Furthermore, we outline the potential challenges and open problems that need to be addressed further for more scalable SDN control planes [13].

R. Li et al. stated that the design of software architecture is one of the difficult tasks in the modern component-based software development which is based on the idea that develop software systems by assembling appropriate off-the-shelf components with a well-defined software architecture. Component-based software development has achieved great success and been extensively applied to a large range of application domains from real-time embedded systems to online web-based applications. In contrast to traditional approaches, it requires software architects to address a large number of non-functional requirements that can be used to quantify the operation of system. Moreover, these quality attributes can be in conflict with each other. In practice, software designers try to come up with a set of different architectural designs and then identify good architectures among them. With the increasing scale of architecture, this process becomes time-consuming and error-prone. Consequently architects could easily end up with some suboptimal designs because of large and combinatorial search space. In this paper, we introduce AQOSA (Automated Quality-driven Optimization of Software Architecture) toolkit, which integrates modeling technologies, performance analysis techniques, and advanced evolutionary multi objective optimization algorithms to improve non-functional properties of systems in an automated manner [14].

N. Rankovic et al. illustrated that in this chapter, two different architectures of Artificial Neural Networks (ANN) are proposed as an efficient tool for predicting and estimating software effort. Artificial Neural Networks, as a branch of machine learning, are used in estimation because they tend towards fast learning and giving better and more accurate results. The search/optimization embraced here is motivated by the Taguchi method based on Orthogonal Arrays (an extraordinary set of Latin Squares), which demonstrated to be an effective apparatus in a robust design. This article aims to minimize the magnitude relative error (MRE) in effort estimation by using Taguchi's Orthogonal Arrays, as well as to find the simplest possible architecture of an artificial Neural Network for optimized learning. A descending gradient (GA) criterion has also been introduced to know when to stop performing iterations. Given the importance of estimating software projects, our work aims to cover as many different values of actual efficiency of a wide range of projects as possible by division into clusters and a certain coding method, in addition to the mentioned tools. In this way, the risk of error estimation can be reduced, to increase the rate of completed software projects[15].

M. Chmielewski et al. illustrated that the development of highly specialized mobile applications and systems has risen for several years, as we observe rapid deployment of innovative biomedical sensors and wearable technologies. The experiences gathered over 10 years of mobile medical software development, provide practical recommendations for architectural concepts utilized in analytical health-based services. Constructed systems and mobile applications in majority of cases utilize biomedical signals to identify health state of a patient, as well as to evaluate or estimate the intensity of disease symptoms. Based on these experiences, this paper proposes architectural concepts, for both mobile and web-based components aimed at acquisition and processing of biomedical data in large scale medical systems dedicated for monitoring of patients. This work provides construction details of wearable-based mobile systems with specialized aimed at health state identification and

monitoring. Undertaken construction decisions have been confirmed and justified based on functional and stress tests of system components. The first deployment attempt of designed architecture and its implementation has been aimed at remote, mobile monitoring of elderly people, and preconfigured for crucial health event recognition - fainting, stroke, cardiac arrest, seizures, and some classes of neurological disorders and derivatives of such conditions [16].

A. Koziolek et al. ststed thatb the design decisions for complex, component-based systems impact multiple quality of service (QoS) properties. Often, means to improve one quality property deteriorate another one. In this scenario, selecting a good solution with respect to a single quality attribute can lead to unacceptable results with respect to the other quality attributes. A promising way to deal with this problem is to exploit multi-objective optimization where the objectives represent different quality attributes. The aim of these techniques is to devise a set of solutions, each of which assures an optimal trade-off between the conflicting qualities. Our previous work proposed a combined use of analytical optimization techniques and evolutionary algorithms to efficiently identify an optimal set of design alternatives with respect to performance and costs. This paper extends this approach to more QoS properties by providing analytical algorithms for availability-cost optimization and three-dimensional availability-performance-cost optimization. We demonstrate the use of this approach on a case study, showing that the analytical step provides a better-than-random starting population for the evolutionary optimization, which lead to a speed-up of 28% in the availability-cost case [1].

DISCUSSION

In this study, a unique tool for optimising embedded system architectures, named Arche-Opterix, is introduced. This tool offers plug-in techniques to replace the optimization engine, the quality evaluation algorithms, and the constraints checking. It leverages the AADL as the underlying architecture description language. A specific multi-objective, multi-constraint component deployment problem has been utilised to validate the tool. Similar to other problems, this one has been addressed by using three constraints component position, component collocation, and memory consumption along with two common quality indicators (data transmission reliability and communication overhead). The early implementation of an evolutionary algorithm produced positive results, but there is still potential for advancement and a number of intriguing research problems that need to be resolved. The design team and other researchers should add more quality evaluation processes to ArcheOpterix in the future techniques that assess quality characteristics in quality domains that are important for embedded systems, such as performance, reliability, security, timeliness, and resource consumption. The programme will also be expanded with more optimization heuristics since ArcheOpterix should also function as an experiment platform for optimization techniques. The effectiveness of these heuristics for benchmark problems can then be used to compare how well they perform. Additionally, a good diversity of solutions should be found using optimization methods. As a result, these optimization methods also need to include elements that improve and preserve variety. Last but not least, ArcheOpterix is currently available as a standalone Eclipse plug-in. However, a close interface with OSATE would be advantageous to enhance tool performance and have access to a bigger collection of assessment techniques.

CONCLUSION

In order to narrow the search to realistic areas of the search space, this study introduces a unique extension of multi-criteria architectural optimization that takes limitations for quality

needs into account. To enable the declaration of optimization objectives and quality requirements, we expanded the already-existing Quality of service Modeling Language. We convert the QML specifications into constraints for an optimization issue. To make the search concentrate on the feasible space, we apply the already employed constraint domination approach. The time it takes software architects to find worthwhile solutions can be cut down with this expansion. In a case study, we showed this aptitude. In comparison to the previous, unconstrained technique, our extension found solutions in the interesting portions of the objective space on average more than 35% faster. Using this method in various stages of the software architecture design process can be interesting. First, the method can be used following a preliminary phase of architectural development that focuses on functional needs (definition of components and interfaces). The optimization process can use this architecture as a starting point to enhance the non-functional features. A more high level decision can be made using the optimization to evaluate the potential of the various alternatives. This brings us to our second point: the optimization could already be used to support decisions during the architectural design. Last but not least, by modelling additional high level choices as transformations, these choices might be incorporated as degrees of freedom in the optimization process, allowing the optimization to explore various combinations of choices. By including limitations for quality attributes, which can convey that a given quality is sufficient and we are not interested in selling other qualities for further improvement in this area, we intend to expand the technique. Since we don't wish to treat these bounds as infeasible regions while optimizing architectures, they are not taken into consideration at this time. Additionally, we intend to do a more thorough validation that will enable us to draw more precise statistical conclusions about the outcomes of our effort.

REFERENCES

[1] A. Koziolek, D. Ardagna, en R. Mirandola, “Hybrid multi-attribute QoS optimization in component based software systems”, *J. Syst. Softw.*, 2013, doi: 10.1016/j.jss.2013.03.081.

[2] Y. He en M. A. Schnabel, “Computational architectural integrated implementation: A digital process from initial design to fabrication”, 2017. doi: 10.1201/9781315198101-87.

[3] R. Etemaadi en M. R. V. Chaudron, “Distributed optimization on super computers: Case study on software architecture optimization framework”, 2014. doi: 10.1145/2598394.2605686.

[4] S. Phommixay, M. L. Doumbia, en D. Lupien St-Pierre, “Review on the cost optimization of microgrids via particle swarm optimization”, *Int. J. Energy Environ. Eng.*, 2020, doi: 10.1007/s40095-019-00332-1.

[5] G. SHI, C. GUAN, D. QUAN, D. WU, L. TANG, en T. GAO, “An aerospace bracket designed by thermo-elastic topology optimization and manufactured by additive manufacturing”, *Chinese J. Aeronaut.*, 2020, doi: 10.1016/j.cja.2019.09.006.

[6] G. Guizzo, T. E. Colanzi, en S. R. Vergilio, “Applying design patterns in the search-based optimization of software product line architectures”, *Softw. Syst. Model.*, vol 18, no 2, bll 1487–1512, Apr 2019, doi: 10.1007/s10270-017-0614-9.

[7] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, en N. de Freitas, “Taking the Human Out of the Loop: A Review of Bayesian Optimization”, *Proc. IEEE*, vol 104, no 1, bll 148–175, Jan 2016, doi: 10.1109/JPROC.2015.2494218.

[8] J. S. Gray, J. T. Hwang, J. R. R. A. Martins, K. T. Moore, en B. A. Naylor, “OpenMDAO: an open-source framework for multidisciplinary design, analysis, and optimization”, *Struct. Multidiscip. Optim.*, vol 59, no 4, bll 1075–1104, Apr 2019, doi: 10.1007/s00158-019-02211-z.

[9] T. Akiba, S. Sano, T. Yanase, T. Ohta, en M. Koyama, "Optuna", in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, Jul 2019, bll 2623–2631. doi: 10.1145/3292500.3330701.

[10] R. Clune, J. J. Connor, J. A. Ochsendorf, en D. Kelliher, "An object-oriented architecture for extensible structural design software", *Comput. Struct.*, vol 100–101, bll 1–17, Jun 2012, doi: 10.1016/j.compstruc.2012.02.002.

[11] A. Krallis, P. Pladis, V. Kanellopoulos, V. Saliakas, V. Touloupides, en C. Kiparissides, "Design, Simulation and Optimization of Polymerization Processes Using Advanced Open Architecture Software Tools", in *Computer Aided Chemical Engineering*, 2010, bll 955–960. doi: 10.1016/S1570-7946(10)28160-9.

[12] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, en I. Meedeniya, "Software Architecture Optimization Methods: A Systematic Literature Review", *IEEE Trans. Softw. Eng.*, vol 39, no 5, bll 658–683, Mei 2013, doi: 10.1109/TSE.2012.64.

[13] M. Karakus en A. Durresi, "A survey: Control plane scalability issues and approaches in Software-Defined Networking (SDN)", *Comput. Networks*, vol 112, bll 279–293, Jan 2017, doi: 10.1016/j.comnet.2016.11.017.

[14] R. Li, R. Etemaadi, M. T. M. Emmerich, en M. R. V. Chaudron, "An evolutionary multiobjective optimization approach to component-based software architecture design", in *2011 IEEE Congress of Evolutionary Computation (CEC)*, Jun 2011, bll 432–439. doi: 10.1109/CEC.2011.5949650.

[15] B. G. Galuzzi, I. Giordani, A. Candelieri, R. Perego, en F. Archetti, "Hyperparameter optimization for recommender systems through Bayesian optimization", *Comput. Manag. Sci.*, 2020, doi: 10.1007/s10287-020-00376-3.

[16] M. Chmielewski, P. Pieczonka, M. Kukiełka, en T. Gutowski, "Software architecture optimization of mobile biomedical sensor-based tools providing analytical services for disease diagnostics and assistance", *Procedia Manuf.*, vol 44, bll 575–582, 2020, doi: 10.1016/j.promfg.2020.02.253.

CHAPTER 12

OBJECT-ORIENTED FRAMEWORK FOR SPECIFIC ARCHITECTURE OF SOFTWARE

Dr. Narendra Kumar Sharma, Assistant Professor
Department of Computer Science and Engineering, Sanskriti University, Mathura, Uttar Pradesh, India
Email Id- narendra@sanskriti.edu.in

ABSTRACT:

Architectural understanding has played a part in discussion on design, reuse, and adaptation for over a decade. The phrase has gained a lot of popularity in recent years, and efforts are being made to determine specific what is meant by architectural knowledge. The latest developments in architectural performance management are covered in this chapter. Following the results of a thorough literature study, we present four major perspectives on architectural knowledge. We describe major kinds of architectural knowledge and analyses four different outcomes for the business that have their roots in the abovementioned views, all of which are based on software architecture and knowledge organizational theory. State-of-the-art approaches take a more comprehensive stance and integrate various viewpoints in a single architectural knowledge management approach, in contrast with traditional approaches, which were limited to a single metaphysics when it came to tools, methods, and methodologies for architectonic performance management.

KEYWORDS:

Computer Software, IoT, Optical Sensors, Sensors, Wireless Sensors.

INTRODUCTION

Object-oriented frameworks are application skeletons, which reflect the basic characteristics of a particular application domain. When developing applications from such a domain, it will probably be more efficient to use such a framework rather than to start from scratch. A framework is a kind of 'instant program', that sometimes even may be a complete, ready-to-run application, but it will normally allow you to customize its look and feel to your own taste. Object-oriented frameworks is an attempt to capture the common characteristics within a certain application domain, and make them available for reuse. Only those characteristics that are common are hardwired into the code. Therefore, users of a framework are still free to handicraft those parts that give their applications the individual touch. The first more commonly used framework was the Model-View-Controller framework found in the Smalltalk-80 user interface. It allowed users to connect different visual presentations to the state of a Model object. These Views were automatically notified each time the state was changed, and were able to ask the Model for the new values of the properties they were representing[1].

A change in the Model object were thus immediately reflected on the screen. Today, frameworks are considered a very promising technology for reifying proven software designs, targeting particular functionality's such that the user interfaces and operating systems and particular application domains such that fire-alarm systems and real time avionics. Frameworks like MacApp; ET++; Interviews; ACE; Microsoft's MFC and DCOM; JavaSoft's RMI, AWT and Beans; OMG's CORBA play an increasingly important role in

contemporary software development. Early Frameworks were normally monolithic, i.e., object-oriented software architectures making up an entire application within some specific domain, but later versions are also restricting themselves to various subsystems. Due to the fact that these smaller frameworks are serving the role as design elements, they may seem to coincide with the Design Pattern concept, as specified in. There is, however, an important difference between the two, because these smaller grained frameworks still contain executable code, while design patterns are merely codeless descriptions of how to implement certain features. In addition, patterns are more universal tool in the sense that they are normally not tied to a particular application domain[2], [3].

Domain-Specific Development Environment

A domain-specific development environment (DSDE) supports the application development based on a DSSA. A DSDE has its own architecture that usually has three levels.

i. Productivity Tools

On top of a formal component model, there are a number of tools that facilitate a convenient application development, e.g., cogitation editors, semantic checkers, component repositories, generators, etc. An important tool is the constraint checker. Possible approaches to checking design constraints include attribute grammars, temporal logic, and a special type of first order logic.

ii. Formal Component Model

The formal component model is defined through the reference architecture and lies at the heart of a DSDE. The mapping of an application architecture onto the underlying layer is done by a generator. One has to decide whether to use compositional or transformational generator technology.

iii. Support Frameworks

Support frameworks implement the application component model. Both the frameworks and the reference architecture could be developed at the same time on an evolutionary basis. Support frameworks could already be portable, which would simplify the generation process. A critical aspect of the design for any large software system is its gross structure represented as a high-level organization of computational elements and interactions between those elements. Broadly speaking, this is the software architectural level of design. The structure of software has long been recognized as an important issue of concern. However, recently software architecture has begun to emerge as an explicit field of study for software engineering practitioners and researchers. Evidence of this trend is apparent in a large body of recent work in areas such as module interface languages, domain specific architectures, architectural description languages, design patterns and handbooks, formal underpinnings for architectural design, and architectural design environments.

What exactly do we mean by the term software architecture? As one might expect of a field that has only recently emerged as an explicit focus for research and development, there is currently no universally-accepted definition. Moreover, if we look at the common uses of the term architecture in software, we find that it is used in quite different ways, often making it difficult to understand what aspect is being addressed. Among the various uses are is that the architecture of a particular system, as in the architecture of this system consists of the following components and an architectural style, as in this system adopts a client-server architecture and the general study of architecture, as in \the papers in this journal are about architecture.

As definitions go, this is not a bad starting point. But definitions such as this tell only a small part of the story. More important than such explicit definitions, is the locus of effort in research and development that implicitly has come to define the field of software architecture. To clarify the nature of this effort it is helpful to observe that the recent emergence of interest in software architecture has been prompted by two distinct trends. The first is the recognition that over the years designers have begun to develop a shared repertoire of methods, techniques, patterns and idioms for structuring complex software systems[4], [5].

For example, the box and line diagrams and explanatory prose that typically accompany a high-level system description often refer to such organizations as a pipeline," a blackboard-oriented design or a client-server system. Although these terms are rarely assigned precise definitions, they permit designers to describe complex systems using abstractions that make the overall system intelligible. Moreover, they provide significant semantic content that informs others about the kinds of properties that the system will have: the expected paths of evolution, its overall computational paradigm, and its relationship to similar systems.

The second trend is the concern with exploiting specific domains to provide reusable frameworks for product families. Such exploitation is based on the idea that common aspects of a collection of related systems can be extracted so that each new system can be built at relatively low cost by instantiating the shared design. Familiar examples include the standard decomposition of a compiler which permits undergraduates to construct a new compiler in a semester, standardized communication protocols which allow vendors to interoperate by providing services at different layers of abstraction, fourth-generation languages which exploit the common patterns of business information processing, and user interface toolkits and frame- works which provide both a reusable framework for developing interfaces and sets of reusable components, such as menus, and dialogue boxes.

Generalizing from these trends, it is possible to identify four salient distinctions:

i. Focus of Concern

The first distinction is between traditional concerns about design of algorithms and data structures, on the one hand, and architectural concerns about the organization of a large system, on the other. The former has been the traditional focus of much of computer science, while the latter is emerging as a significant and different design level that requires its own notations, theories, and tools. In particular, software architectural design is concerned less with the algorithms and data structures used within modules than with issues such as gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives.

ii. Nature of Representation

The second distinction is between system description based on definition use structure and architectural description based on graphs of interacting components. The former modularizes a system in terms of source code, usually making explicit the dependencies between use sites of the code and corresponding definition sites. The latter modularizes a system as a graph, or configuration, of components and connectors. Components define the application-level computations and data stores of a system. Examples include clients, servers, filters, databases, and objects. Connectors define the interactions between those components. These interactions can be as simple as procedure calls, pipes, and event broadcast, or much more complex, including client-server protocols, database accessing protocols, etc.

iii. Instance Versus Style

The third distinction is between architectural instance and architectural style. An architectural instance refers to the architecture of a specific system. Box and line diagrams that accompany system documentation describe architectural instances, since they apply to individual systems. An architectural style, however, defines constraints on the form and structure of a family of architectural instances. For example, a pipe and filter architectural style might define the family of system architectures that are constructed as a graph of incremental stream transformers. Architectural styles prescribe such things as a vocabulary of components and connectors (for example, filters and pipes), topological constraints (for example, the graph must be acyclic), and semantic constraints (for example, filters cannot share state). Styles range from abstract architectural patterns and idioms (such as \client-server" or \layered" organizations), to concrete \reference architectures" (such as the ISO OSI communication model or the traditional linear decomposition of a compiler).

iv. Design Methods versus Architectures

A fourth distinction is between software design methods such as object-oriented design, structured analysis, and JSD and software a[6], [7]rchitecture. Although both design methods and architectures are concerned with the problem of bridging the gap between requirements and implementations, there is a significant difference in their scopes of concern. Without either software design methods or a discipline of software architecture design, the implementer is typically left to develop a solution using whatever ad hoc techniques may be at hand. Design methods improve the situation by providing a path between some class of system requirements and some class of system implementations. Ideally, a design method defines each of the steps that take a system designer from the requirements to a solution. The extent to which such methods are successful often depends on their ability to exploit constraints on the class of problems they address and the class of solutions they provide. One of the ways they do this is to focus on certain styles of architectural design. For example, object-oriented methods usually lead to systems formed out of objects, while others may lead more naturally to systems with an emphasis on data flow. In contrast, the field of software architecture is concerned with the space of architectural designs. Within this space object-oriented and data ow structures are but two of the many possibilities. Architecture is concerned with the trade-offs between the choices in this space the properties of different architectural designs and their ability to solve certain kinds of problems. Thus design methods and architectures complement each other: behind most design methods are preferred architectural styles, and different architectural styles can lead to new design methods that exploit them.

LITERATURE REVIEW

D. Le et al. stated that the Object-oriented domain-driven design (DDD) aims to iteratively develop software around a realistic model of the application domain, which both thoroughly captures the domain requirements and is technically feasible for implementation. The main focus of recent work in DDD has been on using a form of annotation-based domain specific language (aDSL), internal to an object-oriented programming language, to build the domain model. However, these works do not consider software modules as first-class objects and thus lack a method for their development. In this chapter, we tackle software module development with the DDD method by adopting a generative approach that uses aDSL. To achieve this, we first extend a previous work on module-based software architecture with three enhancements that make it amenable to generative development. We then treat module configurations as first-class objects and define an aDSL, named MCCL, to express module configuration

classes. To improve productivity, we define function MCCGEN to automatically generate each configuration class from the module's domain class. We define our method as a refinement of an aDSL-based software development method from a previous work. We apply meta-modelling with UML/OCL to define MCCL and implement MCCL in a Java software framework. We evaluate the applicability of our method using a case study and formally define an evaluation framework for module generativist. We also analyse the correctness and performance of function MCCGEN. MCCL is an aDSL for module configurations. Our evaluation shows MCCL is applicable to complex problem domains. Further, the MCCs and software modules can be generated with a high and quantifiable degree of automation. Conclusion: Our method bridges an important gap in DDD with a software module development method that uses a novel aDSL with a module-based software architecture and a generative technique for module configuration[8], [9].

B. Alshemaimri et al. stated that the Database code fragments exist in software systems by using Structured Query Language (SQL) as the standard language for relational databases. Traditionally, developers bind databases as back ends to software systems for supporting user applications. However, these bindings are low□level code and implemented to persist user data, so Object Relational Mapping (ORM) frameworks take place to database access details. Both approaches are prone to problematic database code fragments that negatively impact the quality of software systems. We survey problematic database code fragments in the literature and examine antipatterns that occur in low□level database access code using SQL and high□level counterparts ORM frameworks. We also study problematic database code fragments in different and popular software architectures such as Service□Oriented Architecture, Microservice Architecture, and Model View Controller. We create a novel categorization of both SQL schema and query antipatterns in terms of performance, maintainability, portability, and data integrity. This article reviews database antipatterns including SQL antipatterns and framework□specific antipatterns in terms of their impact on nonfunctional requirements such as performance, maintainability, portability, and data integrity.

M. Ghareb et al. stated that explores a new framework for calculating hybrid system metrics using software quality metrics aspect-oriented and object-oriented programming. Software metrics for qualitative and quantitative measurement is a mix of static and dynamic software metrics. It is noticed from the literature survey that to date, most of the architecture considered only the evaluation focused on static metrics for aspect-oriented applications. In our work, we mainly discussed the collection of static parameters, long with AspectJ-specific dynamic software metrics. The structure may provide a new direction for research while predicting software attributes because earlier dynamic metrics were ignored when evaluating quality attributes such as maintainability, reliability, and understandability of Asepect Oriented software. Dynamic metrics based on the fundamentals of software engineering are equally crucial for software analysis as are static metrics. A similar concept is borrowed with the introduction of dynamic software metrics to implement aspect-riented software development. Currently, we only propose a structure and model using static and dynamic parameters to test the aspect-oriented method, but we still need to validate the proposed approach[10], [11].

M. Amor et al. illustrated that the production of maintainable and reusable agents depends largely on how well the agent architecture is modularized. Most commercial agent toolkits provide an Object-Oriented (OO) framework, whose agent architecture does not facilitate separate (re)use of the domain-specific functionality of an agent from other concerns. This paper presents Mala, an agent architecture that combines the use of Component-based

Software Engineering and Aspect-Oriented Software Development, both of which promote better modularization of the agent architecture while increase at the architectural level. Malaca supports the separate (re)use of the domain-specific functionality of an agent from other communication concerns, providing explicit support for the design and configuration of agent architectures and allows the development of agent-based software so that it is easy to understand, maintain and reuse.

R. Taylor et al. stated that the objective of software development using domain-specific software architectures (DSSA) is reduction in time and cost of producing specific application systems within a supported domain, along with increased product quality, improved manageability, and positioning for acquisition of future business. Key aspects of the approach include software reuse based on parameterization of generic components and interconnection of components within a canonical solution framework. Viability of the approach depends on identification and deep understanding of a selected domain of applications. The DSSA approach, to be effectively applied, requires a variety of support tools, including repository mechanisms, prototyping facilities, and analysis tools. This curriculum module describes the DSSA approach, representative examples, supporting tools, and processes.

B. Belhomme et al. illustrated that the completely new ray tracing software has been developed at the German Aerospace Center. The main purpose of this software is the flux density simulation of heliostat fields with a very high accuracy in a small amount of computation time. The software is primarily designed to process real sun shape distributions and real highly resolved heliostat geometry data, which means a data set of normal vectors of the entire reflecting surface of each heliostat in the field. Specific receiver and secondary concentrator models, as well as models of objects that are shadowing the heliostat field, can be implemented by the user and be linked to the simulation software subsequently. The specific architecture of the software enables the provision of other powerful simulation environments with precise flux density simulation data for the purpose of entire plant simulations. The software was validated through a severe comparison with measured flux density distributions. The simulation results show very good accordance with the measured results.

R. Tu et al. illustrated that the Virtual Enterprise model affords the valid instruction for rapid establishing and successful running of Virtual Enterprise. However, authors perceive that low quality and low efficiency are serious restriction factor to the development of Virtual Enterprise model. In order to overcome above-mentioned embarrassment in Virtual Enterprise modeling, authors put forward applying software reuse technology and Domain Engineering theory to establishing the Domain Specific Software Architecture of Virtual Enterprise, then develop application system and establish the reusable component library in terms of Domain Specific Software Architecture of Virtual Enterprise. On the one hand, the quality and efficiency of modeling can be promoted remarkably. On the other hand, the model of Virtual Enterprise can be reused in the same domain.

J. Zhu et al. illustrated that the rapid development of technology, software is rapidly evolving with emerging applications. Chips that fail to adapt to software such that the application-specific integrated circuits, ASICs suffer from a short lifecycle and high nonrecurring engineering (NRE) costs. Meanwhile, as the projection of Moore's law and Dennard scaling are decreasing, energy efficiency has shown a diminishing return with new technologies. The computing capacity of general-purpose processors is limited due to power budgets. Consequently, future chips must jointly optimize flexibility, power efficiency, and ease of programmability. Reconfigurable chips combine the high flexibility of a general-purpose processor and high energy efficiency of ASIC by providing on-demand customization of their

architectures. This article thoroughly reviews the development and architecture of reconfigurable chips. Moreover, the future challenges of reconfigurable chips are analyzed. Based on these challenges, future directions are also discussed.

B. Senyapj et al. ststed that the Interior architectural education and practice employ various general-purpose software packages. This study problematizes that as none of these packages is developed specifically for interior architectural design process and purposes, both interior architecture education and market seek ways to fulfill their specific needs. It is argued that currently interior architecture does not fully benefit from digital opportunities. A specific software package for interior architecture will enable the discipline to put forth its assets and manifest its existence. Consequently, this study proposes a domain specific model for interior architectural software. Initially, general-purpose and domain specific computer aided architectural design (CAAD) software used in interior architecture are determined. Then, selected software packages are analyzed according to Szalapaj's set of features: 'drawing', 'transformation', 'view', 'rendering' and 'other'. Based on these analyses, domain specific requirements for interior architecture are obtained. Consequently, questionnaires and interviews are performed with interior architectural students and professionals in order to determine the user needs. Finally, based on the findings, a software model for interior architecture is proposed.

A. Gopalakrishnan et al. illustrated that the Software Engineering has evolved over many years but stays human centric as it relies significantly on the technical decisions made by humans. Modeling the problem statement and arriving at the architecture and design revolves in the minds of software architects and designers. Many of the decisions stays in architect's minds and are only present in the models. The abstraction structures in software design are deeper than in other disciplines, since the final design is program code. This distinction leads to software architecture and design a highly interwoven process. The early design decisions are otherwise termed architectural decisions which compose software architecture. The architectural decisions are at an intermediate abstraction level with higher probability of reuse, but still not effectively reused even within the same organization. The most effective cases of reuse in software is with architecture patterns and design patterns. The paper points to the fact that patterns are successfully reused due to the quality of the descriptions which include problem, solution pair and supporting example. The paper focuses on intra-organizational reuse, based on Domain Specific Software Architectures and the descriptions containing domain model, decision trees, architectural schema and rationale. It further tries to analyze three different use cases in the light of these elements and analyze if major hindrance of reuse is 'Rationale of decisions not well understood' than the commonly stated 'Not Invented here', supported with a survey of software engineers.

R. Weinreich et al. stated that the Software architecture is a central element during the whole software life cycle. Among other things, software architecture is used for communication and documentation, for design, for reasoning about important system properties, and as a blueprint for system implementation. This is expressed by the software architecture life cycle, which emphasizes architecture-related activities like architecture design, implementation, and analysis in the context of a software life cycle. While individual activities of the software architecture life cycle are supported very well, a seamless approach for supporting the whole life cycle is still missing. Such an approach requires the integration of disparate information, artifacts, and tools into one consistent information model and environment. In this article we present such an approach. It is based on a semi-formal architecture model, which is used in all activities of the architecture life cycle, and on a set of extensible and integrated tools supporting these activities. Such an integrated approach provides several benefits. Potentially

redundant activities like the creation of multiple architecture descriptions are avoided, the captured information is always consistent and up-to-date, extensive tracing between different information is possible, and interleaving activities in incremental development and design are supported.

O. Pedreira et al. illustrated that the gamification has been applied in software engineering to improve quality and results by increasing people's motivation and engagement. A systematic mapping has identified research gaps in the field, one of them being the difficulty of creating an integrated gamified environment comprising all the tools of an organization, since most existing gamified tools are custom developments or prototypes. In this paper, we propose a gamification software architecture that allows us to transform the work environment of a software organization into an integrated gamified environment, i.e., the organization can maintain its tools, and the rewards obtained by the users for their actions in different tools will mount up. We developed a gamification engine based on our proposal, and we carried out a case study in which we applied it in a real software development company. The case study shows that the gamification engine has allowed the company to create a gamified workplace by integrating custom-developed tools and off-The-shelf tools such as Redmine, TestLink, or JUnit, with the gamification engine. Two main advantages can be highlighted: (i) our solution allows the organization to maintain its current tools, and (ii) the rewards for actions in any tool accumulate in a centralized gamified environment.

C. Venters et al. ststed that the Context Modern societies are highly dependent on complex, large-scale, software-intensive systems that increasingly operate within an environment of continuous availability, which is challenging to maintain and evolve in response to the inevitable changes in stakeholder goals and requirements of the system. Software architectures are the foundation of any software system and provide a mechanism for reasoning about core software quality requirements. Their sustainability the capacity to endure in changing environments is a critical concern for software architecture research and practice. Problem Accidental software complexity accrues both naturally and gradually over time as part of the overall software design and development process. From a software architecture perspective, this allows several issues to overlap including, but not limited to: the accumulation of technical debt design decisions of individual components and systems leading to coupling and cohesion issues; the application of tacit architectural knowledge resulting in unsystematic and undocumented design decisions; architectural knowledge vaporization of design choices and the continued ability of the organization to understand the architecture of its systems; sustainability debt and the broader cumulative effects of flawed architectural design choices over time resulting in code smells, architectural brittleness, erosion, and drift, which ultimately lead to decay and software death. Sustainable software architectures are required to evolve over the entire lifecycle of the system from initial design inception to end-of-life to achieve efficient and effective maintenance and evolutionary change. Method This article outlines general principles and perspectives on sustainability with regards to software systems to provide a context and terminology for framing the discourse on software architectures and sustainability. Focusing on the capacity of software architectures and architectural design choices to endure over time, it highlights some of the recent research trends and approaches with regards to explicitly addressing sustainability in the context of software architectures. Contribution The principal aim of this article is to provide a foundation and roadmap of emerging research themes in the area of sustainable software architectures highlighting recent trends, and open issues and research challenges.

J. W. Kruize et al. stated that the smart farming is a management style that includes smart monitoring, planning and control of agricultural processes. This management style requires

the use of a wide variety of software and hardware systems from multiple vendors. Adoption of smart farming is hampered because of a poor interoperability and data exchange between ICT components hindering integration. Software Ecosystems is a recent emerging concept in software engineering that addresses these integration challenges. Currently, several Software Ecosystems for farming are emerging. To guide and accelerate these developments, this paper provides a reference architecture for Farm Software Ecosystems. This reference architecture should be used to map, assess design and implement Farm Software Ecosystems. A key feature of this architecture is a particular configuration approach to connect ICT components developed by multiple vendors in a meaningful, feasible and coherent way. The reference architecture is evaluated by verification of the design with the requirements and by mapping two existing Farm Software Ecosystems using the Farm Software Ecosystem Reference Architecture. This mapping showed that the reference architecture provides insight into Farm Software Ecosystems as it can describe similarities and differences. A main conclusion is that the two existing Farm Software Ecosystems can improve configuration of different ICT components. Future research is needed to enhance configuration in Farm Software Ecosystems.

DISCUSSION

The three approaches that have been discussed in the previous sections, according to the criteria, use the same terminology, only the names of the terms change, showing the lack of a unified language. They share the fact of considering that the quality characteristics wanted or expected high-level quality characteristics in a software product must be defined and quantified measured in order to be assured. External and internal quality views are considered. The high-level characteristics, that may affect the exit or failure of the final system, cannot in general be directly measured. They must be “refined” in order to get the measurable aspects. Moreover, these measures are used to link or relate the low-level characteristics, which are measurable, with the high-level characteristics. In this way, a trade-off to detect the dependencies among these characteristics is established. The definition of these links is always performed empirically or on the basis of experience. On the other hand, the approaches differ mostly on the stage of development where the quality model is applied. However, an important issue is that at design stage, all the approaches could be used. From our point of view, this stage is very important because it concerns the definition of the system architecture, characterized by non-functional properties. Nevertheless the ABAS approach, specific to this stage, does not offer any guideline. Finally, an important research issue is the extension of the software development methods that do not consider explicitly a quality model, with one of the three quality model approaches studied. Those offering guidelines should be better candidates, or the use of an extended ABAS with ISO 9126 or Dromey’s design model. Moreover, since these approaches lack a common language, the specification of the quality models studied using notational standards, such as UML (Unified Modelling Language) should be considered. In UML is used to model architectures of real-time systems, where the selection of an architecture meeting precise quality requirements is crucial.

CONCLUSION

This paper presents an approach to integrate frameworks with domain specific languages (DSL). We argue that DSLs allows the domain expert to formalize the specification of a software solution immediately without worrying about implementation decisions and the framework complexity. The code for the variation points is specified in DSLs that are transformed (or compiled) to generate the framework instantiation code. During the transformation the framework instantiation restrictions may be verified. The case studies have shown that the proposed approach may enhance very much the instantiation process. It is

important to note that DSLs can be transformed into other DSLs, thus creating a domain network, in a way similar to that described in, providing an easy implementation path for new DSLs. An approach for the derivation of the framework instantiation restrictions based on UML specifications is shown in, as well as tool support for the transformations. We are now working on a more elaborated version of the supporting environment, based on UML case tools and specific transformational systems.

REFERENCES

[1] M. Ozkaya en F. Erata, "A survey on the practical use of UML for different software architecture viewpoints", *Inf. Softw. Technol.*, vol 121, bl 106275, Mei 2020, doi: 10.1016/j.infsof.2020.106275.

[2] T. Gu, M. Lu, L. Li, en Q. Li, "An Approach to Analyze Vulnerability of Information Flow in Software Architecture", *Appl. Sci.*, vol 10, no 1, bl 393, Jan 2020, doi: 10.3390/app10010393.

[3] A. Baabad, H. B. Zulzalil, S. Hassan, en S. B. Baharom, "Software architecture degradation in open source software: A systematic literature review", *IEEE Access*. 2020. doi: 10.1109/ACCESS.2020.3024671.

[4] M. M. Soto-Cordova, S. León-Cárdenas, K. Huayhuas-Caripaza, en R. M. Sotomayor-Parian, "Proposal for a software architecture as a tool for the fight against corruption in the regional governments of Peru", *Int. J. Adv. Comput. Sci. Appl.*, 2020, doi: 10.14569/IJACSA.2020.0110786.

[5] S. Farshidi, S. Jansen, en J. M. van der Werf, "Capturing software architecture knowledge for pattern-driven design", *J. Syst. Softw.*, 2020, doi: 10.1016/j.jss.2020.110714.

[6] S. Moaven en J. Habibi, "A fuzzy-AHP-based approach to select software architecture based on quality attributes (FASSA)", *Knowl. Inf. Syst.*, 2020, doi: 10.1007/s10115-020-01496-7.

[7] *et al.*, "The Principle of Architecture First in Software Project Management Minimizes the Cost of Software Development Process: A Review", *Int. J. Innov. Technol. Explor. Eng.*, 2020, doi: 10.35940/ijitee.a8154.1110120.

[8] O. Sievi-Korte, I. Richardson, en S. Beecham, "Software architecture design in global software development: An empirical study", *J. Syst. Softw.*, 2019, doi: 10.1016/j.jss.2019.110400.

[9] J. Cruz-Benito, F. J. García-Peñalvo, en R. Therón, "Analyzing the software architectures supporting HCI/HMI processes through a systematic review of the literature", *Telemat. Informatics*, 2019, doi: 10.1016/j.tele.2018.09.006.

[10] Q. Q. G. Wuniri, X. P. Li, S. L. Ma, J. H. Lü, en S. Q. Zhang, "Modelling and Verification of High-order Typed Software Architecture and Case Study", *Ruan Jian Xue Bao/Journal Softw.*, 2019, doi: 10.13328/j.cnki.jos.005749.

[11] C. Ellwein, A. Elser, en O. Riedel, "Production planning and control systems - A new software architecture Connectivity in target", 2019. doi: 10.1016/j.procir.2019.02.089.

CHAPTER 13

DIALYSIS SOFTWARE ARCHITECTURE DESIGN EXPERIENCES

Dr. Abhishek Kumar Sharma, Assistant Professor
Department of Computer Science and Engineering, Sanskriti University, Mathura, Uttar Pradesh, India
Email Id- abhishek.sharma@sanskriti.edu.in

ABSTRACT:

The demonstration of a method for software maintainability prognosis during software architecture design. The technique employs the need definition as its first input, followed by the design of the infrastructure, software engineers' skills, and, perhaps, historical data. It then predicts average typical effort for a maintenance task. The method uses scenario to clarify the maintainability constraints and examine the architecture to foresee maintainability. The approach is demonstrated through the design of the programming language for a hemodialysis machine and is based on substantial expertise in both architecture design phase design. Future research will involve experimentation for the method's appraisal and validation.

KEYWORDS:

Computer Science, Information Technology, Software Engineering, Software Design.

INTRODUCTION

Software architecture design is an art and today only a few, sketchy methods exist for designing software architecture. The challenge facing the software architect is to find an optimal balance in software qualities to make the resulting application able to fulfil its quality requirements. The tools and techniques available for the software architect are scarce, i.e. design patterns, software architecture patterns, and various ADLs with accompanying analysis tools. In this list of tools and techniques we are missing time-proven methods for evaluation and assessment of architecture and software architecture design methods. Proposals exist, but none has been proven by time. In our work towards better and more efficient methods for design and assessment of software architecture we have participated in research and design projects with a number of industry partners. These projects have given us some hard-earned hands on experience of what really makes the design of software architecture difficult [1].

Dialysis systems present an area in the domain of medical equipment where competition has been increasing drastically during recent years. The aim of a dialysis system is to remove water and certain natural waste products from the patient’s blood. Patients that have, generally serious, kidney problems and consequently produce little or no urine use this type of system. The dialysis system replaces this natural process with an artificial one. The research project aimed at designing a new software architecture for the dialysis machines produced by Althin Medical. The software of the existing generation products was exceedingly hard to maintain and certify. The partners involved in the project were Althin Medical, EC-Gruppen and the University of Karlskrona/Ronneby. The goal for EC-Gruppen was to study novel ways of constructing embedded systems, whereas our goal was to study the process of designing software architecture and to collect experiences [2].6

An overview of a dialysis system is presented in figure 1. The system is physically separated into two parts by the dialysis membrane. On the left side the dialysis fluid circuit takes the water from a supply of a certain purity (not necessarily sterile), dialysis concentrate is added using a pump. A sensor monitors the concentration of the dialysis fluid and the measured value is used to control the pump. A second pump maintains the flow of dialysis fluid, whereas a third pump increases the flow and thus reduces the pressure at the dialysis fluid side. This is needed to pull the waste products from the patient's blood through the membrane into the dialysis fluid. A constant flow of dialysis fluid is maintained by the hydro mechanic devices that ensure exact and steady flow on each side [3].

On the right side of Figure 1, the extra corporal circuit, i.e. the blood part, has a pump for maintaining a specified blood flow on its side of the membrane. The patient is connected to this part through two needles usually located in the arm that take blood to and from the patient. The extra corporal circuit uses a number of sensors, e.g. for identifying air bubbles, and actuators, e.g. a heparin pump to avoid cluttering of the patients' blood while it is outside the body. However, these details are omitted since they are not needed for the discussion in the paper.

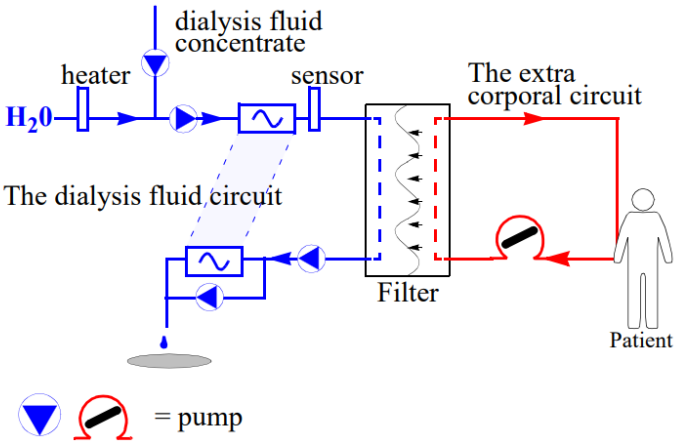

Figure 1: Represented that the Schematic of Dialysis Machine [4]**.**

The dialysis process, or treatment, is by no means a standard process. A fair collection of treatments exists including, for example, Haemo Dialysis Filtration (HDF) and Ultra Filtration (UF) and other variations, such as single needle/single pump, double needle/single pump. Treatments are changed due to new research results but also since the effectiveness of a particular treatment decreases when it is used too long for a patient. Although the abstract function of a dialysis system is constant, a considerable set of variations exists already. Based on experience the involved company anticipates several additional changes to the software, hardware and mechanical parts of the system that will be necessary in response to developments in medical research.

Legacy Architecture

As an input to the project, the original application architecture was used. This architecture had evolved from being only a couple of thousand lines of code very close to the hardware to close to a hundred thousand lines mostly on a higher level than the hardware API. The system runs on a PC-board equivalent using a real-time kernel/operating system. It has a graphical user interface and displays data using different kinds of widgets. It is a quite complex piece of software and because of its unintended evolution, the structure that was once present has deteriorated substantially. The three major software subsystems are the Man Machine Interface (MMI), the Control System, and the Protective system as mention in Figure 2.

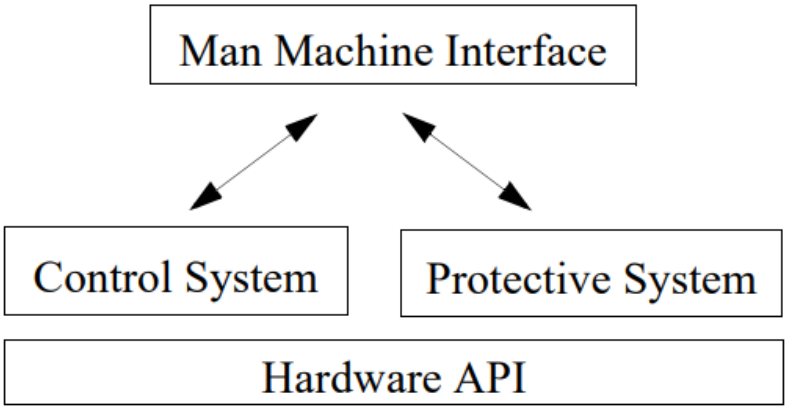

Figure 2: Represented that the Legacy System Decomposition [5]**.**

The MMI has the responsibilities of presenting data and alarms the user, i.e. a nurse, and getting input, i.e., commands or treatment data, from the user and setting the protective and control system in the correct modes. The control system is responsible for maintaining the values set by the user and adjusting the values according to the treatment selected for the time being. The control system is not a tight-loop process control system, only a few such loops exists, most of them low-level and implemented in hardware.

The protective system is responsible for detecting any hazard situation where the patient might be hurt. It is supposed to be as separate from the other parts of the system as possible and usually runs on an own task or process. When detecting a hazard, the protective system raises an alarm and engages a process of returning the system to a safe-state. Usually, the safe-state is stopping the blood flow or dialysis-fluid flow. The documented structure of the system is no more fine-grained than this and to do any change impact analysis, extensive knowledge of the source code is required.

Maintainability

Our partner company's actual hemo dialysis equipment has a propensity of being difficult to maintain. Software has become more complicated to comprehend and manage with each revamped edition that includes bug fixes and function additions. The new dialysis system family's software design need, among other things, be far more maintenance than the current systems when it comes of corrective but especially adaptive maintenance.

i. The problem has been challenging in existing systems because it has been challenging to recognise and see dependencies between numerous software components.

ii. A steady supply of new and likely to experience greater sparks adaptive maintenance. Examples include novel medicines, control algorithms, and safety standards in addition to new mechanical parts like compressors, heaters, and AD/DA converters. The system must be upgraded with all these additional criteria as quickly as possible. It is practically always necessary to alter the software when making modifications to the system's mechanics or hardware. All of these additions have harmed fundamental structure of the software inside this current system, making it difficult to maintain and making subsequent updates more difficult to implement. The system had to be scalable to maintain, which was possibly its most necessary feature.

Reusability

It should be possible to reuse the software created for the dialysis machine. Hemodialysis machines already come in a variety of types, and as the market demands more customization, there will likely be a greater need for more hemodialysis models. Of course, there should be a high level of reuse between various hemodialysis equipment models.

Safety

As an extension of the patient's blood flow, hemodialysis devices have a number of potential hazards for the patient, some of which could be fatal. The system has exceptionally stringent safety criteria since the patient's safety is given a very high priority. The hemodialysis system should recognise the emergence of such conditions and restore the dialysis machine and the patient to a state that poses no threat to the patient, i.e. a safe-state, rather than exposing the dialysis patient to any risks. Such preventative steps to reach a safe state include actions like stopping the dialysis fluid if concentrations are outside of range and stopping the blood flow if air bubbles are found in the extracorporeal system. By defining a variety of hazard circumstances, related thresh-hold values, and the procedure to be followed in order to reach the safe-state, the safety requirements standard for haemodialysis devices has already partially translated this requirement into functional requirements. A few other safety-related factors, though, are not addressed. For instance, if a pump's communication breaks down, the system should be able to assess the risk, take the appropriate action to make the patient safe, and alert the nurse that a service technician is needed.

Real-timeliness

In the sense that decisions must be made within a few microseconds during normal operation, hemodialysis is not a very time-critical process by nature. Once the flows, concentrations, and temperatures are established during a normal treatment, the process merely has to be monitored. However, when a risk or fault scenario materialises, response time becomes crucial. The hemodialysis machine must react very fast to restore the system to a safe state in the event of a detected hazard, such as air being detected in the extra corporal unit. The safety standard for hemodialysis devices includes timings for these situations.

Demonstrability

The safety of the patient is crucial, as was already said. An impartial certification organization must certify each structure in order to make sure that hemodialysis machines marketed comply with safety laws. Every new update of the software requires a new certification process, which significantly raises the price of creating and maintaining haemodialysis devices. Making it simple to demonstrate that the software performs the necessary safety functions as required will help to lower the cost of certification. We refer to this need as demonstrability.

Design Techniques

For the project, we utilised the architectural design technique developed by our research team, Architecture and Composition of Software (ARCS), which is shown in Figure 3. The requirement definition is where the ARCS process gets started. The architect creates an architecture from this input data that is mostly focused on the functional needs. The initial archetypes are present in this architecture's first iteration. The way we use the word "archetype" and define it is different. We define an archetype as a fundamental abstraction that is used to model the architecture of an application. Archetypes typically change as a

result of design revisions. Various assessment methods are used to evaluate the architecture. The ARCS method employs four types of evaluation:

i. Scenario Based Evaluation

Using this method, software characteristics are presented as normal or probable scenarios. Maintainability, for instance, may be defined as change scenarios outlining probable modifications, with the implementation of the changes requiring the least amount of architecture modification possible.

ii. Mathematical Modelling (including metrics & statistics)

This a technique were product and process data are used to make predictions about the potential qualities of a resulting product or task.

iii. Simulation

This approach resembles scenarios, but it is more suited to dynamic aspects like performance and dependability. The software quality characteristic is predicted using the performance of the architecture, which has been studied in a simulation environment. For instance, safety may be assessed by modelling how the haemodialysis architecture would operate in various hazard scenarios.

iv. Experience Based Reasoning

This method is the most popular and works well as an addition to other methods. Experience designers frequently recognize designs which aren't sufficiently meeting specific quality needs. Further investigation may be carried out using the other, more objective procedures, based on the original identification.

The architecture design is complete if the results indicate that the potential for the software attributes is adequate. The inspection of the basic architecture typically identifies a number of flaws. By utilizing a selection of accessible transformations, the designer changes the architecture into a new version to address issues. The ARCS approach divides transformations into five categories:

i. When an architecture style is used, the general structure is altered.

ii. Applying an architecture pattern, such as Periodic Objects, adds specific behavioural guidelines to the architecture.

iii. Only a small portion of the architecture is affected by the application of design patterns.

iv. Transforming a requirement for quality into functionality, such as handling robustness by integrating exception handling.

v. Distributing Requirements. For example, response time requirements on the whole system may be decomposed into response time requirements for individual elements.

These transformations only reorganize the domain functionality and affect only the software quality attributes of an architecture. After a set of transformations, architecture evaluation is repeated and the process is iterated until the quality requirements are fulfilled. The method may appear similar to the spiral model presented in, but some important differences in focus and scope exist. In Figure 3 shown the Repeated Evaluation for Control of the Design.

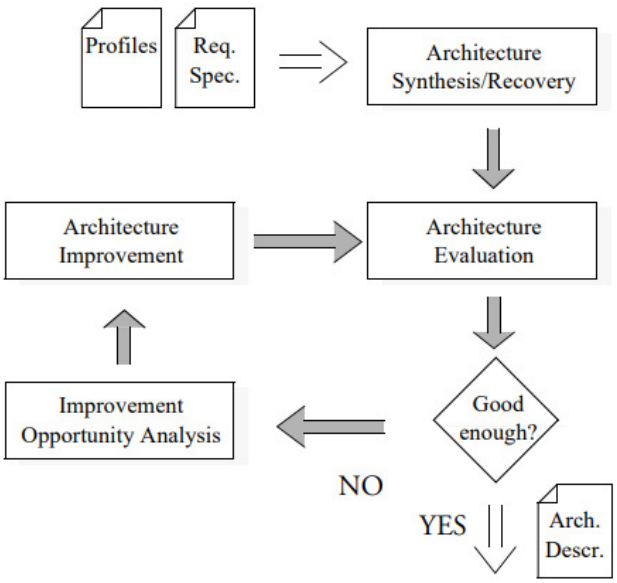

Figure 3: Represented that the Repeated Evaluation for Control of the Design [6]**.**

Too Large Assessment Efforts

There are research communities that have generated thorough assessment and assessment strategies for each of the primary quality needs of the dialysis system architecture. In our view, these methods have three serious weaknesses when it comes to architecture examination. First, they disregard other, just the same as significant features in favor of focusing on one. Second, their examinations frequently take an exorbitant amount of time to conduct because of the deep and elaborate they are. Finally, the methodologies generally call for specific knowledge that is not yet provided during the architecture planning process and are typically reserved for the succeeding design phases.

As a result, during architectural design, evaluation is conducted out in an ad-hoc, intuition-based direction without support from more methods and techniques. This is due to the fact that software architects typically must balance a range of quality requirements, lack the data required by the parameters listed, and work under time constraints. Even though some research has been conducted in this area, there is still a huge market for simple architecture methodological approaches for the various quality criteria, ideally with embedded tool support.

Architecture Abstractions outside Application Domain

Traditional object oriented design techniques, for example, offer pointers and recommendations for selecting the ideal abstractions for the object-centered design. As a general rule, one should represent the key notions from the issue domain. The architectural abstractions, or paradigms, employed in the final version, however, did not exist in the particular application, as we noticed in this project and a number of other projects. Instead, even during design rounds, these archetypes formed and constituted abstract domain functionality arranged to maximize the driving quality attributes. We discovered that the finest appropriate abstraction was discovered when a true comprehension of the notion and its relationships emerged. As a sample, we made use of the domain principles from the first design iteration that we had gathered from reading the documentation and speaking with domain experts. As we learned more about the demands and anticipated behaviour of the system, we repeated several times the design, switching the abstractions utilised in the architecture design from domain ideas to archetypes that take the demands for quality into

account. We became increasingly conscious of how the performance requirements would need to cooperate during the design rounds. For instance, even while employing design patterns might aid in flexibility in some scenarios, demonstrability and real-timeliness were difficult to ensure necessitating the discovery of alternatives abstractions.

LITERATURE REVIEW

P. Reyes-Delgado et al. illustrated that the importance of Software Architecture (SA) design has been acknowledged as a very important factor for a high-quality software development. Different efforts in both industry and academia have produced multiple system development methodologies (SDMs) that include SA design activities. In addition, standardization bodies have defined different recommendations regarding Software Architecture design. However, in industry Software Architecture best practices are currently poorly employed. This fact constrains the benefits that industry can potentially obtain from Software Architecture design in software development. In this paper, we analyze the degree to which the four main recognized SDMs-RUP (Rational Unified Process), MSF (Microsoft Solutions Framework), MBASE (Model-Based System Architecting and Software Engineering), and RUP-SOA (Rational Unified Process for Service-oriented Architecture) - adhere to the best practices of Software Architecture design. Our analysis points out some of the most important strengths and weaknesses regarding Software Architecture design and highlights some of the most relevant issues of Software Architecture design that need to be incorporated into such methodologies [7].

M. Jaiswal et al. illustrated that the software architecture defined as strategic design of an activity concerned with global requirements and its solution is implemented such as programming paradigms, architectural styles, component-based software engineering standards, architectural patterns, security, scale, integration, and law-governed regularities. Functional design, also described as tactical design, is an activity concerned with local requirements governing what a solution does such as algorithms, design patterns, programming idioms, refactoring, and low-level implementation. In this paper I would like to introduce some concepts of software architecture, and software design as well as relationship between them [8].

J. Bishung et al. Illustrated that the Software architecture and design is an important component in the software engineering field. This aspect of software engineering covers the functional and non-functional requirements of any system being proposed to be developed, while software architecture deals with non-functional requirements, software design entails the functional requirements. The objective of this paper is to critically analyze current topics in Software architecture and design. The method of analysis involved the use of inclusion and exclusion criteria of papers published in journals and conferences. From the analysis, the result showed that, of 35 papers used in analysis, 34.3% discussed stakeholders' involvement and decisions in software design. 17.1% for design quality, 20% examined software reuse while 11.4% discussed software evaluation and 8.6% of papers reviewed discussed software management, evolution and software development life cycle each which should be more focused as it is the fundamentals of software design and architecture. From the analysis derived, stakeholder's involvement and decision in software design is an integral part in software building for effective use. Thereby making researchers dwell more on the topic. The least discussed topics was due to the expectations of researchers. Expecting readers to have a fore knowledge of the fundamentals of design which includes software management, evolution and software development life cycle [9].

M. Roldan et al. stated that the Software architecture design is an interactive, complex, decision-making process. Such a design process involves the exploration, evaluation, and composition of design alternatives. Increasingly, new computer-aided tools are available to help designers in these complex activities. However, these tools do not know how design is actually done, in other words, by means of which design activities the final artefact was obtained. In fact, the architectural design knowledge exclusively rests in the mind of designers, and there is an urgent need to move it, as much as possible, to a computer-supported environment that enables the capture of this type of knowledge. This contribution addresses this need by introducing a model for capturing how products under development are generated and transformed along the software architecture design process. The proposed model follows an operational perspective, where architectural design decisions are modelled by means of sequences of operations that are applied on the design products. Situation calculus is used to formally express the existence of an object in a given state of a design process. In addition, this formalism allows us expressing without ambiguities when an operation can be performed in a specific state of the design process [10].

N. Chondamrongkul et al. illustrated that the analyzing security in the architecture design of modern software systems is a challenging task. Emerging technologies utilized in building software systems may pose security threats, so software engineers need to consider both the structure and behavior of architectural styles that employ these supporting technologies. This paper presents an automated approach to security analysis that helps to identify security characteristics at the architectural level. Key techniques used by our approach include the use of metrics, vulnerability identification and attack scenarios. Our modelling is expressive in defining architectural styles and security characteristics. Our analysis approach gives insightful results that allow software engineers to trace through the design to find parts of the system that may be impacted by attacks. We have developed an analysis tool that allows user to seamlessly model the software architecture design and analyses security. The evaluation has been conducted to assess the accuracy and performance of our approach. The results show that our analysis approach performs reasonably well to analyses the security in the architectural design[11].

O. Sievi-Korte et al. ststed that the Global Software Development (GSD), the additional complexity caused by global distance requires processes to ease collaboration difficulties, reduce communication overhead, and improve control. How development tasks are broken down, shared and prioritized is key to project success. While the related literature provides some support for architects involved in GSD, guidelines are far from complete. This paper presents a GSD Architectural Practice Framework reflecting the views of software architects, all of whom are working in a distributed setting. In-depth interviews with architects from seven different GSD organizations revealed a complex set of challenges and practices. We found that designing software for distributed teams requires careful selection of practices that support understanding and adherence to defined architectural plans across sites. Teams used Scrum which aided communication, and Continuous Integration which helped solve synchronization issues. However, teams deviated from the design, causing conflicts. Furthermore, there needs to be a balance between the self-organizing Scrum team methodology and the need to impose architectural design decisions across distributed sites. The research presented provides an enhanced understanding of architectural practices in GSD companies. Our GSD Architectural Practice Framework gives practitioners a cohesive set of warnings, which for the most part, are matched by recommendations [12].

O. Sievi-Korte et al. stated that the Software architecture design is a complex task, requiring handling and balancing multiple tradeoffs. In this paper, the potential of genetic algorithms

(GAs) in automated software architecture design is explored, assuming that software architecture is constructed of patterns. We have implemented different techniques based on GAs with variations on algorithmic operations and evaluation functions. We perform an extensive case study using a real framework system as a benchmark. The solutions are analyzed and compared with the man-made design of the framework. Our purpose is to study what kind of pattern configurations the algorithm is able to produce, how close they are to the ones used by a human and whether modifying the algorithm gives better solutions. Results show that 60% of the patterns proposed by the algorithm can be seen as well-placed, but there are big differences between the techniques and certain patterns are significantly more difficult for the algorithm to handle than others [13].

R. Chen et al. illustrated that the Animation education in the new media era is moving toward the goal of cultivating high-end talents. The development of an architecture-oriented animation studies platform provides guarantee for the training of talents in terms of teaching quality. This research uses the Internet as the medium and mobile phones and computer clients as the main technology platforms, starting from the software architecture and constructing the system model of the animation studies platform according to the Structure-Behavior Coalescence (SBC) method. The core theme of Model-Based Systems Engineering (MBSE) is a modeling language with model consistency of systems structure and systems behavior. This paper developed Structure-Behavior Coalescence State Machine (SBC-SM) as the formal language for the MBSE animation studies platform design model singularity. The model consistency will be fully guaranteed in the MBSE animation studies platform design when the SBC state machine approach is adopted. It not only improves the efficiency of platform development but also reduces the difficulty and risk of platform development[14].

R. Kazman et al. ststed that the Architecture analysis and design methods such as ATAM, QAW, ADD and CBAM have enjoyed modest success and are being adopted by many companies as part of their standard software development processes. They are used in the lifecycle, as a means of understanding business goals and stakeholders concerns, mapping these onto an architectural representation, and assessing the risks associated with this mapping. These methods have evolved a set of shared component techniques. In this paper we show how these techniques can be combined in countless ways to create needs-specific methods in an agile way. We demonstrate the generality of these techniques by describing a new architecture improvement method called APTIA (Analytic Principles and Tools for the Improvement of Architectures). APTIA almost entirely reuses pre-existing techniques but in a new combination, with new goals and results. We exemplify APTIA's use in improving the architecture of a commercial information system [15].

D. Falessi et al. illustrated that the architecture of a software-intensive system can be defined as the set of relevant design decisions that affect the qualities of the overall system functionality; therefore, architectural decisions are eventually crucial to the success of a software project. The software engineering literature describes several techniques to choose among architectural alternatives, but it gives no clear guidance on which technique is more suitable than another, and in which circumstances. As such, there is no systematic way for software engineers to choose among decision-making techniques for resolving tradeoffs in architecture design. In this article, we provide a comparison of existing decision-making techniques, aimed to guide architects in their selection. The results show that there is no "best" decision-making technique; however, some techniques are more susceptible to specific difficulties. Hence architects should choose a decision-making technique based on the difficulties that they wish to avoid. This article represents a first attempt to reason on meta-decision-making, that is, the issue of deciding how to decide [16].

DISCUSSION

Some software architects have overemphasised structure and behaviours rather than the choices that result in such structures and behaviours as a result of the building analogy. Structure and behaviour are not unimportant; rather, they are the outcomes of a cognitive process that must be preserved for the system to evolve sustainably through time. Just as crucial as understanding what someone did is understanding why they did it. If the code is well-organized and annotated, it should be simple to see what they did, but the why is frequently forgotten. Due to the fact that most architectural decisions in software are compromises between competing alternatives, it can be difficult to determine the worth of an alternative until you try a few and evaluate how they perform. It is frequently more helpful to know what was tried and failed than what worked. According to an old proverb, most experience is gained from making poor decisions.This is another reason why software architects still need to be developers without creating and testing something, they can't comprehend or foresee the dynamics at play in a system. Software Architect has to be more than just an honorarium for developers who have stopped working on new projects but yet have information that the company deems beneficial. The process of architecting necessitates a thorough understanding of a system in order to formulate valid hypotheses about quality attributes, as well as the skills necessary to create code and design tests or closely collaborate with team members who are capable of doing so. The field of software architecture need a revamp. Its reputation is harmed by several outdated beliefs about the issues it must address and the best way to do so. The essence of a continuous approach to software architecture is viewing it as a continuous activity focused on developing hypotheses about how the system will fulfil quality criteria and then using empiricism to demonstrate that the system achieves them. Taking control of software design away from groups of individuals who aren't developers and placing it in the hands of those who can make it real and usable, the developers, is another shift that needs to be made. The robustness and sustainability we require from today's applications won't come about until then.

CONCLUSION

This paper presents the architectural design of a hemodialysis system as well as the lessons discovered during the development of the architecture. The following are the primary learnings from the project. First, it can be challenging to evaluate the design for these features and balance quality traits because quality criteria are frequently provided in isolation. Second, the evaluation methods created by the various research communities researching a particular quality attribute, such as performance or reusability, are typically designed for later stages of development and occasionally require excessive work and data that is not available during architecture design. Third, domain analysis cannot be used to infer from the application domain the archetypes that make up a software architecture. Instead, the archetypes stand for sections of domain functionality that have been improved to meet the standards for driving quality. Fourth, we discovered that the design process is iterative by nature, group design meetings are much more productive than individual meetings with architects, and documentation of design decisions is crucial for preserving the design rationale. Fifth, the aesthetics of architectural designs at least have an inter-subjective perception, and both an instinctively appealing design and its absence have proven to be effective indicators. Sixth, the natural tendency of software engineers to perfect solutions and the effort necessary for architecture assessment made it difficult to determine when one was finished with the architectural design. Finally, it might be quite challenging to fully document a software architecture. The architectural plan described in the part before gives some context to our experiences.

REFERENCES

[1] H. van Vliet en A. Tang, "Decision making in software architecture", *J. Syst. Softw.*, vol 117, bll 638–644, Jul 2016, doi: 10.1016/j.jss.2016.01.017.

[2] O. Sievi-Korte, S. Beecham, en I. Richardson, "Challenges and recommended practices for software architecting in global software development", *Inf. Softw. Technol.*, 2019, doi: 10.1016/j.infsof.2018.10.008.

[3] A. Banijamali, P. Heisig, J. Kristan, P. Kuvaja, en M. Oivo, "Software Architecture Design of Cloud Platforms in Automotive Domain: An Online Survey", in *2019 IEEE 12th Conference on Service-Oriented Computing and Applications (SOCA)*, Nov 2019, bll 168–175. doi: 10.1109/SOCA.2019.00032.

[4] R. N. Taylor, "Software architecture and design", in *Handbook of Software Engineering*, 2019. doi: 10.1007/978-3-030-00262-6_3.

[5] C. F. J. Lange, M. R. V. Chaudron, en J. Muskens, "In practice: UML software architecture and design description", *IEEE Softw.*, 2006, doi: 10.1109/MS.2006.50.

[6] F. Sagstetter *et al.*, "Security challenges in automotive hardware/software architecture design", 2013. doi: 10.7873/date.2013.102.

[7] P. Y. Reyes-Delgado, M. Mora, H. A. Duran-Limon, L. C. Rodríguez-Martínez, R. V. O'Connor, en R. Mendoza-Gonzalez, "The strengths and weaknesses of software architecture design in the RUP, MSF, MBASE and RUP-SOA methodologies: A conceptual review", *Comput. Stand. Interfaces*, vol 47, bll 24–41, Aug 2016, doi: 10.1016/j.csi.2016.02.005.

[8] M. Jaiswal, "Software Architecture and Software Design", *SSRN Electron. J.*, 2019, doi: 10.2139/ssrn.3772387.

[9] J. Bishung *et al.*, "A critical analysis of topics in software architecture and design", *Adv. Sci. Technol. Eng. Syst.*, 2019, doi: 10.25046/aj040228.

[10] M. L. Roldán, S. Gonnet, en H. Leone, "Knowledge representation of the software architecture design process based on situation calculus", *Expert Syst.*, vol 30, no 1, bll 34–53, Feb 2013, doi: 10.1111/j.1468-0394.2012.00620.x.

[11] R. Morton, "SAT-037 STUDY PROTOCOL FOR THE SYMPTOM MONITORING WITH FEEDBACK TRIAL (SWIFT): A NOVEL REGISTRY-BASED CLUSTER RANDOMISED TRIAL AMONG ADULTS WITH END-STAGE KIDNEY DISEASE MANAGED ON HAEMODIALYSIS", *Kidney Int. Reports*, 2019, doi: 10.1016/j.ekir.2019.05.059.

[12] O. Sievi-Korte, I. Richardson, en S. Beecham, "Software architecture design in global software development: An empirical study", *J. Syst. Softw.*, 2019, doi: 10.1016/j.jss.2019.110400.

[13] O. Sievi-Korte, K. Koskimies, en E. Mäkinen, "Techniques for Genetic Software Architecture Design", *Comput. J.*, vol 58, no 11, bll 3141–3170, Nov 2015, doi: 10.1093/comjnl/bxv049.

[14] A. leonidas Kangkan, "STUDI PENENTUAN LOKASI UNTUK PENGEMBANGAN BUDIDAYA LAUT BERDASARKAN PARAMETER FISIKA, KIMIA DAN BIOLOGI DI TELUK KUPANG, NUSA TENGGARA TIMUR", *J. Chem. Inf. Model.*, 2019.

[15] R. Kazman, L. Bass, en M. Klein, "The essential components of software architecture design and analysis", *J. Syst. Softw.*, vol 79, no 8, bll 1207–1216, Aug 2006, doi: 10.1016/j.jss.2006.05.001.

[16] D. Falessi, G. Cantone, R. Kazman, en P. Kruchten, "Decision-making techniques for software architecture design", *ACM Comput. Surv.*, vol 43, no 4, bll 1–28, Okt 2011, doi: 10.1145/1978802.1978812.

CHAPTER 14

REENGINEERED SOFTWARE ARCHITECTURE BASED ON SCENARIOS

Dr. Govind Singh, Assistant Professor
Department of Computer Science and Engineering, Sanskriti University, Mathura, Uttar Pradesh, India
Email Id- govind@sanskriti.edu.in

ABSTRACT:

The method for reengineering software architectures is presented in this chapter. The strategy specifically covers the software architecture's performance characteristics. The standard function used to evaluate quality aspects includes scenarios. Design modifications are implemented to enhance quality characteristics that do not really meet the criteria. New instances of assessment and design transformations can be carried out until all requirements are fulfilled. We use the reconfiguration of a prototypical measurement technique into a domain-specific programming language as an example to showcase the process.

KEYWORDS:

Cloud Computing, Human Computer, Internet, IoT Devices, IoT Cloud.

INTRODUCTION

Reengineering a software system is typically started by significant changes to the specifications that the system would meet. Instead of addressing functional requirements, these adjustments frequently focus on the attributes of the product. For instance, the software maintenance of the software system may have declined as a result of architecture erosion. The system is restructured in order to enhance this. As far as we know, there aren't many defined architecture reengineering techniques. The functionality that the system is to deliver is frequently the main emphasis of typical classification design techniques. The system's required software quality requirements receive substantially fewer attention from them. Reusability is addressed via object-oriented methods, although typically no evaluation of the completed image is done [1].

Real-time and fault-tolerant research communities, for example, having suggested design approaches that include design phases for supporting their software application. These methods, however, frequently concentrate on a specific software attribute. Our projects with the business world have informed us that a system is never purely real-time, fault-tolerant, or reusable. Systems should instead offer each one of these characteristics, and even more. Software quality standards, as noted in, frequently clash with one another. For example, real-time versus reusability, flexibility versus efficiency, dependability versus flexibility, etc. System design is challenging since it requires carefully balancing the many software qualities. For the purposes of this discussion, we make a distinction between operational software qualities, such as reliability and performance, and development-related software qualities, such as reusability and maintainability [2].

The degree to which a system can meet its software quality requirements is heavily influenced by the software architecture. The limits for the majority of quality criteria have been established once the application architecture has been finalised. Architectural design and reengineering, on the other hand, are the phases of software development that are the least understood and supported by conventional methods. We feel that the contribution of this work is the practical method for reengineering software architectures that we give, together with an example from the real world, to exemplify it[3].

Architecture of Reengineering Method

During architecture design and reengineering, in our experience, software engineers typically manage software quality requirements through a very informal method. After the system has been put into place, it is checked to see if the specifications for software quality have been met. If not, the system's components are changed. Since iterating during system development is typically quite expensive and because the redesign cannot be planned and budgeted, we don't find this technique to be satisfactory. In Figure 1 shown the Object Model of the Beer can Application.

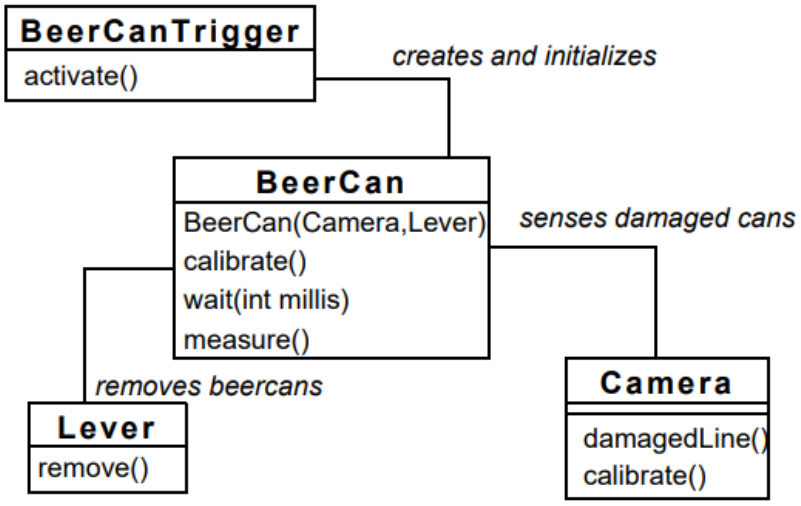

Figure 1: Illustrated that the Object Model of the Beer can Application.

Conventional design approaches have a tendency to place more emphasis on delivering the necessary system functionality than on software quality. This was deemed inadequate by the different software quality-based research groups, who then each suggested their own design approaches for creating real-time, high-performance, and reusable systems. All of these approaches, however, concentrate on a single quality feature and downplay, if at all, the significance of the others. Since the software engineer must balance the many quality criteria for every actual system, we deem these techniques to be unsatisfactory. However, in the absence of a supporting approach, the software engineer builds and reengineers system architectures haphazardly and based only on intuition, with all of the drawbacks that entails. We have created an architectural reengineering process that offers a more practical but objective solution to this problem. We provide a summary of the approach in the next paragraphs of this section and direct readers to for a more thorough review.

Synopsis of the Topics

The modified requirements definition and the current software architecture serve as the input for the architecture reengineering technique. An enhanced architectural design is produced as an outcome. The method's stages are shown visually in Figure 2.

i. Update the architecture to reflect new functional needs. The software attributes are not explicitly addressed at this point, despite the fact that software developers often won't build a system that is less stable or reusable. A preliminary version of the application architectural design is what this produces.

ii. Software quality evaluation each quality attribute (QA) is calculated, generally utilising scenario-based analysis as an evaluation method. The architectural design process is complete if all estimates come in at the specified level of quality or above. If not, the subsequent action is taken.

iii. Modifications to the architecture. At this level, the architecture is improved via QA-optimizing transforms. A whole new iteration of the architectural design is produced by each set of modifications (one or more).

iv. Evaluation of software quality. As soon as the software engineer determines there is no workable solution, the design is once again examined, and the process is repeated from step three until all software quality standards are fulfilled.

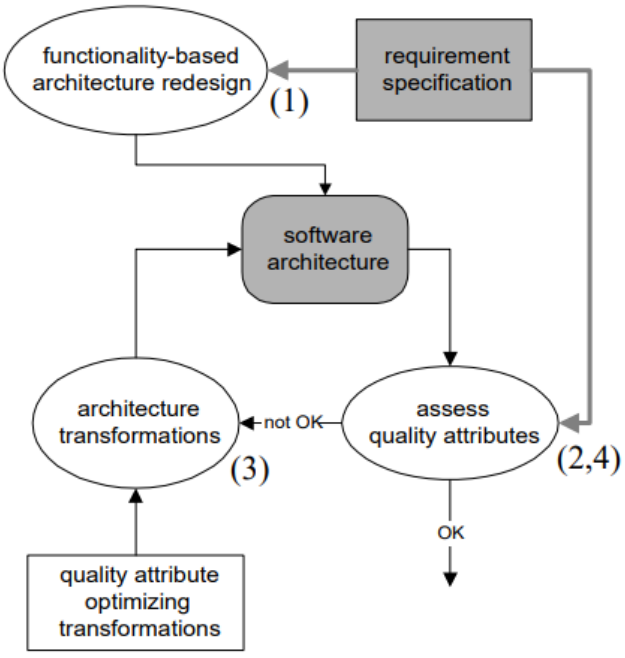

Figure 2: Illustrated that the Outline of the Method.

Through its use in three projects for fire-alarm systems, measurement systems, and dialysis systems the architectural reengineering process has developed. The specific steps of the approach are discussed in greater depth in the sections that follow.

Functionality based Architecture Redesign

A new top-level decomposition of the system's constituent parts is carried out based on the revised requirement definition. Finding and assessing the system's fundamental abstractions is the key challenge at this phase. Despite the fact that these abstractions are represented as objects, according to our experience, these objects are not always present in the application domain. Instead, they are the outcome of a creative process that examines the different domain entities, analyses the most important qualities, and models them as architectural entities. The interconnections between the abstractions are outlined in further depth when the abstractions have been recognised. For instance, the approach used to identify the things that make up the architecture is distinct from conventional object-oriented design techniques. These approaches use a bottom-up approach by first modelling the things that are present in the domain and then organising them into inheritance hierarchies. Starting bottom-up during architectural design and reengineering is not practical in our experience since doing so would need addressing the specifics of the system. Instead, a top-down strategy is preferable.

Assessing Software Quality Requirements

The explicit evaluation of the software attributes of a system or application architecture is one of the key components of the architecture reengineering approach. It is possible to utilise the system itself to evaluate the architecture of the current system. However, there is no actual mechanism to assess after the first transition. Based on the architectural design, it is impossible to evaluate the final system's quality qualities. That would suggest that the architecture is strictly projected in the detailed design and execution. Instead, the objective is to assess the proposed architecture's capacity to meet the standards for software quality. Although this style is very flexible, certain architectural styles, such as layered structures, are less appropriate for systems where performance is a key concern. There are four distinct methods for evaluating quality attributes: scenarios, simulation, mathematical modelling, and experience-based reasoning. The engineer may choose the best method of assessment for each quality feature. Each strategy is detailed in further depth in the sections that follow:

Scenario-based Evaluation

The following procedures are used to evaluate the programme quality using scenarios:

i. Create a sample set of circumstances. The true meaning of the trait is concretized via a series of events. For instance, scenarios that represent common changes in requirements, underlying hardware, etc. may be used to specify the maintainability quality characteristic.

ii. Examine the structure. Each distinct situation establishes an architectural environment. Analysis is used to evaluate how well the architecture performed in

that situation for this quality characteristic. Asking common questions about the qualities of a product might be useful.

iii. Recap the findings. The outcomes of each examination of the architecture and scenario are then compiled into an overall outcome, such as the proportion of scenarios that are approved against those that are rejected.

The unanimity it fosters in the understanding of what a certain software quality really entails is what drives the use of scenarios. Scenarios are a useful tool for combining diverse perspectives on software quality into a unified whole. This viewpoint is more contextually sensitive than the basic definition of software quality and incorporates the particulars of the system to be produced.

In our experience, scenario-based evaluation is very helpful for software attributes that are relevant to development. Change scenarios are a fairly natural way to explain software traits like maintainability. It is also seen in the usage of scenarios for assessing designs. However, the Software Architecture Analysis Method (SAAM) simply examines the architecture in collaboration with stakeholders before thorough design, using just scenarios.

Simulation

A second method for assessing quality characteristics is to simulate the architecture using an implementation of the application architecture. An executable system is created by implementing the major architectural components and simulating the rest. At a reasonable abstraction level, the context in which the system is expected to operate might also be emulated. This implementation may then be used to simulate how an application would behave in different scenarios. Simulation is a helpful addition to the scenario-based method since it can be used to assess operational software features like performance and fault tolerance.

Mathematical Modelling

Numerous research groups, including those focused on high-performance computing, dependability, real-time systems, etc., have created mathematical models, or metrics, to assess software characteristics, particularly those that are operation-related. The mathematical models, in contrast to the other methods, enable static assessment of architectural design models. As both methods are largely appropriate for evaluating operational software characteristics, mathematical modelling is an alternative to simulation.

Experience-based Reasoning

The use of logic and reasoning based on experience is a fourth method for evaluating the quality of software. Experienced software engineers often make insightful observations that may be very useful in avoiding poor design choices and identifying problems that need more analysis. Although most of these encounters are based on anecdotal evidence, several of them may be explained logically.

Architecture Transformation

Once the architecture properties have been assessed, the estimated values are compared to the requirements specification. If one or more of the software qualities are not met, the architecture has to be changed to achieve those. In the architectural reengineering method discussed in this paper, changes to the architecture are performed as architecture transformations. Each transformation leads to a new version of the architecture that has the same functionality, but different values for its quality attributes. Five categories of architecture transformations have been identified. In the sections below, each category is discussed in more detail.

Impose architectural style are presented several architectural styles that improve certain quality attributes for the system the style is imposed upon and impair other software qualities. For example, the layered architectural style, increase the flexibility of the system by defining several levels of abstraction, but generally decrease the performance. With each architectural style, a fitness for each system property is associated. The most appropriate style for a system depends primarily on its software quality requirements. Transforming architecture by imposing an architectural style results in a major reorganisation of the architecture.

Impose Architectural Pattern

A second category of transformations is the use of architectural patterns and these are different from architectural styles in that they are not predominant in the architecture. They are also different from design patterns since they affect the larger part of the architecture. Architectural patterns generally impose a rule on the architecture that specifies how the system will deal with one aspect of its functionality, e.g., concurrency or persistence.

Apply Design Pattern

A less dramatic transformation is the application of a design pattern on a part of the architecture. For instance, an Abstract Factory pattern might be introduced to abstract the instantiation process for its clients. The Abstract Factory increases maintainability, flexibility and extensibility of the system since it encapsulates the actual component types(s) that are instantiated. Nevertheless, it decreases the efficiency of creating new instances due to the additional computation, thereby reducing performance and predictability. Different from imposing an architectural style or pattern, causing the complete architecture to be reorganised, the application of a design pattern generally affects only a limited number of components in the architecture. In addition, a component can be involved in multiple design patterns without creating inconsistencies

Convert Quality Requirements to Functionality

Another type of transformation is the conversion of a software quality requirement into a functional solution. This solution consequently extends the architecture with functionality not related to the problem domain but is used to fulfil a software quality requirement. Exception handling is a well-known example that adds functionality to a component to increase the fault tolerance of the component.

Software Quality Requirements

A number of conditions should be met by the DSSA for measuring systems. In the context of this work, reusability and maintainability are the criteria for software quality that are most relevant. Real-time and robustness criteria for measurement systems must also be met, although we do not explore them in this study. Scenarios are often the most effective way to evaluate development-related software attributes. A collection of scenarios are created for each software quality in the assessment process, which is followed by human execution of the scenarios for the architecture and result interpretation. The evaluation might be carried out completely or statistically. The first method defines a number of scenarios that, when taken together, cover the specific examples of software quality. For instance, a scenario for reuse represents all appropriate methods to reuse the architecture or specific components of it. The design can be utilised to its full potential if all situations go well. The second strategy is creating a selection of situations that serve as a representative sample without include every event that could arise. A measure of how well the architecture satisfies the standards for software quality is the ratio between situations that the architecture can manage and scenarios that it cannot handle well. Obviously, each strategy has its drawbacks. The fact that it is often hard to specify a whole range of situations is a drawback of the first strategy. The second approach's weak issue is the defining of a representative collection of situations since it is not apparent how to determine if a group of scenarios is representative. Despite these drawbacks, scenarios are a valuable method for assessing software development-related characteristics. As was said regarding reusability, there is no established method for evaluating the quality characteristic. Nevertheless, scenario-based assessment relies on the impartiality and imagination of the software engineers who design and implement them. This hasn't proven to be a significant issue for us in our initiatives.

LITERATURE REVIEW

R. Rodríguez-Echeverría et al. stated that the last years one of the main concerns of the software industry has been to reengineer their legacy Web Applications (WAs) to take advantage of the benefits introduced by Rich Internet Applications (RIAs), such as enhanced user interaction and network bandwidth optimization. However, those reengineering processes have been traditionally performed in an ad-hoc manner, resulting in very expensive and error-prone projects. This situation is partly motivated by the fact that most of the legacy WAs were developed before Model-Driven Development (MDD) approaches became mainstream. Then maintenance activities of those legacy was have not been yet incorporated to a MDA development lifecycle. OMG Architecture Driven Modernization (ADM) advocates for applying MDD principles to formalize and standardize those reengineering processes with modernization purposes. In this paper we outline an ADM-based WA-to-RIA modernization process, highlighting the special characteristics of this modernization scenario [4].

M. Baldassarre et al. stated that the Applications security threats are always changing as a result of variables including attacker advancement, the introduction of new technologies, and the utilization of ever-more sophisticated systems. Implementing design and programming techniques that ensure the security of the code on the one hand, and the privacy of the data on the other, are both required in this situation. This article suggests a software development methodology called Privacy Oriented Software Development (POSD), which enhances conventional development procedures by including the tasks required to maintain security

and privacy in software systems. The strategy is founded on five essential components: Privacy by Design, Privacy Design Strategies, Privacy Pattern, Vulnerabilities, and Context. It may be used both forward and backward while creating new systems or re-engineering an old one. The POSD technique in the backward mode is presented in this study along with an experiment within the framework of an industrial project. Results demonstrate that POSD is capable of identifying software vulnerabilities, locating the source code remediation patterns required to fix them, and designing the target architecture required to direct privacy-oriented system reengineering [5].

C. Jefferies et al. illustrated that the Organizations now have more reasons than ever to think about reengineering their current systems to allow access from heterogeneous devices like desktops and mobile devices. Using a service-oriented architecture is one method to do this (SOA). To increase the efficiency of a SOA business process, native language calls may be utilised in place of protocol-based messages, however doing so may reduce the system's adaptability. This article describes a research that compares native language calls to protocol-based messaging in a system reengineering scenario to examine the agility/performance trade-off of employing native language calls in a business process [6].

P. Dugerdil discussed that proposes a reverse-engineering method for understanding and maintaining legacy information systems that borrows heavily from the Unified Process (UP). We demonstrate that the recovery of the system's architecture depends on the reconstruction of the use-case model. First, by using use cases, we may retrace the model of the business process that the system underlies. These use-cases are then examined in order to recreate the system analysis model that represents the high level architecture. The latter will guide the search for the relevant software components in the system called the "hypothetical architecture." Third, in order to identify the software components that carry out the business operations, scenarios based on the use-case model are utilized as their source. The components discovered by performing the scenarios may then be compared to the "hypothetical architecture." The business operations of the business model and the software components of the system may therefore be matched. The maintenance engineer can better comprehend these pieces' functions thanks to this [7].

M. Riebisch et al. illustrated that the given the magnitude and need for frequent, fast changes in modern software systems, architectural quality is a crucial component. Architectural choices for business systems that are essential to success carry a significant risk to the market share and perhaps the very life of the organization. Both the design and refactoring procedures depend on these choices. Making choices is a tough and dangerous process due to the complexity of the options, such as ambiguous, conflicting aims, unknown consequences, and harmful settings. A methodical approach to decision-making helps reduce risks. We presented decision theory techniques in a previous study to help people make such judgments rationally. In both architectural design and refactoring, the strategy for analyzing architectural choice alternatives is introduced in this study. This technique incorporates components of ALMA, a scenario-based assessment technique. The use of the enhanced decision-making process is shown through a real-world case [8].

R. Buhr discussed that the high-level interpretation, design, development, and reengineering of all sorts of systems, the capacity to assign behavior to architecture is crucial from object-oriented programmer to parallel and distributed computer systems. Scenarios are an excellent method to do this, however common scenario approaches that start with intercomponent

wiring, such message sequence charts, do not scale up effectively. A novel, scenario-based method of assigning behavior to infrastructure that addresses the scale-up issue is provided via use case mappings. The notation makes it possible to depict behavior patterns of whole systems in terms of causal routes without having to make use of wiring by allowing compact, composite maps to be created. The paper's goal is to persuade software and system engineers in spite of and precisely because of its simplicity and lack of detail that the method has depth and offers value [9].

P. Marinoet al. illustrated that the authors of this research are now working on reengineering a half terabyte data warehouse for a telecom firm (DW). The data warehouse context is first described, with a particular emphasis on three interconnected and important issues: data source integration, component reuse, and performance limitations. The issues with the runtime system's true implicit architecture are then discussed and pointed out. The suggested architecture is then given. It features two views: one that is static and built using filters and pipes, and the other that is dynamic and built with an event-driven scheduler rather than a fixed-time one. In the runtime scenario, the scheduler pulls the pipe configurations from the metadata repository, reads them, and then runs the intended filters in real time to turn the data into business intelligence [10].

S. Davu et al. illustrated that the assuming infrastructure is available, mobile IP allows disconnection-free handoff. In advance, intermediary software agents must be set up on the Internet. Although this infrastructure-based mobility strategy provides access to mobile hosts, implementation expenses and considerable handoff and tunneling delays are also involved. In this study, we examine an alternative mobility strategy that employs solely end-point technology, has a substantially quicker loss-free handoff, and does not need any such infrastructure. The Interactive Protocol for Mobile Networks (IPMN) End-to-End protocol intelligently handles handoff depending on data from the MAC Layer. Renewal of the current connections is done in response to the network address change by modifying the TCP/IP stack at the end points. To further make end-point reengineering simpler, it employs a unique inter-protocol communication architecture. However, the TCP/IP protocol software does not need to undergo any functional changes. The IPMN delivers blazingly fast event-based handoff and far quicker and simpler transmission than MIP, apart from the differences in deployment conditions. We compare the two in terms of performance using a detailed model [11].

E. Folmer et al. illustrated that over the years the software engineering community has increasingly realized the important role software architecture plays in fulfilling the quality requirements of a system. The quality attributes of a software system are, to a large extent determined by the systems software architecture .Usability is an essential part of software quality. The usability of software has traditionally been evaluated on completed systems. Evaluating usability at completion introduces a great risk of wasting effort on software products that are not usable. A scenario based assessment approach has proven to be successful for assessing quality attributes such as modifiability and maintainability. It is our conjecture that scenario based assessment can also be applied for usability assessment. This paper presents and describes a scenario based assessment method to evaluate whether a given software architecture meets the usability requirements. The Scenario based Architecture Level Usability Assessment method consists of five main steps, goal selection, usage profile creation, software architecture description, scenario evaluation and interpretation [12].

N. Niu et al. illustrated that the industrial information engineers may choose from a variety of software architectural alternatives when creating their corporate information systems. Even while those recommendations and the associated methodology aid engineers in choosing the best design, there are few systematic techniques for assessing software architecture. A scenario-based approach has been suggested to evaluate how software architecture influences the fulfilment of business objectives in order to pick optimal software architecture from a variety of possibilities. The created technique delivers exceptional insights into software development and may be included into the industrial informatics practice of an organization at a reasonable cost, according to the empirical assessment on the selection of a supply chain software tool [13].

M. A. Babar et al. illustrated that the creation of scenarios for stakeholder meetings to characterize a desired set of quality characteristics is one of the most crucial concerns in scenario-based software architecture assessment. Studying the efficiency of such gatherings is a crucial research subject since they are expensive to organize. In this work, we provide the results of an empirical research on software architecture assessment that looked at the number of scenarios obtained and lost throughout the review process to determine the efficacy of scenario creation sessions. The results of the experiment data analysis raise the issue of whether meetings for building scenarios are helpful since more current scenarios were lost as a consequence of these meetings than new scenarios were obtained [14].

DISCUSSION

It is desirable to analyse a proposed software system to see how well it satisfies required quality standards. Lack of a shared knowledge of high level design and a lack of a basic grasp of many of the quality features are some of the factors making such analysis challenging. Some of the problems involved in the high level design of software systems are being explained as a result of the current increase in interest in software architecture. In this article, we'll demonstrate how to use software architectural ideas to analyse complicated software systems for desirable qualities. By employing scenarios to record crucial behaviours affecting the system under examination, we make up for the absence of a basic grasp of how to articulate these features. We will discuss our experiences with scenario-based examination of software system architectural descriptions. Scenarios are concise descriptions of how a system will likely be used, from both the developer and end-user perspectives. The Software Architecture Analysis Method is a systematic approach that analyses architectures using scenarios (SAAM).

CONCLUSION

The technique for reengineering software architectures that was provided in this work offers a useful tool for assessing and redesigning architectures. The approach employs four methods scenarios, simulation, mathematical modelling, and experience-based reasoning for evaluating design. There are five different architecture transformations that may be used to enhance the architecture: architectural style imposition, architectural pattern application, design pattern usage, functionality conversion of quality needs, and distribution of quality requirements. We used a real-world system from the field of measurement systems a system for inspecting beer cans to demonstrate the technique. This system was redesigned into a measurement system-specific software architecture. The criteria for the DSSA's reusability

and maintainability were the major emphasis of this work. For evaluating every need, scenarios were established. Despite the lack of a widely accepted method for evaluating the quality features of systems, this research demonstrates that scenarios provide practitioners a potent tool. Since each transformation has an issue that is clearly addressed, other transformations are looked into, and the justification for the design choices is recorded, architecture transformations provide an objective means to redesign. The reengineering approach will need to be expanded to accommodate additional quality standards, and more industry case studies and projects will use it.

REFERENCES

[1] I. Zyrianoff *et al.*, "Architecting and Deploying IoT Smart Applications: A Performance–Oriented Approach", *Sensors*, vol 20, no 1, bl 84, Des 2019, doi: 10.3390/s20010084.

[2] S. M. Sharafi, "SHADD: A scenario-based approach to software architectural defects detection", *Adv. Eng. Softw.*, vol 45, no 1, bll 341–348, Mrt 2012, doi: 10.1016/j.advengsoft.2011.10.012.

[3] A. Alreshidi, A. Ahmad, A. B. Altamimi, K. Sultan, en R. Mehmood, "Software architecture for mobile cloud computing systems", *Futur. Internet*, 2019, doi: 10.3390/fi11110238.

[4] R. Rodríguez-Echeverría, J. M. Conejero, P. J. Clemente, J. C. Preciado, en F. Sánchez-Figueroa, "Modernization of Legacy Web Applications into Rich Internet Applications", in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2012, bll 236–250. doi: 10.1007/978-3-642-27997-3_24.

[5] M. T. Baldassarre, V. S. Barletta, D. Caivano, en M. Scalera, "Privacy Oriented Software Development", in *Communications in Computer and Information Science*, 2019, bll 18–32. doi: 10.1007/978-3-030-29238-6_2.

[6] C. Jefferies, P. Brereton, en M. Turner, "A comparison of binding technologies for multi-channel access", *Front. Artif. Intell. Appl.*, 2009, doi: 10.3233/978-1-60750-052-0-149.

[7] P. Dugerdil, "Reengineering Process Based on the Unified Process", in *2006 22nd IEEE International Conference on Software Maintenance*, Sep 2006, bll 330–333. doi: 10.1109/ICSM.2006.50.

[8] M. Riebisch en S. Wohlfarth, "Introducing Impact Analysis for Architectural Decisions", in *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)*, Mrt 2007, bll 381–392. doi: 10.1109/ECBS.2007.46.

[9] R. J. A. Buhr, "Use case maps for attributing behaviour to system architecture", in *Proceedings of the 4th International Workshop on Parallel and Distributed Real-Time Systems*, 1996, bll 3–10. doi: 10.1109/WPDRTS.1996.557425.

[10] P. Marino, C. Siguenza, J. Nogueira, F. Poza, en M. Dominguez, "An event driven software architecture for enterprise-wide data source integration", in *Proceedings International Conference on Information Technology: Coding and Computing (Cat. No.PR00540)*, 2000, bll 140–145. doi: 10.1109/ITCC.2000.844197.

[11] S. Davu, R. Y. Zaghal, en J. I. Khan, "End-to-End High Performance Mobility without Infrastructure", in *2005 IEEE International Conference on Electro Information Technology*, 2005, bll 1–8. doi: 10.1109/EIT.2005.1627012.

[12] E. Folmer, J. Van Gurp, en J. Bosch, "Scenario-based Assessment of Software Architecture Usability", *Assessment*, 2003.

[13] N. Niu, L. Da Xu, en Z. Bi, "Enterprise Information Systems Architecture—Analysis and Evaluation", *IEEE Trans. Ind. Informatics*, vol 9, no 4, bll 2147–2154, Nov 2013, doi: 10.1109/TII.2013.2238948.

[14] M. Ali Babar, H. Shen, S. Biffl, en D. Winkler, "An Empirical Study of the Effectiveness of Software Architecture Evaluation Meetings", *IEEE Access*, vol 7, bll 79069–79084, 2019, doi: 10.1109/ACCESS.2019.2922265.

CHAPTER 15

COMPONENT BASED DESIGN FOR SOFTWARE DEVELOPMENT

Ms. Rachana Yadav, Assistant Professor, Department of Computer Science, Jaipur National University, Jaipur, India, Email Id- Rachana.yadav@jnujaipur.ac.in

ABSTRACT:

Two of the most prevalent engineering paradigms in the modern software community and industry are component-based software engineering (CBSE) and service-oriented software engineering (SOSE). Both paradigms have resemblance in many ways, despite the fact that they both pursued separate growth paths and have divergent foci. This has made it more difficult to grasp and apply ideas that are similar to one another or are the same concepts but have been given different names. In this chapter it talk about just the idea of combining the two paradigms' advantages to satisfy non-functional goals. In order for practitioners and academics to become aware of the key challenges of both paradigms and to provide them with inputs for further using them in a satisfactory and complementary manner, this paper will contribute by elucidating the characteristics of CBSE and SOSE, reducing the gap among both them, and uniting the two worlds.

KEYWORDS:

Computer Science, Information Technology, Software Engineering, Software Design.

INTRODUCTION

After the creation of the data, architectural, and interface designs, component level design, also known as procedural design, takes place. The goal is to create functional software using the design model. Yet, the operational programme has a low degree of abstraction whereas the current design model has a pretty high level. The translation process may be complex, leaving room for the insertion of minor flaws that are tough to detect and fix in the later phases of the software development process. These comments were spoken a long time ago, yet they still hold true today. We must adhere to a set of design rules while converting the design model into source code so that we can accomplish the translation while also avoiding "introducing defects to start with [1].

Programming languages may be used to express the component-level design. In essence, the design model serves as a guide for creating the programme. An alternate strategy is to express the procedural design using an intermediary that can be quickly converted into source code, such as a graphical, tabular, or text-based representation. The data structures, interfaces, and algorithms created should adhere to a number of well-known procedural design principles regardless of the method used to express the component level design. These principles assist us to prevent mistakes as the procedural design develops. We analyse these design principles in this chapter.

Structured Programming

Early in the 1960s, the foundations of component-level design were established, and Edsgar Dijkstra and his colleagues' efforts helped to firmly establish them. The idea of creating any programme from a set of limited logical structures was put out by Dijkstra and others in the late 1960s. "Maintenance of functional domain" was stressed by the constructions. As a result, a reader could more readily follow the procedural flow since each construct had a known logical structure and was entered at the top and departed at the bottom. Sequence, condition, and repetition are the structures. The processing steps that are necessary for any algorithm's definition are implemented via sequence. Repetition enables looping, while condition offers the option for selective processing depending on some logical occurrence. Structured programming, an essential component-level design method, is based on these three notions [2].

The purpose of the structured constructs was to restrict the software's procedural architecture to a select few predictable processes. According to complexity metrics, the employment of structured structures improves readability, testability, and maintainability by lowering programme complexity. A human comprehension process known as chunking is also facilitated by the usage of a restricted set of logical constructions. Think about how you are reading this page in order to comprehend this procedure. Instead, then reading individual letters, you identify patterns or groups of letters that come together to create words or sentences. Instead of reading the design or code line by line, the structured constructs are logical chunks that enable a reader to identify the procedural aspects of a module. When instantly recognised logical patterns are discovered, understanding is improved.

Graphical Design Notation

An image is worth a thousand words, but it's crucial to understand which picture and which thousand words are being used. There is no denying that visual representations of processes, such as flowcharts and box diagrams, are valuable for conveying procedural information. The incorrect image, however, can result in the incorrect programme if graphical tools are utilised improperly. A line (arrow) of control connecting two processing boxes symbolises the sequence. Condition, also known as if then-else, is represented as a decision diamond that, if true, triggers processing in the then-part, and, if false, triggers processing in the else-part. Two fundamentally different forms are used to illustrate repetition. As long as a condition is true, a loop job is repeatedly carried out by the do while statement. A repeat till first performs the loop task, checks a condition, and then repeats the task until the test is unsuccessful. The figure's selection (or select-case) construct is essentially an expansion of the if-then-else statement. Up to a true condition is fulfilled, a parameter is checked via a series of choices, and then a case part processing route is carried out [3].In general, when an escape from a collection of nested loops or nested conditions is needed, the dogmatic usage of solely the structured structures might result in inefficiencies. Most importantly, further complicating every logical test along the escape route might impair software control flow, raise the chance of mistake, and degrade readability and maintainability. Two possibilities are left to the designer. The first is a redesign of the procedural representation to eliminate the need for an escape branch at a nested point in the control flow, and the second is a controlled violation of the structured constructs via the design of a confined branch out of the nested flow. While the first option is undoubtedly the best one, the second may be used without going against the spirit of organised programming.

Functionality based Architectural Design

Architecture design is the process of converting a set of requirements into a software architecture that fulfils, or at least facilitates the fulfilment of, the requirements. The method for architecture design presented in this part of the book has a focus on the explicit evaluation of and design for quality requirements, but that does not mean that the functional requirements are irrelevant.

Before one can start to optimize an architecture for quality requirements, the first step must be to design a first version of the architecture based on the functional requirements. The first design phase in the method consists of three main steps. The first step concerned with determining the context of the system under design, the interfaces of the system to the external entities it interacts with and the behaviour the system should exhibit at the interfaces.

The second step is the identification of the archetypes that is the main architectural abstractions, and the relations between the archetypes. Our experience is that finding these archetypes is very important especially for the later phases. Architecture transformations tend to build additional structures around the archetypes for fulfilling quality requirements. The final step in functionality based architectural design is the description of system instances using the archetypes and the system interfaces. Since the architecture, for instance in the case of product-line architectures, may be required to support a number of different instantiations, these have to be specified explicitly to verify that the system, in addition to the commonality also supports the required variability [4].

The assumption underlying our approach to architectural design is that starting from the functional requirements does not preclude the optimization of quality requirements during the later architecture design stages. We agree that no pure separation can be achieved, i.e. an architectural design based on functional requirements only will still have values for its quality attributes.

However, our position is that an objective and repeatable architectural design method must be organized according to our principles since it is unlikely that an architectural design process does not require iterations to optimize the architecture. Since an architecture based primarily on functional requirements is more general and can be reused as input for systems in the same domain but with different quality requirements. On the other hand, it is unlikely that a software architecture that fulfils a particular set of quality requirements will be applicable in a domain with different functional requirements.

Defining the System Context

All software has to interface with one or more external entities. Different from what one may suspect; it is the externally visible behaviour of a system that is the thing that counts. All our efforts as software architects and engineers are judged from this perspective, although it is difficult to maintain this viewpoint since virtually all of our efforts are spending on the internals of software systems. In Figure 1 shown the UML Sequence Diagram.

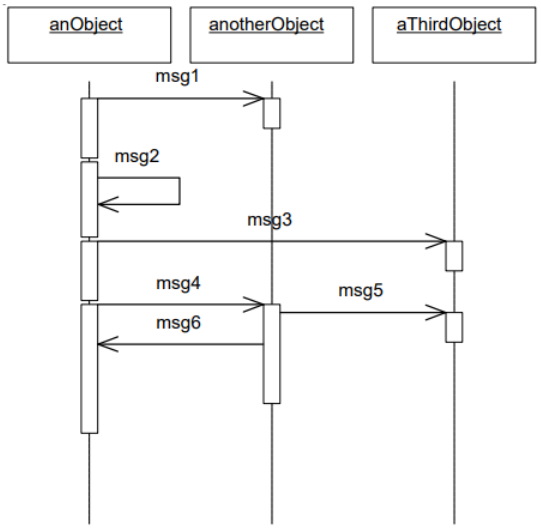

Figure 1: Illustrated that the UML Sequence Diagram[5]**.**

The entity at the other end of an interface may be located at a lower-level, a higher level or at the same-level as the system that we are designing. Examples of lower-level entities include network interface and sensor or actuator interfaces, whereas same-level entities often are systems that address a different functional domain, but need to communicate because of system integration requirements. For instance, in the case of Securitas Larm, high-end fire-alarm system has to interface with other building automation systems to achieve more intelligent behaviour. For instance, if no humans are supposed to be in one part of the building, the particle-density sensors should be more sensitive then when smoking persons might be walking around in that part. In the latter case, sensors should not activate the alarm when temporary peeks in particle density are detected. Higher-level entities may be system integration software, e.g., in the case of the fire-alarm system, the building automation integration system, or human beings, e.g., operators of the system or other users.

Explicitly defining the system in terms of the functionality and quality required on its interfaces is an important starting point once the requirements by the customer have been defined. It allows one to distribute requirements to the interfaces and to define the various quality requirements more precisely. Interface-specific requirements allow for the specification of both operational and development quality requirements. For example, performance, real-time and reliability requirements can be expressed on the services provided on the interface. In addition, maintainability and flexibility requirements can be expressed in terms of the likely changes at the interface. An additional reason for explicitly defining the system context and boundaries is because our experience is that there is a natural tendency to include more and more aspects during design.

In the situations that explicit effort is spent on software architecture design, there generally also is an understanding that this process should be allowed to take time. Because there is no extreme time pressure, software architects, in our experience, try to extend the domain of the design because each of these extensions will improve the applicability of the architecture and allow for likely future requirements to be integrated more easily. The problem, however, is that these extensions increase the design and development cost, resulting in the situation where the current development project budget is partially used for maintenance activities in

response to likely, but not certain future requirements. Management may easily react to this development using the well-known tool, i.e., strict deadlines, not allowing for a sufficiently thought-through architecture. The more fruitful approach to this is to explicitly define the system boundaries and the functional and quality requirements, requiring software engineers to stick to their appointed domain. In the case of a software architecture design for a product-line, the definition of the system context is somewhat more complicated since products in the product-line tend to have variations in the interfaces they provide.

The product-line architecture has to support the superset over all products of the interfaces, functional requirements and quality requirements without sacrificing cost and resource efficiency for the low-end product. The latter may prove difficult to achieve in practice, but, in our experience, the reasons for these difficulties are often located in the reusable assets rather than in the architecture. Since the architecture is primarily facilitating the fulfilment of functional and quality requirements, it often quite possible to exclude or short-cut parts of the architecture for low-end products. In later chapters, we address the implementation of variability in reusable assets, including the issues related to excluding functionality for low-end instantiations of reusable components.

Concluding, the following activities are part of the system context definition step:

A. Define the interfaces of the system to external entities. These entities may be located at a lower level, a higher level or at the same level.

B. Associate functional and quality requirements with each interface. Both operational and development quality requirements can be associated with interfaces.

C. In the case of a product-line architecture, the variability in the interfaces supported by the various products in the product-line should be explicitly identified and specified. The cost and resource efficiency of low-end products should not be sacrificed for the requirements of high-end products.

Identifying the Archetypes

Once the boundaries for the system have been defined, the next step is to identify and define the core abstractions based on which the system is structured. We refer to these core abstractions as archetypes. It is of critical importance to successful architecture design that the architect finds a small set of, often highly abstract, entities that, when combined, are able to describe the major part of the system behaviour at an abstract level.

These entities form the most stable part of the system and supposed not to be changed or only in very limited ways. Our experience is that even relatively large systems can be described in terms of a small number of archetypes. It is important to note that the archetypes are radically different from subsystems. Whereas subsystems decompose system functionality into a number of big chunks, the archetypes represent stable units of abstract functionality that appear over and over again in the system. In the examples later in this chapter, the difference will be exemplified more clearly.

The process of identifying the entities that make up the architecture is different from, for instance, traditional object-oriented design methods as proposed by, among others. Those methods start by modelling the entities present in the domain and organize these in inheritance hierarchies, i.e. a bottom-up approach. Our experience is that during architectural design it is not feasible to start bottom-up since that would require dealing with the details of the system. Instead one needs to work top-down. Architecture entity identification is related to domain analysis methods, but some relevant differences exist. First, although archetypes are modelled as domain objects, our experience is that these objects are not found immediately in the application domain. Instead, they are the result of a creative design process that, after analysing the various domain entities, abstracts the most relevant properties and models them as architecture entities. Once the abstractions are identified, the interactions between them are defined in more detail.

A second difference between architecture design and domain analysis is that the architecture of a system generally covers multiple domains. Once one has reduced the archetype candidates to a small and manageable set that has proven some stability, the relations between the archetypes are identified and defined. The types of relations are generally domain specific and describe control and/or data flow in the system. The relations should not be generic relation types as, for instance, the generalization and aggregation relations in object-oriented modeling. The presence of these relations, especially generalization, between archetypes is suspicious and one should reconsider whether the involved archetypes should perhaps be merged. Within the domain of architecture description languages, architectures are described in terms of components and connectors. The connectors are explicitly modelled entities representing the relations between components.

When using particular architectural styles, the connectors are style specific as well. For instance, in the pipes and filters architectural style, the pipes are represented as connectors. However, connectors are not equivalent to relations between archetypes. Instead, connectors represent one way of implementing relations. When the components implementing two related archetypes provide a good match, the relation may not be represented explicitly but rather through normal message passing. However, if some mismatch between the components exists, then the necessary glue code can be implemented in a connector that then becomes a first-class entity in the implementation. Small groups of related archetypes tend to form system-specific ‘architectural patterns’ that are applicable in many locations in the system instantiations. The ‘architectural patterns’ may prove to have a wider validity than just the system context and may actually suit more systems in an application domain.

Describing System Instantiations

The identified collection of archetypes captures the most stable and core abstractions of the system domain. However, these abstractions do not provide a system description. To describe the system under design, an instantiation of the archetypes and the relations between them is required. Since the archetypes capture core abstractions in the system domain, they are generally instantiated in many places in the system in many different concrete forms. Since a system instantiation is concerned with the structure of concrete instances of the final product, a decomposition of the system into subsystems is generally relevant. A recursive decomposition of the system into a hierarchy of subsystems helps to deal with the complexity of software systems. The complexity of a software system does not have to be the result of sheer size, it can also result from a multitude of interfaces to the system or because of highly

prioritized, but strongly conflicting quality requirements. For instance, the new generation of certain types of embedded systems, e.g., handheld devices, have extreme flexibility and performance requirements. The cost-effective implementation of these conflicting quality requirements is a major challenge for the software engineers involved. The recursive decomposition of the system into subsystems is populated with instantiated archetypes.

At the leaf levels of the system, the subsystems are assigned individual instances of archetypes. The generic relations between the archetypes identified during the previous step are instantiated together with the instantiation of the involved archetypes. The assignment of archetypes to subsystems and specification of relations between subsystems allows for a verification of the match between the domain abstractions represented by the archetypes and the concrete system instantiation. If mismatches are identified, this is generally the indication of a problem that needs to be investigated to make sure that no fundamental mistakes have been made that often prove extremely costly to repair at a later stage. It is the description of the system instantiation that we consider to be the architecture of a software system, i.e. its decomposition into its main components. However, this decomposition does not need to be single level, but may incorporate a two or more levels of decomposition for critical parts of the system.

As we will discuss in later chapters, the goal of software architecture design is to specify, early in the development process, a system structure that allows for the fulfilment of the system requirements. In certain cases, it is required to perform more detailed analysis of critical parts of the system in order to be able to state with sufficient certainty that the system will fulfil its quality requirements. Product-line architectures need to support multiple system instantiations, since the individual products in the family have unique requirements. During the definition system instantiations, explicit attention has to be directed to the variation in system instantiation for each product. Although the difficulties of providing the required variability are primarily found in the implementation of the reusable assets, uncareful design of the software architecture can lead to unwanted rigidness in dimensions where flexibility is required.

Summarizing, the following activities take place during the definition of system instantiations:

i. The system is recursively decomposed into subsystems.

ii. Each subsystem is either populated with instantiated archetypes that fulfil the functionality required from the system or is represented by an individual instantiated archetype.

iii. The generic relations between the archetypes are instantiated for the instantiated archetypes and a verification of the match between abstractions and the concrete system decomposition is performed.

iv. Sufficient variability of product-line architectures is verified by the definition of multiple system instantiations, representing different products.

LITERATURE REVIEW

S. Gahlot et al. illustrated that the designs a single component-based metric to measure the complexity of any software in any phase of software development life cycle. The metric is designed on the basis of existing coupling and cohesion metrics like normalized hamming code (NHD), lack of cohesion in methods (LCOM), conceptual coupling (CoCC), structural and semantic coupling metric (SSCM). The designed metric also covers the coupling between parent and its inherited class, static import, anonymous class contribution and the coupling between inner and outer class to analyze the complexity of software precisely. The analysis of the metric has been done on seven industrial and academic projects against existing state of art coupling and cohesion metric i.e., NHD, COCC, SSCM, LCOM5 and method attribute cohesion metric. The result and analysis shows the significance of the designed metric [6].

I. Polato et al. illustrated that the Heterogeneous models have been alternative solution to problems involving more than one application domain. Over the last decade, models of computation have been investigated, offering new possibilities of use in several applications. This paper presents the Extended Dataflow model, an extension of the original dataflow model, with support to event handling. In Extended Dataflow a model specification using XML has been developed and it is presented along with its model syntax. We discuss its operational semantics which aims at avoiding ambiguities and inconsistency. In addition, the reuse of third party components within the model is illustrated through the case study of a digital filtering system [7].

P. Arato et al. stated that the In order to cope with the increasing complexity of system design, component-based software engineering advocates the reuse and adaptation of existing software components. However, many applications particularly embedded systems consist of not only software, but also hardware components. Thus, component-based design should be extended to systems with both hardware and software components. Such an extension is not without challenges though. The extended methodology has to consider hard constraints on performance as well as different cost factors. Also, the dissimilarities between hardware and software have to be resolved. In this paper, the authors propose such an extended component-based design methodology to include hardware components as well. This methodology allows the designer to work at a very high level of abstraction, where the focus is on functionality only. Non-functional constraints are specified in a declarative manner, and the mapping of components to hardware or software is determined automatically based on those constraints in the so-called hardware/software partitioning step [8].

D. Alonso et al. stated that the Real-Time Systems have some characteristics that make them particularly sensitive to architectural decisions. The use of Frameworks and Components has proven effective in improving productivity and software quality, especially when combined with Software Product Line approaches. However, the results in terms of software reuse and standardization make the lack of portability of both the design and component-based implementations clear. This separation is supported by the automatic integration of the code obtained from the input models into object-oriented frameworks. The article also details the architectural decisions taken in the implementation of one of such frameworks, which is used as a case study to illustrate the proposed approach. Finally, a comparison with other alternative approaches is made in terms of development cost [9].

H. Wei et al. stated that the Componentization is an important method to improve the reusability of robot software and reduce the difficulty of system design. In this paper, we propose a component-based design framework for robot software architecture. First, the robot system is functionally decomposed into reusable components. On this basis, the static model and run-time model of component are established, and a component interface definition language based on the model is designed. Second, a lightweight middleware is proposed according to the communication mode between robot components, and a component development tool and a visual component assembly environment based on the middleware are designed to facilitate the developers. Finally, an application based on the framework is introduced to verify the validation of the design framework [10].

J. Dong et al. stated that the Software patterns are a new design paradigm used to solve problems that arise when developing software within a particular context. Patterns capture the static and dynamic structure and collaboration among the components in a software design. A key promise of the pattern-based approach is that it may greatly simplify the construction of software systems out of building blocks and thus reuse experience and reduce cost. However, it also introduces significant problems in ensuring the integrity and reliability of these composed systems because of their complex software topologies, interactions and transactions. There is a need to capture these features as a contract through a formal model that allows us to analyze pattern-based designs. In this paper, we provide an overview of a formal framework for ensuring the integrity of the compositions in object-oriented designs by providing mathematically rigorous modeling and analysis techniques for object-oriented systems comprising pattern-based designs as the basic building blocks or design components [11].

H. Jhingai et al. ststed that this chapter presents a quick way to develop your system with custom components in Delphi. Use the above method can minimize code duplication, improve work efficiency. In this paper, we apply this method to the development of information management system in a university based on the Data Snap three-tier architecture. Through practice, we find that the workload of the original three months, and we only need 20 days to complete. Therefore, adopting the rapid software development method based on component design proposed in this paper can not only shorten the programming time of management systems such as ERP and information management by more than two thirds, but also bring great advantages in system development [12].

J. Sharp and S. Ryan discussed that Component-Based Software Development (CBSD) is considered by many as the next revolution in systems development. Its focus is on the integration of pre-fabricated software components to build systems that increase portability and flexibility. CBSD purports to address the problem of systems which are delivered behind schedule, over-budget, and inadequately meeting user requirements. A major contribution of this work is providing a solid theoretical foundation using Simon's problem-solving model and the tenets of design science to the emerging CBSD paradigm which has lacked a theoretical-base. In doing so, we construct a theoretical framework of the CBSD development phases by synthesizing the current CBSD related literature and developing propositions for guiding future research. This framework clearly differentiates between component and systems development in the CBSD approach. Implications for both research and practice are also discussed [13].

P. Rana and R. Singh discussed that the Component based software engineering (CBSE) is based on the concept of reusability. CBSE is upcoming paradigm where emphasis is laid on reuse of existing component and rebuilds a new component. Software metrics are used to check the complexity of software. Many software metrics have been proposed for CBS to measure various attributes like complexity, cohesion, coupling etc. Many different cohesion and coupling metrics have been developed. For quality software the cohesion should be high and coupling should be low. The aim of this paper is to develop adequate coupling, cohesion and interface metrics. Graph notation and concept of weights have been used to illustrate proposed metrics and evaluate the results accordingly [14].

P. Panchal stated that the Software Reliability is the likelihood of disappointment free software activity for a predefined timeframe in a predetermined climate. Software Reliability is additionally a significant factor influencing framework dependability, Software Reliability without inordinate constraints. Different methodologies can be utilized to improve the dependability of part-based software framework; be that as it may, it is difficult to adjust advancement time and spending plan with software reliability. We register the dependability of the different component-based software framework utilizing fuzzy logic approach. We can plan and figure the reliability of the component-based software frameworks by utilizing neural organization, neuro-fuzzy and hereditary calculation likewise and so. There are different traits of the reliability to plan and investigation for the part based software framework[15].

DISCUSSION

Architecture design connects the activity of requirements engineering to conventional detailed design by providing a top-level design incorporating the main design decisions. In this chapter, we discussed the first step in the architecture design process, i.e., designing the first version of the architecture based on the functional requirements. The method for architecture design presented in this part of the book has a focus on the explicit evaluation of and design for quality requirements, but that does not mean that the functional requirements are irrelevant. Before one can start to optimize an architecture for quality requirements, the first step must be to design a first version of the architecture based on the functional requirements. The first design phase discussed in this chapter consists of three main steps. The first step concerned with determining the context of the system under design, the interfaces of the system to the external entities it interacts with and the behaviour the system should exhibit at the interfaces. In addition, the variability required from the components is specified. The second step is the identification of the archetypes, i.e., the main architectural abstractions, and the relations between the archetypes. Our experience is that finding these archetypes is very important especially for the later phases. Architecture transformations tend to build additional structures around the archetypes for fulfilling quality requirements. The final step in functionality-based architectural design is the recursive decomposition of the system into subsystems and the description of system instances using the archetypes and the system interfaces. Since the architecture, for instance in the case of product-line architectures, may be required to support a number of different instantiations, these have to be specified explicitly to verify that the system, in addition to the commonality also supports the required variability.

CONCLUSION

In this chapter we have described how component-based applications are built by using basic constructs like components, ports and contracts. Next, different types of models have been discussed. These models allow one to subdivide the application into coherent parts. An explicit construct for annotating nonfunctional constraints has been defined. Contracts have a specification and a runtime meaning: they are used to annotate constraints at design time, and to monitor these at runtime. This runtime monitoring of contracts is valuable if one wants to detect non-functional failures at runtime. Of course, a contract-based approach does not prove the correctness of the application. This requires the use of static verification methods. The approach that was presented in this paper is based on some well-known principles: components, constraint specification languages, contracts, and runtime monitoring support. We have combined and extended these basic principles, to enable the construction of embedded software with support for the specification and runtime verification of non-functional constraints.

REFERENCES

[1] B. Lee, H. Choi, B. Min, en D. E. Lee, "Applicability of formwork automation design software for aluminum formwork", *Appl. Sci.*, 2020, doi: 10.3390/app10249029.

[2] L. C. Hopkins *et al.*, "Participation in structured programming may prevent unhealthy weight gain during the summer in school-aged children from low-income neighbourhoods: Feasibility, fidelity and preliminary efficacy findings from the Camp NERF study", *Public Health Nutr.*, 2019, doi: 10.1017/S1368980018003403.

[3] E. Trunzer, A. Wullenweber, en B. Vogel-Heuser, "Graphical modeling notation for data collection and analysis architectures in cyber-physical systems of systems", *J. Ind. Inf. Integr.*, 2020, doi: 10.1016/j.jii.2020.100155.

[4] H. B. Christensen en K. M. Hansen, "An empirical investigation of architectural prototyping", *J. Syst. Softw.*, 2010, doi: 10.1016/j.jss.2009.07.049.

[5] A. A. Alsanad, A. Chikh, en A. Mirza, "A Domain Ontology for Software Requirements Change Management in Global Software Development Environment", *IEEE Access*, 2019, doi: 10.1109/ACCESS.2019.2909839.

[6] S. Gahlot* en R. S. Chhillar, "Design and Implementation of Component based Metric for Software Complexity Measurement", *Int. J. Recent Technol. Eng.*, vol 8, no 3, bll 1093–1098, Sep 2019, doi: 10.35940/ijrte.C4249.098319.

[7] I. Polato en A. M. Silva Filho, "A Component-based Approach to Embedded Software Design", *Electron. Notes Theor. Comput. Sci.*, vol 160, bll 255–273, Aug 2006, doi: 10.1016/j.entcs.2006.05.027.

[8] P. Arató, Z. Ádám Mann, en A. Orbán, "Extending component-based design with hardware components", *Sci. Comput. Program.*, vol 56, no 1–2, bll 23–39, Apr 2005, doi: 10.1016/j.scico.2004.11.003.

[9] D. Alonso, J. Á. Pastor, P. Sánchez, B. Álvarez, en C. Vicente-Chicote, "Generación Automática de Software para Sistemas de Tiempo Real: Un Enfoque basado en Componentes, Modelos y Frameworks", *Rev. Iberoam. Automática e Informática Ind. RIAI*, vol 9, no 2, bll 170–181, Apr 2012, doi: 10.1016/j.riai.2012.02.010.

[10] H. Wei, X. Duan, S. Li, G. Tong, en T. Wang, "A component based design framework for robot software architecture", in *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Okt 2009, bll 3429–3434. doi: 10.1109/IROS.2009.5354161.

[11] J. Dong, P. S. C. Alencar, en D. D. Cowan, "Automating the analysis of design component contracts", *Softw. Pract. Exp.*, vol 36, no 1, bll 27–71, Jan 2006, doi: 10.1002/spe.681.

[12] H. Jingyi, L. Juchao, W. Yaming, en Z. Yalin, "Research on Method of Rapid Software Development Based on Component Design", *Int. J. Adv. Network, Monit. Control.*, vol 3, no 1, bll 58–61, Jan 2018, doi: 10.21307/ijanmc-2018-010.

[13] J. H. Sharp en S. D. Ryan, "A theoretical framework of component-based software development phases", *ACM SIGMIS Database DATABASE Adv. Inf. Syst.*, vol 41, no 1, bll 56–75, Feb 2010, doi: 10.1145/1719051.1719055.

[14] P. Rana en R. Singh, "A Design of Cohesion and Coupling Metrics for Component based Software Systems", *Int. J. Comput. Appl.*, vol 146, no 4, bll 22–27, Jul 2016, doi: 10.5120/ijca2016910670.

[15] S. Alcaraz-Corona, J. L. C. Mata, en F. Torres-Castillo, "Exploratory factor analysis for software development projects in Mexico", *Stat. Optim. Inf. Comput.*, 2019, doi: 10.19139/soic.v7i1.512.

CHAPTER 16

FUNCTIONALITY BASED DESIGN ILLUSTRATION

Ms. Surbhi Agarwal, Associate Professor, Department of Computer Science, Jaipur National University, Jaipur, India, Email Id-surbhiagarwal2k19@jnujaipur.ac.in

ABSTRACT:

Functionality refers to whether a design works and helps the users meet their goals and needs. From products such as chairs or tables to designs such as books or web interfaces, functionality varies but is everywhere. When a design is highly functional, it does what it's expected to do and does it well. Functionality design would be to document that when a user clicks a specific link, a pop-up box will ask for their name or when a user clicks the print button, the document will print.

KEYWORDS:

Computer Science, Information Technology, Software Engineering, Software Design.

INTRODUCTION

In this chapter, we illustrate the functionality-based design for the example systems discussed in the previous chapter. Although based on real-world systems, the designs presented here have been modified and simplified for illustrative purposes. Because of this, the designs presented here may seem somewhat small and naive, but since these designs will be transformed in to incorporate the quality requirements, we are forced to keep the size of the initial designs small.

i. Fire Alarm Systems

A fire-alarm system is a relatively autonomous system, but it does provide a number of interfaces to its context. The first issue to decide whether the mechanical and hardware parts of the detectors and alarm devices are part of the system or not. Since we are concerned with the architecture of the software system, we consider those parts to be external and consequently interfaces exist between the software system and the physical detectors and alarm devices. As second issue that we need to decide upon is whether the communication system is part of the system at hand, because the fire-alarm system is highly distributed in nature. In this case, we decide that communication is included as a part since it forms an integral part of the fire-alarm system functionality. A second interface of the system is towards the operator of the system. In the case of an alarm, but also for activating and deactivating parts of the system and monitoring its behaviour[1].This amount of variability of the functionality of the interface is very large, but one can identify a number of core issues that need to be retrievable via the operator interface, such as the location of an alarm or fault warning in the building. Part of this interface is the interaction with external contacts that

need to be notified when the system enters certain states, e.g. alarm, such as the local fire station. A third interface, although related to the previous, is concerned with the interaction to other building automation systems. Other systems may be interested in certain events that take place in the fire-alarm system and may request to be notified. Similarly, the fire-alarm system may want to affect the state and behaviour of other systems, e.g. in case of a fire in a part of the building, the passage-control system may be ordered to unlock all doors in that part allowing people to leave the building without having to use their cards and codes at every door [2].

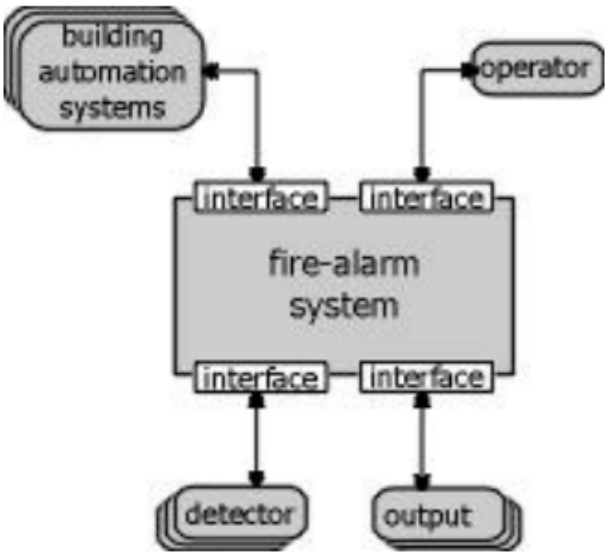

Figure 1: Illustrated that the Interfaces of the Fire Alarm System [3]**.**

In Figure 1, the interfaces provided by the fire-alarm system are presented graphically. As discussed earlier, in a real design, one would assign functional and quality requirements to the identified interfaces and define the interaction at these interfaces in more detail.

However, we leave this step here to avoid exposing unnecessary details of the system. Identifying the archetypes [4]. When searching for entities that grasp the behaviour of several entities and are still abstract, one can detect a number of candidates. Among these, we will use the following as archetypes:

i. Point:

The notion of a point represents highest-level abstraction concerning fire-alarm domain functionality. It is the abstraction of the two subsequent archetypes.

ii. Detector:

This archetype captures the core functionality of the fire-detection equipment, including smoke and temperature sensors.

iii. Output:

The output archetype contains generic output functionality, including traditional alarms, such as bells, extinguishing systems, operator interfaces and alarm notification to, e.g., fire stations.

iv. Control Unit:

Since a fire-alarm system is a distributed system by nature, small groups of points are located at control units that interact with the detectors and outputs in the group. Control units are connected to a network and can communicate. The latter is of crucial importance since the detector alarms in one control unit should often lead to the activation of outputs in other control units. In Figure 2, the relations between the archetypes are shown. As discussed earlier, detector and output are specializations of point and points are contained in control units. Control units communicate with other control units to exchange data about detectors and to activate outputs.

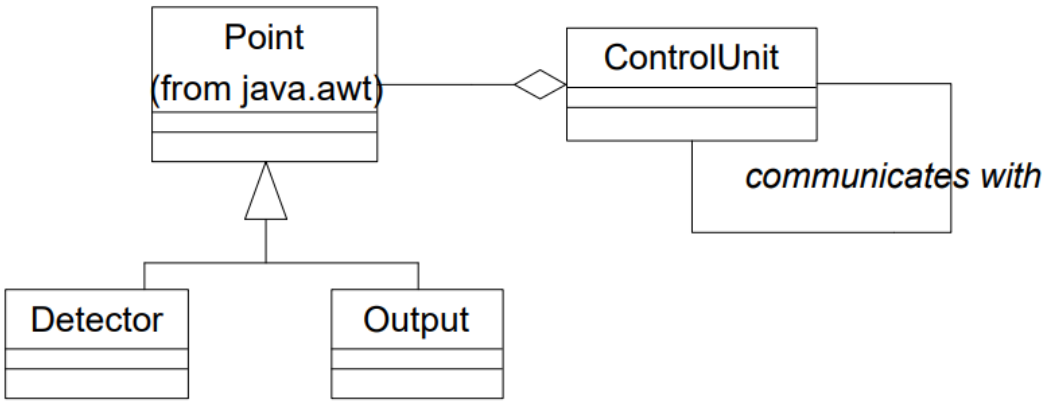

Figure 2: Represented that the Relations between the Fire Alarm System Archetypes [5]**.**

Describing System Instantiations

The first activity in this step is to identify subsystems. The actual system design is decomposed in six main subsystems. However, since we scaled down the actual system for illustrative purposes, we identify only those subsystems directly related to the identified archetypes for the example fire-alarm system as mention in Figure 3. Subsystems of the fire-alarm system to understand the instantiation of the fire-alarm system, to understand the instantiation of the fire-alarm system, we present two system instantiation that are at the two extreme ends of the complexity scale. The first system, represents a small system that might be found, for example, in a single-family house. It consists of a small set of detectors, five smoke detectors in the example, one control unit and two outputs that is a sound alarm and a simple LED-based user interface. The functionality available to the user is to activate or deactivate the system and the feedback from the system is an indication for alarm and one for faults that is internal system errors.

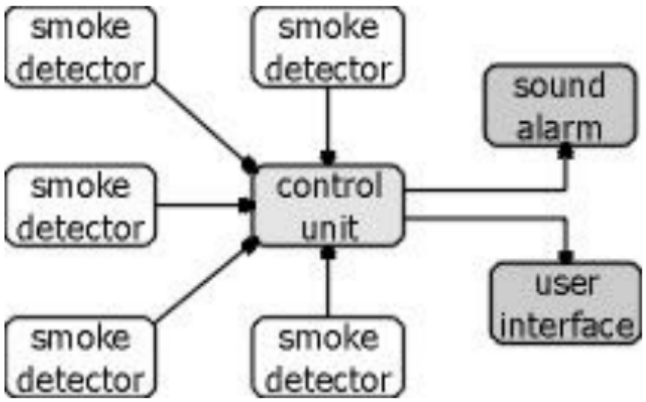

Figure 3: Represented that the Small Fire Alarm System Instantiation [6]**.**

The fire-alarm system covers a site consisting of two buildings and each building is divided into 4 sections which is shown in Figure 4. Each section is supervised by a control unit. One of the control units has an operator interface as a point connected to it. Since the control units are able to communicate with each other, the operator can monitor the complete system from the control unit that the interface is connected to.

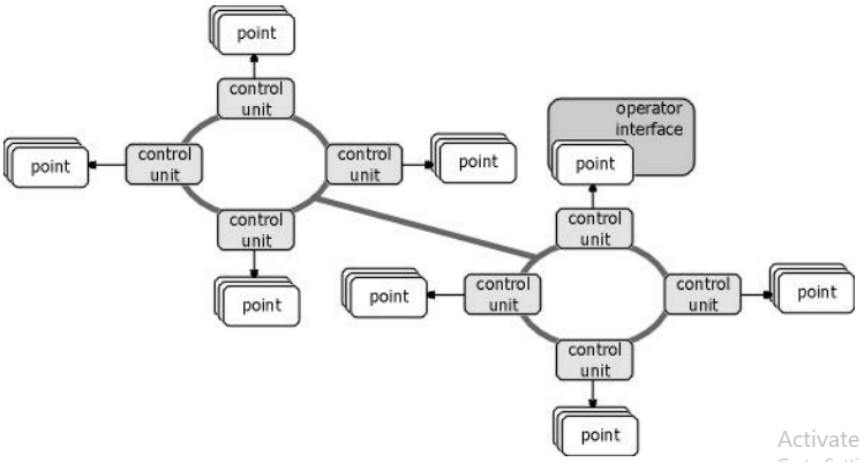

Figure 4: Represented that the Two-building Fire-alarm System.

LITERATURE REVIEW

B. Lee et al. illustrated that the developed formwork automation design software to three target structures, we reviewed the applicability of the formwork automation design software for the aluminum formwork. To apply the formwork automation design software, we built an aluminum formwork library based on the conversion of two-dimensional (2D) computer-aided design (CAD) data to three-dimensional building information modeling data for all the components of the aluminum formwork. The results of the automated formwork layout on the target structures using the formwork automation design software confirmed that the wall and deck members were laid out by the set algorithm according to the formwork size and direction. However, because of the limited functionality of the software, the level of completion of the formwork layout was found to be lower than that of the manual formwork

layout based on 2D CAD data. The currently developed software is based on a simple algorithm, but has a drawback in that the automated layout is limited to only some of its members. Therefore, additional research should be conducted on the development of advanced software through the diversification of the algorithm, automation of preprocessing of the mesh, and analysis of the relationships of all the members comprising the formwork [7].

R. Jarrah et al. illustrated that the ranking of ten structural analysis software applications in terms of six factors: Standardization, Reliability, Longevity, Usability, Price, and Functionality. The study surveyed structural design engineers from various countries around the world, collecting their opinions on the relative importance between the six factors. The respondents were also asked to score ten structural analysis programs for each of the six factors. The factor weights were derived using two methods: Analytic Hierarchy Process (AHP) and a hybrid method that combines AHP with Shannon's entropy. The weighted average of the scores was then used to rank the preference of the programs. The results indicate that the factors of most concern for users are Reliability and Functionality, while Price was of the least concern. Significant differences in preferences were also between certain groups based on location and years of experience. The programs can be classified into three groups: one program that is highly favored, a set of programs that are Above Average, and a set of programs that are Average[8].

P. Ambure et al. stated that the Quantitative structure-activity relationships (QSAR) modeling is a well-known computational technique with wide applications in fields such as drug design, toxicity predictions, nanomaterials, etc. However, QSAR researchers still face certain problems to develop robust classification-based QSAR models, especially while handling response data pertaining to diverse experimental and/or theoretical conditions. In the present work, we have developed an open-source standalone software "QSAR-Co" to setup classification-based QSAR models that allow mining the response data coming from multiple conditions. The software comprises two module the first one is the Model development module and next is the screen or predict module. This user-friendly software provides several functionalities required for developing a robust multitasking or multimarket classification-based QSAR model using linear discriminant analysis or random forest techniques, with appropriate validation, following the principles set by the Organization for Economic Co-operation and Development (OECD) for applying QSAR models in regulatory assessments [9].

I. Almomani and A. Alromi discussed that the applying software engineering processes is vital to critical and complex systems including security and networking systems. Nowadays, Wireless Sensor Networks and their applications are found in many military and civilian systems which make them attractive to security attackers. The increasing risks and system vulnerabilities of WSNs have encouraged researchers and developers to propose many security solutions including software-based Intrusion Detection Systems (IDSs). The main drawbacks of current IDSs are due to the lack of clear, structured software development processes. Unfortunately, a substantial gap has been observed between WSN and SE research communities. Integrating SE and WSNs is an emerging topic that will be expanded as technology evolves and spreads in all life aspects. Consequently, this paper highlighted the importance of Requirement Engineering, Software Design, and Testing when developing IDSs for WSNs. Three software IDS designs were proposed in this study: Scheduling, Broadcast, and Watchdog designs. The three designs were compared in terms of consumed

energy and network lifetime. Moreover, conclusions were drawn in regard to applying software engineering processes to IDSs to deliver the required functionalities, with respect to operational constraints, with an improved performance, accuracy and reliability [10].

R. Spicer et al. illustrated that the field of metabolomics has expanded greatly over the past two decades, both as an experimental science with applications in many areas, as well as in regards to data standards and bioinformatics software tools. The diversity of experimental designs and instrumental technologies used for metabolomics has led to the need for distinct data analysis methods and the development of many software tools. To compile a comprehensive list of the most widely used freely available software and tools that are used primarily in metabolomics. A comprehensive list of the most used tools was compiled. Each tool is discussed within the context of its application domain and in relation to comparable tools of the same domain. This review presents the most widely used tools for metabolomics analysis, categorized based on their main functionality. As future work, we suggest a direct comparison of tools' abilities to perform specific data analysis tasks e.g. peak picking [11].

L. Melgar Estrada et al. illustrated that the variety of specialized tools designed to facilitate analysis of audio-visual media are useful not only to media scholars and oral historians but to other researchers as well. Both Qualitative Data Analysis Software packages and dedicated systems created for specific disciplines, such as linguistics, can be used for this purpose. Software proliferation challenges researchers to make informed choices about which package will be most useful for their project. This paper aims to present an information science perspective of the scholarly use of tools in qualitative research of audio-visual sources. It provides a baseline of affordances based on functionalities with the goal of making the types of research tasks that they support more explicit that is transcribing, segmenting, coding, linking, and commenting on data). We look closely at how these functionalities relate to each other, and at how system design influences research tasks [12].

A. Gleadall discussed that a new concept is presented for the design of additive manufacturing procedures, which is implemented in open-source software called Full Control GCode Designer. In this new design approach, the user defines every segment of the print-path along with all printing parameters, which may be related to geometric and non-geometric factors, at all points along the print-path. Machine control code (GCode) is directly generated by the software, without the need for any programming skills and without using computer-aided design (CAD), STL-files or slicing software. Excel is used as the front end for the software, which is written in Visual Basic. Case studies are used to demonstrate the broad range of structures that can be designed using the software, including: precisely controlled specimens for printer calibration, parametric specimens for hardware characterization utilizing hundreds of unique parameter combinations, novel mathematically defined lattice structures, and previously inconceivable 3D geometries that are impossible for traditional slicing software to achieve. Parametric design files use a few bytes or kilobytes of data to describe all details that are sent to the printer, which greatly improves share ability by eliminating any risk of errors being introduced during STL file conversion or due to different users having inconsistent slicer settings. Adjustable parameters allow GCode for revised designs to be produced instantly, instead of the laborious traditional routine using multiple software packages and file conversions. The full Control design concept offers new opportunities for creative and high-precision use of additive manufacturing systems. It facilitates design for additive manufacturing (DfAM) at the smallest possible scale based on the fundamental nature of the process[13].

W. Ben et al. illustrated that the evolving of Fifth Generation (5G) networks is becoming more readily available as a significant driver of the growth of new applications and business models. Vehicular Ad hoc Networks (VANETs) and Software Defined Networking (SDN) represent the critical enablers of 5G technology with the development of next-generation intelligent vehicular networks and applications. In recent years, researchers have focused on the integration of SDN and VANET, and looked at different topics related to the architecture, the benefits of software-defined VANET services, and the new functionalities to adapt them. However, the security and robustness of the complete architecture is still questionable and have been largely neglected by the research community. Moreover, the deployment and integration of different entities and several architectural components drive new security threats and vulnerabilities. In this paper, first, we survey the state-of-the-art SDN based Vehicular ad-hoc Network (SDVN) architectures for their networking infrastructure design, functionalities, benefits, and challenges. Then we discuss these architectures against major security threats that violate the key security services such as availability, privacy, authentication, and data integrity. We also discuss different countermeasures for these threats. Finally, we present the lessons learned with the directions of future research work towards provisioning stringent security solutions in new SDVN architectures. To the best of our knowledge, this is the first work that presents a comprehensive survey and security analysis on SDVN architectures, and we believe that it will help researchers to address various challenges (e.g., flexible network management, control and high resource utilization, and scalability) in vehicular communication systems which are required to improve the future intelligent transportation systems [14].

D. Spinellis et al. illustrated that the Unix has evolved for almost five decades, shaping modern operating systems, key software technologies, and development practices. Studying the evolution of this remarkable system from an architectural perspective can provide insights on how to manage the growth of large, complex, and long-lived software systems. Along main Unix releases leading to the FreeBSD lineage we examine core architectural design decisions, the number of features, and code complexity, based on the analysis of source code, reference documentation, and related publications. We report that the growth in size has been uniform, with some notable outliers, while cyclomatic complexity has been religiously safeguarded. A large number of Unix-defining design decisions were implemented right from the very early beginning, with most of them still playing a major role. Unix continues to evolve from an architectural perspective, but the rate of architectural innovation has slowed down over the system's lifetime. Architectural technical debt has accrued in the forms of functionality duplication and unused facilities, but in terms of cyclomatic complexity it is systematically being paid back through what appears to be a self-correcting process. Some unsung architectural forces that shaped Unix are the emphasis on conventions over rigid enforcement, the drive for portability, a sophisticated ecosystem of other operating systems and development organizations, and the emergence of a federated architecture, often through the adoption of third-party subsystems. These findings have led us to form an initial theory on the architecture evolution of large, complex operating system software[15].

D. Falessi et al. stated that the architecture of a software-intensive system can be defined as the set of relevant design decisions that affect the qualities of the overall system functionality; therefore, architectural decisions are eventually crucial to the success of a software project. The software engineering literature describes several techniques to choose among architectural alternatives, but it gives no clear guidance on which technique is more suitable than another, and in which circumstances. As such, there is no systematic way for software

engineers to choose among decision-making techniques for resolving tradeoffs in architecture design. In this article, we provide a comparison of existing decision-making techniques, aimed to guide architects in their selection. The results show that there is no "best" decision-making technique; however, some techniques are more susceptible to specific difficulties. Hence architects should choose a decision-making technique based on the difficulties that they wish to avoid. This article represents a first attempt to reason on meta-decision-making, that is, the issue of deciding how to decide [16].

DISCUSSION

Architecture design connects the activity of requirements engineering to conventional detailed design by providing a top-level design incorporating the main design decisions. In this chapter, we discussed the first step in the architecture design process, i.e., designing the first version of the architecture based on the functional requirements. The method for architecture design presented in this part of the book has a focus on the explicit evaluation of and design for quality requirements, but that does not mean that the functional requirements are irrelevant. Before one can start to optimize an architecture for quality requirements, the first step must be to design a first version of the architecture based on the functional requirements. The first design phase discussed in this chapter consists of three main steps. The first step concerned with determining the context of the system under design, the interfaces of the system to the external entities it interacts with and the behaviour the system should exhibit at the interfaces. In addition, the variability required from the components is specified. The second step is the identification of the archetypes, i.e., the main architectural abstractions, and the relations between the archetypes. Our experience is that finding these archetypes is very important especially for the later phases. Architecture transformations tend to build additional structures around the archetypes for fulfilling quality requirements. The final step in functionality-based architectural design is the recursive decomposition of the system into subsystems and the description of system instances using the archetypes and the system interfaces. Since the architecture, for instance in the case of product-line architectures, may be required to support a number of different instantiations, these have to be specified explicitly to verify that the system, in addition to the commonality also supports the required variability.

CONCLUSION

It is obvious that CBD and CBSE are in the very first phase of their lives. CBD is recognized as a new, powerful approach that will, if not revolutionize, at least significantly change the development of software and software use in general. We can expect that components and component-based services will be widely used by non-programmers for building their applications. Tools for building such applications by component assembly will be developed. Automatic component update over the Internet, already present today in many applications, will be a standard means of application improvement. Another trend we can see is the standardization of domain-specific components on the interface level. This will make it possible to build applications and systems from components purchased from different vendors. The standardization of domain-specific components requires the standardization of domain-specific processes. Widespread work on standardization in different domains is already in progress, a typical example is OPC Foundation, working on a standard interface to make possible interoperability between automation/ control applications, field systems/devices and business/office applications). Support for the exchange of information

between components, applications, and systems distributed over the Internet will be further developed. Works related to XML will be further expanded.

REFERENCES

[1] K. Kogure en Y. Takasaki, "GIS for empirical research design: An illustration with georeferenced point data", *PLoS One*, 2019, doi: 10.1371/journal.pone.0212316.

[2] M. J. Jafari, M. Pouyakian, A. khanteymoori, en S. M. Hanifi, "Reliability evaluation of fire alarm systems using dynamic Bayesian networks and fuzzy fault tree analysis", *J. Loss Prev. Process Ind.*, 2020, doi: 10.1016/j.jlp.2020.104229.

[3] J. Chen en J. Fu, "Fire alarm system based on multi-sensor Bayes network", 2012. doi: 10.1016/j.proeng.2012.01.349.

[4] W. H. Dong, L. Wang, G. Z. Yu, en Z. Bin Mei, "Design of Wireless Automatic Fire Alarm System", 2016. doi: 10.1016/j.proeng.2016.01.149.

[5] S. Festag, "False alarm ratio of fire detection and fire alarm systems in Germany - A meta analysis", *Fire Safety Journal*. 2016. doi: 10.1016/j.firesaf.2015.11.010.

[6] I. M. Al Shereiqi en M. mad Sohail, "Smart Fire Alarm System Using IOT", *J. Student Res.*, 2020, doi: 10.47611/jsr.vi.882.

[7] B. Lee, H. Choi, B. Min, en D. E. Lee, "Applicability of formwork automation design software for aluminum formwork", *Appl. Sci.*, 2020, doi: 10.3390/app10249029.

[8] H. Barracosa, C. B. de los Santos, M. Martins, C. Freitas, en R. Santos, "Ocean Literacy to Mainstream Ecosystem Services Concept in Formal and Informal Education: The Example of Coastal Ecosystems of Southern Portugal", *Frontiers in Marine Science*. 2019. doi: 10.3389/fmars.2019.00626.

[9] P. Ambure, A. K. Halder, H. González Díaz, en M. N. D. S. Cordeiro, "QSAR-Co: An Open Source Software for Developing Robust Multitasking or Multitarget Classification-Based QSAR Models", *J. Chem. Inf. Model.*, vol 59, no 6, bll 2538–2544, Jun 2019, doi: 10.1021/acs.jcim.9b00295.

[10] I. Almomani en A. Alromi, "Integrating Software Engineering Processes in the Development of Efficient Intrusion Detection Systems in Wireless Sensor Networks", *Sensors*, vol 20, no 5, bl 1375, Mrt 2020, doi: 10.3390/s20051375.

[11] R. Spicer, R. M. Salek, P. Moreno, D. Cañueto, en C. Steinbeck, "Navigating freely-available software tools for metabolomics analysis", *Metabolomics*, vol 13, no 9, bl 106, Sep 2017, doi: 10.1007/s11306-017-1242-7.

[12] L. Melgar Estrada en M. Koolen, "Audiovisual Media Annotation Using Qualitative Data Analysis Software: A Comparative Analysis", *Qual. Rep.*, Mrt 2018, doi: 10.46743/2160-3715/2018.3035.

[13] K. McChesney en J. Aldridge, "Weaving an interpretivist stance throughout mixed methods research", *Int. J. Res. Method Educ.*, 2019, doi: 10.1080/1743727X.2019.1590811.

[14] W. Ben Jaballah, M. Conti, en C. Lal, "Security and design requirements for software-defined VANETs", *Comput. Networks*, vol 169, bl 107099, Mrt 2020, doi: 10.1016/j.comnet.2020.107099.

[15] S. Soylucicek, "Looking through the Sphere; Illustration in virtual reality", *Glob. J. Arts Educ.*, 2019, doi: 10.18844/gjae.v9i2.3953.

[16] D. Falessi, G. Cantone, R. Kazman, en P. Kruchten, "Decision-making techniques for software architecture design", *ACM Comput. Surv.*, vol 43, no 4, bll 1–28, Okt 2011, doi: 10.1145/1978802.1978812.

CHAPTER 17

ASSESSMENT OF SOFTWARE ARCHITECTURE

Ms. Rachana Yadav, Assistant Professor, Department of Computer Science, Jaipur National University, Jaipur, India, Email Id- Rachana.yadav@jnujaipur.ac.in

ABSTRACT:

The computation's algorithms and data structures are no longer the primary exploitable flaws when software systems grow in size. As systems are assembled from a variety of parts, a new set of design issues arise due to how the entire system is organized in the software architecture. Informally drawn diagrams of descriptive language, module interconnectivity languages, templates and architectures for systems that cater to the requirements of certain domains, and management theories of component integration methods are just a few of the approaches that this level of technology has been approached.

KEYWORDS:

Computer Science, Information Technology, Software Engineering, Software Design.

INTRODUCTION

One of the core features of the architectural design method is that the quality attributes of a system or application architecture are explicitly evaluated during architecture design; thus without having a concrete system available. Although several notable exceptions exist, our experience is that the traditional approach in software industry is to implement the system and then measure the actual values for the quality system properties. The obvious disadvantage is that potentially large amounts of resources have been put on building a system that does not fulfil its quality requirements. In the history of software engineering, several examples of such systems can be found. Being able to estimate the quality attributes of the system already during early development stages is important to avoid such mishaps [1].

However, the question is how to measure system properties based on an abstract specification such as an architectural design. For obvious reasons, it is not possible to measure the quality attributes of the final system based on the architecture design. Instead, the goal is to evaluate the potential of the designed architecture to reach the required levels for its quality requirements.

For example, some architectural styles, e.g. layered architectures, are less suitable for systems where performance is a major issue, even though the flexibility of this style is relatively high. The assessment of an architecture can have different goals, depending on the ambition level of the software architect and the applicability of the assessment techniques used:

i. Relative Assessment

At the lowest ambition level, the software architect is interested in the comparison of two candidate architectures and is concerned with what architecture is more suited for a particular quality attribute. These two architectures may either be two completely different alternatives, but it may also be two subsequent versions of an architecture, where the latter has been transformed to improve the assessed quality attribute or another. For instance, one may assess two architectures for maintainability to decide which of the two is easiest to maintain. The main disadvantage is that relative assessment only gives a 'boolean' answer, e.g. architecture A is more suited than architecture B for a given quality attribute. In the situation where the software architect has two alternative architectures, A and B, and two quality attributes, e.g. performance and maintainability, and one of the architectures is more suited for performance and the other more suited for maintainability, the architect has too little information to make a decision concerning which alternative is more viable. The one architecture may be only slightly worse for performance, but considerably better for maintainability, but relative assessment gives no information concerning this [2]

ii. Absolute Assessment

At a higher ambition level, the software architect is interested in making absolute statements about the quality attributes of the software architecture. Examples of these are statements about the throughput of the system, average response times of individual actions and the maintenance cost of the system. If the architect is able to perform assessments at the absolute level with an acceptable accuracy, then it is possible to compare the assessment results to the requirement specification and decide on whether the system will fulfil all its requirements, including the quality requirements before the system is actually build. In addition, the comparison of alternative architectures or subsequent versions of an architecture become much more informed and the architect has quantifiable, objective means to select alternatives.

A disadvantage of this approach is that although we know what level the architecture provides for the assessed quality attributes, we have no information about the theoretically maximum (or minimum) values for the quality attributes. Thus, we have assessment results that, for instance, predict performance and maintainability levels, but we have no clue whether evolving this architecture or a fundamentally different architecture will provide, potentially, much higher performance and much lower maintenance cost[3].

iii. Assessment of Theoretical Maximum1

At the highest level of assessment we assess an architecture both for its current level and for its theoretical maximum or minimum for the relevant quality attributes. The gap between current and maximum levels allows us to determine whether evolving the architecture is still useful and whether we need to start making trade-offs between quality attributes, and potentially renegotiate with the stake-holders to change the requirement specification, or that either evolving the current architecture or a fundamentally different architecture would be able to incorporate both quality requirements without conflicts. In our experience, the currently available techniques for architecture assessment allows us to make absolute statements about quality attributes at the software architecture level although perhaps not at

the level of accuracy that we would like, but we have no means to predict theoretical maximum or minimum values for software architectures [4].

Assessing Software Architectures Profiles

Quality requirements such as performance and maintainability are, in our experience, generally specified rather weakly in industrial requirement specifications. In some of our cooperation projects with industry, the initial requirement specification contained statements such as the maintainability of the system should be as good as possible and the performance should be satisfactory for an average user". Such subjective statements, although well intended, are useless for the evaluation of software architectures. For example, discusses the quantitative specification of quality requirements and presents useful examples. Several research communities, e.g. performance, real-time and reliability, have developed techniques used for the specification and assessment of their particular quality requirement. Typical for these techniques is that they tend to require considerable effort from the software engineer for creating specifications and predictions [5].

Secondly, since the ambition is to produce detailed and accurate results, these techniques generally require information about the system under development that is not available during architectural design, but earliest during detailed design. Since we are interested in making predictions early in the design process, before the important, architectural design decisions cannot be revoked, other techniques are required that do not require as much detailed information and, consequently, may lead to less precise results, but give at least indications of the quality of the prediction. Finally, it is important to note that the software engineering industry at large has not adopted the techniques developed by the quality-attribute research communities. One explanation might be that within an engineering discipline, each activity is a balance of investment and return. These techniques may not have provided sufficient return-on-investment, from the perspective of industrial software engineers, to be economically viable.

Complete and Selected Profiles

There are two ways of specifying profiles for quality attributes, i.e. the complete and the selected profiles. When defining a complete profile for a quality attribute, the software engineer defines all relevant scenarios as part of the profile. For example, a usage profile for a relatively small system may include all possible scenarios for using the system perhaps excluding exceptional situations. Based on this complete scenario set, the software engineer is able to perform an analysis of the architecture for the studied quality attribute that, in a way, is complete since all possible cases are included. It should, at this point, be clear to the reader that complete profiles only work in a limited number of cases. It requires systems to be relatively small and one is only able to predict complete scenario sets for some quality attributes. For instance, in order to predict the maintainability of the system, the definition of a complete profile assumes that one is able to define all changes that will be required from the system during its operation, or during a predefined period of time. It is safe to assume that this is impossible in all but highly artificial situations [6].

The alternative to complete profiles are selected profiles. Selected profiles are analogous to the random selection of sample sets from populations in statistical experiments. Assuming the

sample set is selected according to some requirements, among others randomness, the results from the sample set can be generalized to the population as a whole. One of the major problems in experiment-based research is the truly random selection of elements from the population, because of practical or ethical limitations. However, even if one does not succeed to achieve random selection, but is forced to make a structured selection of elements for the sample set, the research methodology developed for a weaker form of experimentation, known as quasi-experimentation, allows one to still make scientifically validated statements.

Defining Profiles

The most difficult issue in defining a profile is, obviously, the selection of scenarios that become part of the profile. Since these scenarios are selected and defined by software engineers and other stakeholders, it is hard to claim that this process is random. In our experience, totally unsupported selection and definition of scenarios leads, in some cases, to situations where particular types of scenarios, e.g. changes to the user interface, become overrepresented. To address this, we divide the process of profile specification into two main steps:

i. Definition of Scenario Categories

The first step in profile specification is the decomposition of the scenario 'population' into a number of smaller populations that cover particular aspects of the system. For instance, in the case of usage profiles, one may identify different users of the system, e.g. local user, remote user, operator, etc. To give a second example, in the case of maintenance profiles, one may identify changes to the different interfaces to the context of the system, e.g. the hardware, the communication protocols, the user-interface, etc. In our experience, we normally define around 6 categories, but this is heavily dependent on the type of system and the intentions of the software architects for defining the profiles[7].

ii. Selection and Definition of Scenarios for each Category

In the second step, the software architects select, for each category, a set of scenarios that is representative for the sub-population. Of course, we moved the problem of representativeness one level down from the profile as a whole to the category. However, in our experience, when dealing with a particular category, e.g. hardware changes in a maintenance profile, it is considerably easier to cover all relevant aspects in that category. In addition, even if the scenarios within a category are not representative, the resulting profile will still be closer to the ideal compared to not using categories. Finally, in our experience, we select and define up to 10 scenarios per category for quality attributes that are crucial for the system and 3 to 5 scenarios for important quality attributes. The fact that humans are part of the process of selecting scenarios and categories can be considered a weakness, especially compared to automated random selection. However, it is not possible to perform random selection for the definition and the alternative is to not use scenarios and architecture analysis at all. And, as discussed in the introductory chapter, we know what the lack of assessment early in the design process leads.

Quality Attribute Profiles

In the disposition, we have, up to this point, implicitly indicated that each quality attribute has an associated profile. Although this is true for several system attributes, some profiles can actually be used for assessing more than one quality attribute. In this section, we briefly describe the most important quality attributes and their associated profiles. The following quality attributes can be considered as the most relevant from a general software system perspective.

Scenario-based Assessment

In the remainder of this chapter, a number of approaches to architecture assessment are discussed. The approaches have different advantages and disadvantages, but tend to complement each other. In this section, we discuss scenario-based assessment of software architectures. Scenario-based assessment is directly depending on the profile defined for the quality attribute that is to be assessed. The effectiveness of the technique is largely dependent on the representativeness of the scenarios. If the scenarios form accurate samples, the evaluation will also provide an accurate result. Scenario-based assessment of functionality is not new. Several object-oriented design methods use scenarios to specify the intended system behaviour, e.g. use-cases and scenarios. The main difference to the object-oriented design methods is twofold.

First, we use scenarios for the assessment of quality attributes, rather than for describing and verifying functionality. Second, in addition to use scenarios, we also use other scenarios that define other quality attributes, e.g. change and hazard scenarios. If, however, the software architect decides to use traditional use-cases during architectural design and the use profile is a selected profile, it might be important to define the use-cases independent of the use profile. The reason is that the architecture design will, most likely, be optimized for the set of use-cases. If the set of use cases and the use profile are the same, then one can no longer assume that the assessment of the architecture based on the profile is representative for the scenario population as a whole. However, while developing the scenarios, it is not necessary to develop two sets. The two sets could be generated later by randomly dividing the initially specified set of scenarios. In addition, depending on the system, it might be necessary to develop new scenarios for evaluation purposes if the design is iterated a number of times.

Scenario-based assessment consists of two main steps:

i. Impact Analysis

As an input to this step, the profile and the software architecture are taken. For each scenario in the profile, the impact on the architecture is assessed. For a change scenario, the number of changed and new components and the number of changed and new lines of code could be estimated. For performance, the execution time of the scenario could be estimated based on the path of execution, the predicted component execution times and the delays at synchronization points. The results for each scenario are collected and summarized.

ii. Quality Attribute Prediction

Using the results of the impact analysis, the next step is to predict the value of the studied quality attribute. For performance, the scenario impact data can be used to calculate

throughput by combining the scenario data and the relative frequency of scenarios. For maintainability, the impact data of the change scenarios allows one to calculate the size in changed and new lines of code for an average change scenario. Using a change request frequency figure that is either estimated or based on historical data, one can calculate a total number of changed and new lines of code. Using historical data within the company or figures from the research literature, the software architect can calculate the maintenance cost by, for instance, multiplying the number of work hours per maintained line of code with the total number of maintained lines of code.

Simulation-based Assessment

The assessment technique discussed in the previous section is rather static in that no executing dynamic model is used. An alternative is simulation-based assessment in which a high-level implementation of the software architecture is used. The basic approach consists of an implementation of the components of the architecture and an implementation of the context of the system.

The context, in which the software system is supposed to execute, should be simulated at a suitable abstraction level. This implementation can then be used for assessing the behaviour of the architecture under various circumstances. Once a simulated context and high-level implementation of the software architecture are available, one can use scenarios from the relevant profiles to assess the relevant quality attributes. Robustness, for example, can be evaluated by generating or simulating faulty input to the system or by inserting faults in the connections between architecture entities. The process of simulation-based assessment consists of a number of steps:

i. Define and Implement the Context

The first step is to identify the interfaces of the software architecture to its context and to decide how the behaviour of the context at the interfaces should be simulated. It is very important to choose the right level of abstraction, which generally means to remove most of the details normally present at system interfaces. For instance, for an actuator in the context of the system requiring a duty cycle to be generated, the simulated actuator would accept, for instance, a flow or temperature setting.

Especially for embedded systems where time often plays a role, one has to decide whether time-related behaviour should be implemented in the system. For instance, when increasing the effect of the heater in the dialysis system, the actual water temperature will increase slowly until a new equilibrium is met. The software architect has to decide whether such delays in effects between actuators and sensors should be simulated or not. This decision is depending on the quality attributes that the software architect intends to assess and the required accuracy of the assessment.

Finally, for architecture simulation as a whole, but especially for the simulation of the system context, one has to make an explicit balance between cost and benefit. One should only implement at the level of realism required to perform the assessments one is interested.

ii. Implement the Architectural Components

Once the system context has been defined and implemented, the components in the software architecture are con strutted. The description of the architecture design should at least define the interfaces and the connections of the components, so those parts can be taken directly from the design description. The behaviour of the components in response to events or messages on their interface may not be specified as clearly, although there generally is a common understanding, and the software architect needs to interpret the common understanding and decide upon the level of detail associated with the implementation.

Again, the domain behaviour that is implemented for each component as well as the additional functionality for collecting data is again dependent on the quality attributes that the software architect intends to assess. For some quality attributes, additional architecture description data needs to be associated with the components. For instance, in the case of performance assessment, estimated execution times may be associated with operations on the component interfaces. In that case, generally a simulated system clock is required in order to be able to calculate throughput and average response time figures.

iii. Implement Profile

Depending on the quality attribute that the software architect intends to assess using simulation, the associated profile will need to be implemented in the system. This generally does not require as much implementation effort as the context and the architecture, but the software architect should be able to activate individual scenarios as well as run a complete profile using random selection based on the normalized weights of the scenarios. For example, in virtually all cases, the use profile needs to be implemented. The software architect generally is interested in performing individual use scenarios to observe system behaviour, but also to simulate the system for an indefinite amount of time using a scenario activator randomly selecting scenarios from the profile.

iv. Simulate System and Initiate Profile

At this point, the complete simulation, including context, architecture and profile(s) is ready for use. Since the goal of the simulation is to assess the software architecture, the software architect will run the simulation and activate scenarios in a manual or automatic fashion and collect results.

The type of result depends on the quality attribute being assessed. It is important to note that for several quality attributes, the simulation will actually run two profiles. For instance, for assessing safety, the system will run its use profile in an automated manner, and the hazard scenarios will, either manually or automatically, be activated. Whenever a hazard scenario occurs, data concerning the system context is collected. For example, in the dialysis system example, all values relevant to the safety of the patient, e.g. blood temperature, concentrate density, air bubbles, heparin density, etc. are collected from the simulation to see if any of these values, perhaps temporarily, exceed safety boundaries.

v. Predict Quality Attribute

The final step is to analyse the collected data and to predict the assessed quality attribute based on the data. Depending on the type of simulation and the assessed quality attribute, excessive amounts of data may be available that need to condense. Generally, one prefers to automate this task by extending the simulation with functionality for generating condensed output or using other tools. To give an example, for performance assessment, the system may have run tens of thousands of use scenario instances and collected the times required to execute the scenario instances. All this data needs to be condensed to average execution times per scenario, perhaps including a standard deviation. This allows one to make statements about system throughput, scenario-based throughput and average response times of individual scenarios.

Mathematical Model-based Assessment

Various research communities, e.g. high-performance computing, reliable systems real-time systems, etc., have developed mathematical models that can be used to evaluate especially operational quality attributes. Different from the other approaches, the mathematical models allow for static evaluation of architectural design models. The software engineer can develop a mathematical representation that can be analysed. Mathematical modelling is an alternative to simulation since both approaches are primarily suitable for assessing operational quality attributes. However, the approaches can also be combined. For instance, performance modelling can be used to estimate the computational requirements of the individual components in the architecture. These results can then be used in the simulation to estimate the computational requirements of different use scenarios in the architecture.

The process of model-based assessment consists of the following steps:

i. Select and Abstract a Mathematical Model

As mentioned in the introduction, most quality attribute-oriented research communities have developed mathematical models for assessing 'their' quality attribute. The models are generally well-established, at least within the community, but tend to be rather elaborate in that much, rather detailed, data and analysis is required. Consequently, part of the required data is not available at the architectural level and the technique requires too much effort for architecture assessment as part of an iterative design process. Thus, the software architect is required to abstract the model. This may result in less precise prediction, but that is, within limits, acceptable at the software architecture design level.

ii. Represent the Architecture in Terms of the Model

The mathematical model that has been selected and abstracted does not necessarily assume that the system it models consists of components and connections. For instance, real-time task models assume the system to be represented in terms of tasks. Consequently, the architecture needs to be represented in terms of the model.

iii. Estimate the Required Input Data

The model, even when abstracted, often requires input data that is not included in the basic architecture definition. This data, then, has to be estimated and deduced from the requirement specification and the designed software architecture. For instance, real-time task models require data about, among others, priority, frequency, deadline and computational requirements of tasks and information about the synchronization points in the system.

iv. Predict the Quality Attribute

Once the model is defined, the architecture expressed in terms of the model and all required input data available, the software architect is able to calculate the resulting prediction for the assessed quality attribute. In some cases, for instance non-trivial performance assessments based on the performance engineering method may require more advanced approaches.

LITERATURE REVIEW

D. Mulfari et al. stated that the Assistive Technology includes hardware peripherals, software applications and systems that enable a user with a disability to use a PC. Thus, when a disabled user needs to work in a particular environment has to properly configure the used PC. However, often, the configuration of AT software interfaces is not trivial at all. This paper presents the software design, implementation, and evaluation of a computer system architecture providing a software user-friendly man machine interface for accessing AT software in cloud computing. The main objective of such an architecture is to provide a new type of software human-computer interaction for accessing AT services over the cloud. Thus, end users can interact with their personalized computer environments using any physical networked PC. The advantage of this approach is that users do not have to install and setup any additional software on physical PCs and they can access their own AT virtual environments from everywhere. In particular, the usability of prototype based on the Remote Desktop Protocol (RDP) is evaluated in both private and public cloud scenarios [8].

A. Bhatt et al. illustrated that the Reproducible Software Environment is an open-source software tool enabling computationally reproducible scientific results in the geospace science community. Resen was developed as part of a larger project called the Integrated Geoscience Observatory, which aims to help geospace researchers bring together diverse datasets from disparate instruments and data repositories, with software tools contributed by instrument providers and community members. The main goals of InGeO are to remove barriers in accessing, processing, and visualizing geospatially resolved data from multiple sources using methodologies and tools that are reproducible. The architecture of Resen combines two mainstream open-source software tools, Docker and JupyterHub, to produce a software environment that not only facilitates computationally reproducible research results, but also facilitates effective collaboration among researchers. In this technical paper, we discuss some challenges for performing reproducible science and a potential solution via Resen, which is demonstrated using a case study of a geospace event. Finally we discuss how the usage of mainstream, open-source technologies seems to provide a sustainable path towards enabling reproducible science compared to proprietary and closed-source software [9].

P. Pandey discussed that the design of software solution for delivery as a shared service over Cloud requires specific considerations. In this chapter we describe an approach for design of infrastructure resource management as a service for use by group of institution based on Service Oriented Architecture, Software-as-a-Service, and Cloud Computing paradigms. Our goal in this paper is to propose an architecture mechanism that allows the hiding of a large quantity of data as possible in a database to provide them to other institutions without resulting much difference between the original institution having the data and the other institute accessing them significantly. A fusion of implementation, analysis and evaluation has to be done for hiding information [10].

J. Zuckerman et al. illustrated that the Oone of the most critical aspects of integrating loosely-coupled accelerators in heterogeneous SoC architectures is orchestrating their interactions with the memory hierarchy, especially in terms of navigating the various cache-coherence options: from accelerators accessing off-chip memory directly, bypassing the cache hierarchy, to accelerators having their own private cache. By running real size applications on FPGA-based prototypes of many-accelerator multi-core SoCs, we show that the best cache-coherence mode for a given accelerator varies at runtime, depending on the accelerator's characteristics, the workload size, and the overall SoC status. Chameleon applies reinforcement learning to select the best coherence mode for each accelerator dynamically at runtime, as opposed to statically at design time. It makes these selections adaptively, by continuously observing the system and measuring its performance. Cohmeleon is accelerator-agnostic, architecture independent, and it requires minimal hardware support. Cohmeleon is also transparent to application programmers and has a negligible software overhead. FPGA-based experiments show that our runtime approach offers, on average, a 38% speedup with a 66% reduction of off-chip memory accesses compared to state-of-the-art design-time approaches. Moreover, it can match runtime solutions that are manually tuned for the target architecture[11].

G. Baruffa et al. ststed that due to the growing number of devices accessing the Internet through wireless networks, the radio spectrum has become a highly contended resource. The availability of low-cost radio spectrum monitoring sensors enables a geographically distributed, real-Time observation of the spectrum to spot inefficiencies and to develop new strategies for its utilization. The potentially large number of sensors to be deployed and the intrinsic nature of data make this task a Big Data problem. In this work we design, implement, and validate a hardware and software architecture for wideband radio spectrum monitoring inspired to the Lambda architecture. This system offers Spectrum Sensing as a Service to let end users easily access and process radio spectrum data. To minimize the latency of services offered by the platform, we fine tune the data processing chain. From the analysis of sensor data characteristics, we design the data models for MongoDB and Cassandra, two popular NoSQL databases. A MapReduce job for spectrum visualization has been developed to show the potential of our approach and to identify the challenges in processing spectrum sensor data. We experimentally evaluate and compare the performance of the two databases in terms of application processing time for different types of queries applied on data streams with heterogeneous generation rate [12].

M. Raspopović et al. stated that the technology continues to evolve and develop, requirements for effective teaching and learning methodologies are likewise growing and changing. As a result of this trend, Learning Management Systems (LMSs) are usually adopted in certain institutions and sometimes in order to take advantage of technological

developments. However, these LMSs are adapted to a certain degree and may restrain users with its set of tools and functionalities. New requirements imply the need for integration with third-party systems and tools which are often used to increase the efficacy of learning. Properly implemented technology can serve as a tool to create new opportunities for learning systems. This work focuses on the design and technological requirements of the software architecture for integrating an institutional e-learning system, an educational management system, and a social learning environment. This work proposes a software architecture that supports functionalities promoting effective teaching and learning, while giving an overview of the diversity of technologies and tools used in the proposed architecture [13].

H. Breivold et al. illustrated that the Software evaluability describes a software system's ability to easily accommodate future changes. It is a fundamental characteristic for making strategic decisions, and increasing economic value of software. For long-lived systems, there is a need to address evaluability explicitly during the entire software lifecycle in order to prolong the productive lifetime of software systems. For this reason, many research studies have been proposed in this area both by researchers and industry practitioners. These studies comprise a spectrum of particular techniques and practices, covering various activities in software lifecycle. However, no systematic review has been conducted previously to provide an extensive overview of software architecture evaluability research. Objective: In this work, we present such a systematic review of architecting for software evaluability. The objective of this review is to obtain an overview of the existing approaches in analyzing and improving software evaluability at architectural level, and investigate impacts on research and practice. Method: The identification of the primary studies in this review was based on a pre-defined search strategy and a multi-step selection process [14].

DISCUSSION

The reason we stress these approaches is because we hope to progress the state of the art towards quantitative, objective assessment rather than the current state-of-practice that often is subjective and qualitative. However, it is by no means our intention to diminish the value of architecture assessment through objective reasoning based on earlier experiences and logical argumentation. At numerous occasions, we have encountered experienced software architects and engineers who provided valuable insights that proved extremely helpful in avoiding bad design decisions. Although some of these experiences are based on anecdotal evidence, most can often be justified by a logical line of reasoning. This approach is different from the other approaches in that the evaluation process is less explicit and more based on subjective factors such as intuition. The value of this approach should, nevertheless, not be underestimated. Most software architects we have worked with had well-developed intuitions about ‘good’ and ‘bad’ designs. Their analysis of problems often started with the ‘feeling’ that something was wrong. Based on that, an objective argumentation was constructed either based on one of the aforementioned approaches or on logical reasoning. In addition, this approach may form the basis for the other evaluation approaches. For example, an experienced software engineer may identify a maintainability problem in the architecture and, to convince others, define a number of scenarios that illustrate this. To give an example, during the design of the fire alarm system architecture, it was identified that the system is inherently concurrent. Consequently, it was necessary to choose a concurrency model. Earlier experience by some team members in earlier small embedded systems had shown that fine-grain concurrency with a pre-emptive scheduler could be error-prone considering the possibility of race conditions.

CONCLUSION

We have presented three architecture assessment techniques, i.e., scenario-based, simulation-based and mathematical model-based architecture assessment. The scenario-based approach assesses the impact of the scenarios in the profile and predicts the quality attribute based on the impact data. Simulation-based assessment develops an abstract system context that is simulated and a high-level implementation of the architecture. Generally, for practical reasons, also the profile that is used for the assessment is implemented. During the simulation, relevant data is collected and the quality attribute can be predicted using the collected data. Finally, the software architect can use a, often adapted, mathematical model developed by one of the quality attribute research communities. The adapted model can be used to predict the quality attribute. We have discussed the importance of experience in software architecture assessment and design.

Although our goal is to improve the state of practice by providing objective and quantitative means to reason about architectures, it is explicitly not our intention to diminish the value of experience and creative insight in the architecture design process. Experienced and creative software architects and engineers are a necessary ingredient in any successful software development project. Finally, we have briefly mentioned the overall software architecture assessment process. This process can be divided in two parts. The first part is performed once during the design of a software architecture and includes activities such as selecting and defining the relevant quality attributes, developing the associated profiles and selecting an assessment technique for each quality attribute. The second part of the process is performed for each iteration of the architecture and consists of performing the assessment for the relevant quality attributes, collecting the results and to decide upon continuation, renegotiation or termination of the design project.

REFERENCES

[1] C. Hofmeister, P. Kruchten, R. L. Nord, H. Obbink, A. Ran, en P. America, “A general model of software architecture design derived from five industrial approaches”, *J. Syst. Softw.*, vol 80, no 1, bll 106–126, Jan 2007, doi: 10.1016/j.jss.2006.05.024.

[2] A. Baabad, H. B. Zulzalil, S. Hassan, en S. B. Baharom, “Software architecture degradation in open source software: A systematic literature review”, *IEEE Access*. 2020. doi: 10.1109/ACCESS.2020.3024671.

[3] C. Cho, K. S. Lee, M. Lee, en C. G. Lee, “Software Architecture Module-View Recovery Using Cluster Ensembles”, *IEEE Access*, 2019, doi: 10.1109/ACCESS.2019.2920427.

[4] P. Zhang, H. Muccini, en B. Li, “A classification and comparison of model checking software architecture techniques”, *J. Syst. Softw.*, 2010, doi: 10.1016/j.jss.2009.11.709.

[5] J. Bishung *et al.*, “A critical analysis of topics in software architecture and design”, *Adv. Sci. Technol. Eng. Syst.*, 2019, doi: 10.25046/aj040228.

[6] J. Cruz-Benito, F. J. García-Peñalvo, en R. Therón, "Analyzing the software architectures supporting HCI/HMI processes through a systematic review of the literature", *Telemat. Informatics*, 2019, doi: 10.1016/j.tele.2018.09.006.

[7] C. E. Gonzalez, C. J. Rojas, A. Bergel, en M. A. Diaz, "An Architecture-Tracking Approach to Evaluate a Modular and Extensible Flight Software for CubeSat Nanosatellites", *IEEE Access*, 2019, doi: 10.1109/ACCESS.2019.2927931.

[8] D. Mulfari, A. Celesti, en M. Villari, "A computer system architecture providing a user-friendly man machine interface for accessing assistive technology in cloud computing", *J. Syst. Softw.*, 2015, doi: 10.1016/j.jss.2014.10.035.

[9] A. Bhatt, T. Valentic, A. Reimer, L. Lamarche, P. Reyes, en R. Cosgrove, "Reproducible Software Environment: A tool enabling computational reproducibility in geospace sciences and facilitating collaboration", *J. Sp. Weather Sp. Clim.*, 2020, doi: 10.1051/swsc/2020011.

[10] P. Pandey, "Survey: SOA Architecture in Cloud Computing", *Int. J. Data Sci. Technol.*, vol 2, no 1, bl 1, 2016, doi: 10.11648/j.ijdst.20160201.11.

[11] L. Khalid, *Software architecture for business*. 2019. doi: 10.1007/978-3-030-13632-1.

[12] G. Baruffa, M. Femminella, M. Pergolesi, en G. Reali, "Comparison of MongoDB and Cassandra Databases for Spectrum Monitoring As-a-Service", *IEEE Trans. Netw. Serv. Manag.*, vol 17, no 1, bll 346–360, Mrt 2020, doi: 10.1109/TNSM.2019.2942475.

[13] M. Raspopović, S. Cvetanović, D. Stanojević, en M. Opačić, "Software architecture for integration of institutional and social learning environments", *Sci. Comput. Program.*, vol 129, bll 92–102, Nov 2016, doi: 10.1016/j.scico.2016.07.001.

[14] H. P. Breivold, I. Crnkovic, en M. Larsson, "A systematic review of software architecture evolution research", *Inf. Softw. Technol.*, vol 54, no 1, bll 16–40, Jan 2012, doi: 10.1016/j.infsof.2011.06.002.

CHAPTER 18

TRANSFORMATION OF SOFTWARE ARCHITECTURE

Ms. Surbhi Agarwal, Associate Professor, Department of Computer Science, Jaipur National University, Jaipur, India, Email Id-surbhiagarwal2k19@jnujaipur.ac.in

ABSTRACT:

Software architecture is grabbing attention in both the forward- and reverse engineering industries as a means of thinking and communication about software systems particularly their quality. We have already created an effort to integrate these categories using a semantic reengineering approach known CORUM. We provide a precise illustration of an architecting effort with architectural justification in this chapter. In carrying out another assignment, we rebuild the construction, analyses the rebuilt construction, inspire a change in the architecture with contemporary architecture quality standards, and instantiate the architecture.

KEYWORDS:

Software Architecture, Architectural Analysis, Architecture Reconstruction, Transformation

INTRODUCTION

During functionality-based architecture design, the structure of the system has been determined by the application domain and the functional requirements. The identified archetypes and the system instances described using the archetypes are based on the software architect's perception of the domain. Since the perception of the software architect is largely formed based on the culture in which the architect lives and the education that he or she has received. Consequently, it is likely that other software architects and engineers will share the perception of the software architect that designed the initial version of the architecture. The fact that domain understanding is shared among, at least, the software engineering community is an important property of the functionality-based design, since it allows for easy communication between members of the community. Thus, when software development based on the software architect design is initiated, it is relatively easy for the software architect to explain the important concepts underlying the design[1].

In addition, if the software architect disappears during the development, the persons taking over will have an easier task to understand the architecture design and maintain the conceptual integrity. Finally, during maintenance, software maintainers will have easier understanding of the constraints, rules and rationale underlying the architecture, thereby maintaining conceptual integrity and, consequently, slow the software aging process. Thus, the functionality-based design of the architecture is based on domain analysis that is formed by the culture and education and thus, up to some extent, shared by, at least, the software engineering community. As mentioned, this has important advantages. However, there is an important issue that has remained implicit: the functionality-based architecture design may not fulfil the quality requirements put on the system! Performance, maintainability and other quality attributes of the architecture may not be satisfactory [2].

Assessment of the software architecture, as discussed in the previous chapter, is performed to collect information on the quality attributes of the architecture so that these can be compared to the requirements. If one or more of the quality requirements are not satisfied, the

architecture has to be changed to improve the quality attributes. This is the process of architecture transformation, which is the topic of this chapter. Architecture transformation requires the software engineer to analyse the architecture and to decide due to what cause the property of the architecture is inhibited. Often, the assessment generates hints as to what parts or underlying principles cause low scores.

The assessment of the quality attributes is performed assuming a certain context, consisting of certain subsystems, e.g. databases or GUI systems and one or more operating systems and hardware platforms. Consequently, whenever a quality attribute is not fulfilled, one may decide to either make changes to the presumed context of the system architecture or to make changes to the architecture itself. If it is decided that the software architecture, rather than the context or the requirement specification, should be changed the architecture is subjected to a series of one or more architecture transformations. Each transformation leads to a new version of the architecture that has the same domain functionality, but different values for its properties [3].

The consequence of architecture transformations is that most transformations affect more than one property of the architecture; generally some properties positively and others in a negative way. For instance, the Strategy design pattern increases the flexibility of a class with respect to exchanging one aspect of its behaviour. On the down-side, performance is often reduced since instances of the class have to invoke another object for certain parts of their behaviour. In the general case, however, the positive effect of increased flexibility considerably outweighs the minor performance impact. We have identified four categories of architecture transformations, organized in decreasing impact on the architecture, i.e. imposing an architectural style, imposing an architectural pattern, applying a design pattern and converting quality requirements to functionality[4].

One transformation does not necessarily address a quality requirement completely. Two or more transformations might be necessary. In the sections below, each category is discussed in more detail. Although the transformation of a software architecture is necessary to fulfil its quality requirements, there are two important disadvantages of changing the functionality-based architectural design. First, the transformed architecture will not be as close to the shared understanding of the domain, requiring software architects and engineers to spend more time to understand the ‘philosophy’ underlying the architecture. For instance, the functionality-based architectural design of the measurement system consists of four components. The object-oriented framework that we designed for the measurement system domain consists of more than 30 components.

These components have been added through architecture transformations improving the quality attributes, but not changing the domain functionality represented by the architecture. Second, the design tends to blow up in the number of components. Most transformations will take one or a few components and reorganize the functionality by dividing it over more components. To use the aforementioned Strategy design pattern as an example, the pattern transforms one into at least three classes, i.e. the original class without the factored out behaviour, the abstract strategy class defining the interface and, at least, one concrete strategy class providing one variety of the factored out behaviour. Since most transformations increase the number of components in the architecture, it easily becomes the case that an elegant and simple functionality-based architectural design is transformed into a large and complex set of components that bears no visible relation to the initial architecture. During the complete architecture design process, it is of crucial importance to keep things as simple as possible and to search for conceptual integrity, a notion hard to quantify but understood by each software engineer[5].

One may wonder whether the architecture design method presented in this book is not making design overly complicated. It is important to observe that we defined this method based on a number of architecture design projects that we have been involved in. Generally, researchers from our research group formed an architecture design team with software architects from the companies we cooperated with. We tried to reflect about the way the team and its members performed architectural design. Based on the experiences from the projects, we have made the implicit architecture design process as we experienced it explicit.

A second disadvantage that we try to attack in our explicit approach to architecture design is that ad-hoc architecture design approaches tend to lead to mismatches between the perceived problems and the actual problems and between problems and the solutions selected to address those problems. In several industrial architecture designs we have seen the use of styles, patterns or other solution technology that did not match with the problems, i.e. unfulfilled quality requirements, the project members were trying to solve. Subsequently, further analysis showed that the quality requirements the project members tried to solve were not the problematic quality attributes of the architecture [6].

Converting the ad-hoc, implicit architecture design approach into an architecture design method explicitly organizing the process into a number of well-defined steps will help avoid problems as described above. Finally, a note on the level of detail that should be achieved during architectural design. This is a function of the size of the system, the quality requirements and the required level of certainty. If the system is very large, architectural design is unable to penetrate the challenges of the individual parts of the system. The quality requirements, however, are the most important factor deciding on the level of detail. The goal of architectural design is to develop a software architecture that, with a sufficient level of certainty, will fulfil all its requirements, including the quality requirements. Thus, if the quality requirements are very challenging and close to the boundary of the technical capabilities, architectural design needs to go into considerable detail for the critical parts of the system, e.g. down to the behavior of individual classes. The important issue is not to avoid performing design tasks normally considered to be part of detailed design, but to make sure that the final system will fulfils its quality requirements in addition to its functional requirements [7].

The Architecture Transformation Process

Before presenting the categories of architecture transformation techniques that we have identified, we present a process of architecture transformation that intends to put these categories into a context. Transforming the architecture according to an architectural style or a design pattern is not an independent event, but occurs as part of a larger process of problem identification, solution selection and solution application. The first step of the process is to identify what quality requirements are not fulfilled by the current version of the software architecture. In addition, information about the discrepancy between the assessed level for the quality attribute and the requirement is collected. Based on difference between assessed level and the requirement for each quality attribute and their relative importance, we define a ranking of the quality attributes. The ranking indicates what quality attributes should be addressed first[5].

Note that, as mentioned earlier, only those quality attributes are part of the assessment and transformation process that have explicitly been selected by the software architect as having crucial importance for the success of the system, normally up to about five attributes. Finally, the assessed and required levels for the quality attributes that are fulfilled are also noted. This information is used later on in the selection of transformations. The following steps of the

process are, in principle, repeated for each quality attribute. However, since transformations affect more than one quality attribute, decisions concerning the selection of transformation will be based on all the relevant quality attributes.

The second step is to identify, for the quality attribute currently addressed, at what components or locations in the architecture the quality attribute is inhibited. The assessment performed in the previous phase has led to a quantitative prediction, but while assessing the architecture the software architect normally get several hints on what components represent bottlenecks for the quality attribute. For instance, when performing impact analysis during scenario-based maintainability assessment, there often is one that play a role in multiple scenarios. For some reason, the functionality captured by that component is sensitive to requirement changes. Such kinds of indications often give valuable input on what aspects of the architecture need to be changed. The third step is the selection of a transformation that will solve the identified problem spots in the architecture[8].

Generally, several different transformations can be used, differing in scope and impact, but also on their effects on the other quality attributes. For the selection of the most appropriate transformation, it is important to explicitly analyse the effects of the transformation on the other quality attributes. Based on this analysis, we select the transformation that does not affect any unfulfilled quality attributes in a negative manner, but only quality attributes for which there is a satisfying (positive) difference between the assessed and required level. For instance, if the system performance is well satisfied by the current architecture, it is acceptable to select transformations that improve maintainability at the expense of performance. However, if both are currently not fulfilled or the assessed level very close to the required level, one should search for other alternatives.

The fourth and final step in the process is to perform the transformation, meaning that the functionality is reorganized according to, e.g. the selected style or pattern, and that the description of the architecture is updated to incorporate the changes. It is important to keep a record of the versions of the software architecture, the assessed levels for each of the relevant quality attributes and the rationale for each transformation. This is both useful when it proves necessary to backtrack to an earlier version since the team reached a dead-end in the design and for future reference by software engineers doing detailed design, implementation or software maintenance. The architecture design method presented in this part of the book assumes a fully objective and quantitative approach[9].

It presents a picture of an idealized software architecture design process because the technology for several aspects of the method are currently not available or have not been disseminated to software industry. For instance, for some quality attributes no validated assessment techniques are available. In addition, for several of the transformations discussed in the remainder of this chapter, the exact effect on the quality attributes of a software architecture is not obvious.

Part of this is due to the fact that performing a design, basically any design, is fundamentally a creative process that cannot be formalized and automated. However, many parts surrounding the creative process can and should be formalized in order to become objective and, potentially, automated. Thus, the method presented here is as much a vision on how we would want to perform software architecture design as it is a viable way of working today. However, it requires at times that one resorts to, e.g. qualitative reasoning or experience-based decisions.

Concluding, the software architecture transformation process consists of four major steps:

i. Identify the QAs that are not fulfilled,

ii. For each QA, identify the locations where the QA is inhibited,

iii. Select the most appropriate transformation,

iv. Perform the transformation

Impose Architectural Style

The first category of architecture transformation is concerned with imposing an architectural style on the software architecture and it present several architectural styles1 that improve the possibilities for certain quality attributes for the system the style is imposed upon and are less supportive for other quality attributes. Certain styles, e.g. the layered architectural style, increase the flexibility of the system by defining several levels of abstraction, but generally decrease the performance of the resulting system. With each architectural style, there is an associated fitness for the quality attributes. The most appropriate style for a system depends primarily on its quality requirements. Transforming an architecture by imposing an architectural style results in a complete reorganization of the architecture [10].

Often virtually all architectural components are affected and the assignment of functionality to the components is reorganized. In addition, the original connections between the components are often affected. Consequently, imposing an architectural style is a transformation with major, architecture-wide impact. Although architectural styles can be merged up to some extent, styles are not orthogonal in the sense that they can be merged arbitrarily. If a second architectural style is selected for a part of the architecture, it is necessary to make sure that the constraints of the two styles do not conflict with each other. A typical example of a software architecture using two styles is the compiler example in where the standard pipes-and-filters compiler architectural style is complemented with a black-board style [11].

The blackboard contains data that needs to be accessible by multiple filters. Although the resulting software architecture contains both styles, constraints of both styles are violated. The more common case is where a subsystem uses an architectural style that is different than the style used at the system level. This use of different architectural styles leads to less conflicts between styles, provided that the subsystem acts as a correct component at the system level. However, when considering conceptual integrity, our experience is that one is able to use the same archetypes and organizing principles at all levels of the system. Thus, if it is possible to use the same style throughout the system, this is preferable.

Styles and Quality Attributes

Architectural styles have been discussed at length in other publications although the suitability of styles for the various quality attributes is not always discussed to the same extent. In this section, we briefly discuss the most fundamental styles that are generally recognized: Pipes-and-filters. The pipes and filters of this style can be viewed as analogous to a chemical plant, in which the filters initiate chemical processes on the material transported through the pipes. The pipes-and-filters style assumes a data-flow network where data flows through the pipes and is processed by the filters. The most well-known instance of this style is implemented in the Unix operating system, in particular the associated command shells. A second example is the standard compiler architecture taught in virtually each computer science study program. In a standard compiler, the scanner, parser, optimizer and code

generator form the filters whereas pipes transport, for instance, character and token data. There exist many varieties of pipes-and-filter implementations and definitions. Variations include pipeline (linear), systems without feedback loops (a-cyclic) and arbitrary (cyclic graph). The common denominator is that the filters operate asynchronously and have little or no state. However, filters do exchange data through pipes and, consequently, some synchronization occurs in that manner. The way data is transported through pipes can be pushing, pulling or asynchronous [11].

In the first approach, entry of data by the source filter will activate the sink filter. This is typically useful when the system is processing data from an external source. The opposite occurs for the pulling approach, where the sink filter activates the source filter. This typically occurs in a compiler where the parser will ask for tokens from the lexer. Finally, when using the asynchronous approach, the pipe will store data entered by the source filter until the sink filter requests this, thereby decreasing the synchronicity in the system.

The application of each style to transform a software architecture has an associated effect on the quality attributes of the architecture. However, the actual effect is as much depending on the type of system modelled by the software architecture as the selected style. Nevertheless, below, we discuss the general effects of the pipe sand filters style on the quality attributes:

i. Performance

The advantage of the pipes-and-filters style from a performance perspective is that the filters form excellent units of concurrency, allowing for parallel processes which generally improves performance, assuming it is used with care. In addition, since the pipes connect the components, the interface for each component is very narrow, reducing the number of synchronization points. The advantage of the pipes-and-filters style discussed above may, however, turn into a disadvantage for performance as well. If each filter only performs a very small unit of computation for each unit of data, the style will lead to many context switches and copying of data, which affects performance negatively.

ii. Maintainability

The maintainability of a pipes and filters system also has two sides. On the positive side, the configuration of filters is generally very flexible, allowing for even run-time reorganization of pipes and filters. Thus, as long as new requirements can be implemented by new filters and reorganization of the network, maintainability of this style is very good. The disadvantage is that requirement changes often affect multiple filters. A typical example are syntax changes or extensions in a compiler. These are generally orthogonal to the compiler components and, consequently, require changes to the lexer, parser, optimizer and code generation component. Especially in larger systems, the disadvantage of this style is that real-world entities represented by the system are decomposed and part of the functionality of multiple filters. Our experience is that, for the systems that we have worked with, the majority of requirement changes affect more than one filter and that, consequently, the pipes-and-filters style is not particularly suitable for maintainability.

iii. Reliability

The reliability of a pipes-and-filters system is dependent on its topology and, as a result, it is hard to generalize over it. However, since the pipes-and-filters style assumes that each external event causes computation in a series of filters, one may deduce that the reliability may be less than in styles where most events lead to computation in only one, or perhaps a few components. The series of filters requires each filter to deliver the specified result in

order for the system to be successful, i.e. analogous to an 'and'-function in boolean logic. In other styles, the primary component handling the event may still be able to deliver a result even if some of the secondary components used by the primary component fail.

iv. Safety

The line of reasoning used for reliability also holds for safety. The fact that correct computation is dependent on several components, increases the chance that some failure will occur. Thus, in cases where passiveness of the system may cause hazardous situations, this causes decreased safety. On the other hand, the fact that the output of the system generally will occur through one or a few filters allows for local verification of reasonable output values.

v. Security

Pipes-and-filter system generally have small and explicitly defined input and output interfaces and a well-defined component topology. This means that access to the system is only available through the defined interfaces, where identity verification, authorization and encryption/decryption can be performed. The same technique can be used at the component level in systems where information of different security levels is present.

vi. Performance

The layered style organizes computational tasks based on level of abstraction, rather than their computational relation. This generally causes functionality related to an external event or request to be divided over multiple layers. For instance, in the case of communication protocols, over all layers. As a consequence, the computation in response to the external event covers multiple layers as well, requiring several method context switches.

This leads to decreased performance and experience has shown that the layered architecture does cause a performance penalty when used. Consequently, communication protocols are generally not implemented according to the OSI 7-layer model, but rather in a reorganized fashion avoiding the disadvantages associated with passing several layer boundaries. With respect to concurrency, there is an important note to make. The naive approach to adding concurrency to a system built using the layered style is to assign each layer its own thread of control. In general, this does not lead to an increase in performance, but may even lead to decreased performance, due to the number of task context switches required to react to a single event. The alternative approach is to assign the events processed by the layer stack their own thread of control and to implement the layers in a re-entrant fashion. This generally leads to an increase in performance [12].

vii. Maintainability

Maintainability of a system is influenced by the way requirement changes affect the system. If most changes can be implemented by changing one or a small number of components or by adding a new component, then the maintainability of the system will be high. Assuming the way the functionality of a layered system is assigned to its layers in an appropriate way, the maintainability of a layered system is generally relatively high. Layers have few dependencies on other layers, allowing for replacement of a layer without changing its superior and subordinate layer. If, however, the functionality of the system is not organized according to the expected requirement changes, maintainability may be compromised since several layers will need to be changed for requirement changes.

viii. Reliability

Similar to the pipes-and-filters style, the layered style requires computation in all or most layers for each external event or request. Due to this, failure of one layer may cause the system as a whole to fail. Consequently, the reliability may be lower than when using some of the subsequent styles. On the positive side, a higher-level layer may contain functionality for handling faults occurring at its lower-level layer. Because the higher-level layer has a better overview of the ongoing computation, it may be able to deal with failures the lower-level layer would not have been able to manage on its own.

ix. Safety

See pipes-and-filters.

x. Security

The layered style supports security rather well due to the fact that all computation starts at the top-level or at the bottom-level, as in the case of communication protocols. This allows for the insertion of security layers performing authorization and, possibly, encryption and decryption of data.

Object-Orientation

The object-oriented style organizes the system in terms of communication objects. Objects are entities that contain some state and operations to access and change this state. Whereas the state and operations are encapsulated by the object, the signatures of the operations are accessible on the interface of the object. Operations can be accessed by sending messages to an object. A message causes the activation of an operation, which may lead to changes to the internal object state and messages to other objects. Messages are synchronous in that the object sending a message waits until it receives a reply and only then continues with its own computation. Several models extend the basic object-oriented style with various aspects. The concurrent object-oriented style, for example, assumes all objects to be potentially active and an object sending a message to another object will delay the thread sending the message, but other threads may be active within the object. Although objects do not need to be aware of the sender of a message, an object is required to have the identity of an object it intends to send a message to. Since objects, in the course of their computation, generally need to send messages to several other objects, each object is required to maintain references to its acquaintances. Consequently, an object-oriented system can be viewed as a network of connected objects. The object-oriented style, similar to the other styles, affects the quality attributes of the system. Below, these effects are discussed:

i. Performance

The performance of object-oriented programming has received considerable attention in the literature, especially in comparison to conventional structured programming. Although new object-oriented languages, at their introduction, generally are less efficient than traditional languages, this disadvantage is generally largely removed at subsequent versions. Typical examples are C++ and Java. However, using the object-oriented style as an organizing principle at the architectural level is even less controversial since it is generally accepted that a system needs to be broken down into components. The question is just whether these components should be filters, layers, objects or of yet another type. The performance of systems based on the object-oriented style is very much dependent on the principles the

designer used to define the objects, but there is not necessarily a fundamental conflict. For optimal maintainability, objects should be selected so that the most likely requirement changes affect as few objects as possible. For optimal performance, objects should be selected so that the most frequent use scenarios cause computation at as few objects as possible, since performing context switches between objects is expensive. Since requirement changes often affect the use scenarios, the optimal system organization for maintainability and for performance may actually be very close to each other. In our experience, we have seen several examples of this. One important note to make is that, similar to the layered style, the naive way of adding concurrency to an object-oriented system is to use the homogenous approach, i.e. each object has its own thread of control. This is generally not optimal if there are many objects in the system due to the large number of context switches and synchronization points for each use scenario. Instead, threads should be attached to external events that cause considerable amounts computation in the system.

ii. Maintainability

The object-oriented programming paradigm became popular due to claims of increased reusability and maintainability. In our cooperation with industry, we have seen many examples of cases where these claims were actually fulfilled. However, the main issue in achieving maintainability

in object-oriented systems is modelling the right objects. As discussed in the previous section, the likely change scenarios should affect the system as little as possible, and preferably lead to the definition of a new subclass or a new type of object aggregation. One reuse inhibitor is the fact that an object require references to the objects it sends messages to, i.e. its acquaintances. Since the types and number of acquaintances is hard-coded in the class specification, changes will always require class specification to be changed. The implicit invocation style discussed below addresses this problem.

iii. Reliability

The object-oriented style is not particularly positive or negative with respect to reliability. One disadvantage that could be mentioned is that fault handling generally has to be managed inside the object, due to the encapsulation. This may make it harder to have fault handling at higher levels in the system, where more information is available. However, the fact that the system is modelled in terms of relatively independent entities is positive for reliability, since no central entity can cause the system to fail.

iv. Safety

One of the basic organizing principles of the object-oriented style is that real-world entities should, as much as possible, be represented as objects. As a consequence, the real-world entities that may compromise or assure safety are also modelled as objects. The fact that each real-world entity has a one-to-one correspondence to a system entity, is positive for safety since the system entity is better suited to identify hazardous situations and react to them than an organization where the behaviour of the system entity is divided over multiple entities.

v. Security

The object-oriented style both encapsulates and fragments the data contained in the system, being positive and negative aspects, respectively. Authorization of access to the system may be simplified by the fact that system interfaces generally will be represented by objects.

DISCUSSION

One of the observations with respect to the current state of practice in systems development is that distribution is becoming ubiquitous. Most systems consist of parts distributed over multiple nodes or need to communicate with other systems via networks. Consequently, distribution is an integrative part of most systems. The problem of distribution consists of two major aspects. The first is the way in which entities connect to each other. This can be achieved through predefined addresses and connections or, more flexible, through a central broker. The second aspect is the actual communication between remote entities. Again, several solutions exist including remote procedure calls and remote method invocation, distributed streams, a web interface, etc. Finally, one way to deal with distribution is by making it transparent, that is the system entities are unaware of their acquaintances being remote or local. Although much of the functionality related to distribution is transparent in today's approaches to distribution, components are often aware of acquaintances being distributed or not, both when binding and when communicating. The solutions used to achieve distribution in a system are typically architectural patterns since they require all entities in the system that are concerned with communication over address spaces to follow the same set of rules and constraints. Below, we discuss some architectural patterns that are typically used in the context of distributed systems that is brokers, remote method invocation and HTTP.

CONCLUSION

Now that the example has been shown in detail, it is instructive to take a step back and think about what this paper is trying to argue for. We have taken a common example of a legacy application and a common set of reengineering goals decouple client and server knowledge of each other to aid in the future modifiability of the system, increase performance, and make it easy to increase the reliability and we have shown how we can reason about each of these goals at the architectural level. We do this by understanding and focusing on the architectural features that we need to manipulate. We can then instantiate the architectural changes via automated code transformation.

This paper is a reaction to the relatively undisciplined way in which reengineering is done. In particular, some efforts at reengineering are code-based, and some are aimed at architecting a system. But few efforts try to do both, and we are aware of little work that explicitly attempts to tie these two approaches together. But tying the approaches together is crucial for ensuring: the conceptual integrity of the architecture being reengineered, the quality attributes of the resulting system, the preservation of component level qualities. In short, what we are presenting here is a disciplined process for reengineering a system that takes into account all levels of representation and reasoning about software. However, it must be stressed that tool support for this type of reasoning is only just beginning to reach maturity. The approach presented here is far from a fully automated solution.

There are several directions in which the work presented in this paper should be extended. The most obvious of these addresses the issue of scale: we would like to explore the applicability of the techniques presented here in larger, real-world software systems. Also, we hope to validate the use of semantic features for driving architectural transformations; evaluation of the adequacy of the feature set as applied to our example will guide evolution of the features.

Finally, we are interested in initiating a process of cataloging additional architectural elements and their features; this will lead to the development of a catalog of architectural transformations and their potential mappings to code transformations.

REFERENCES

[1] A. Alreshidi, A. Ahmad, A. B. Altamimi, K. Sultan, en R. Mehmood, "Software architecture for mobile cloud computing systems", *Futur. Internet*, 2019, doi: 10.3390/fi11110238.

[2] J. Bishung *et al.*, "A critical analysis of topics in software architecture and design", *Adv. Sci. Technol. Eng. Syst.*, 2019, doi: 10.25046/aj040228.

[3] J. Cruz-Benito, F. J. García-Peñalvo, en R. Therón, "Analyzing the software architectures supporting HCI/HMI processes through a systematic review of the literature", *Telemat. Informatics*, 2019, doi: 10.1016/j.tele.2018.09.006.

[4] A. Ahmad, S. Abdulaziz, A. Alanazi, M. N. Alshammari, en M. Alhumaid, "Software architecture solutions for the internet of things: A taxonomy of existing solutions and vision for the emerging research", *Int. J. Adv. Comput. Sci. Appl.*, 2019, doi: 10.14569/ijacsa.2019.0101073.

[5] L. Khalid, *Software architecture for business*. 2019. doi: 10.1007/978-3-030-13632-1.

[6] N. Medvidovic en G. Edwards, "Software architecture and mobility: A roadmap", *J. Syst. Softw.*, 2010, doi: 10.1016/j.jss.2009.11.004.

[7] S. Nakajima *et al.*, "Command-centric architecture (C2A): Satellite software architecture with a flexible reconfiguration capability", *Acta Astronaut.*, 2020, doi: 10.1016/j.actaastro.2020.02.034.

[8] V. Garousi, M. Felderer, en F. N. Kılıçaslan, "A survey on software testability", *Information and Software Technology*. 2019. doi: 10.1016/j.infsof.2018.12.003.

[9] H. V. Gamido en M. V. Gamido, "Comparative review of the features of automated software testing tools", *Int. J. Electr. Comput. Eng.*, 2019, doi: 10.11591/ijece.v9i5.pp4473-4478.

[10] P. Lu, X. Cong, en D. Zhou, "E-learning-oriented software architecture design and case study", *Int. J. Emerg. Technol. Learn.*, 2015, doi: 10.3991/ijet.v10i4.4698.

[11] Z. Hao, E. Novak, S. Yi, en Q. Li, "Challenges and Software Architecture for Fog Computing", *IEEE Internet Comput.*, 2017, doi: 10.1109/MIC.2017.26.

[12] M. A. Babar, B. Kitchenham, L. Zhu, I. Gorton, en R. Jeffery, "An empirical study of groupware support for distributed software architecture evaluation process", *J. Syst. Softw.*, 2006, doi: 10.1016/j.jss.2005.06.043.

CHAPTER 19

PROGRAMMING LANGUAGES FOR THE SOFTWARE ARCHITECTURE

Mr. Hitendra Agarwal, Associate Professor, Department of Computer Science, Jaipur National University, Jaipur, India, Email Id-hitendra.agrawal@jnujaipur.ac.in

ABSTRACT:

As the size and complexity of software systems increases, the design and specification of overall system structure or software architecture emerges as a central concern. Architectural issues include the gross organization of the system, protocols for communication and data access, assignment of functionality to design elements, and selection among design alternatives. Currently system designers have at their disposal two primary ways of defining software architecture. They can use the modularization facilities of existing programming languages and module interconnection languages; or they can describe their designs using informal diagrams and idiomatic phrases. Then we show that regularities in these descriptions can form the basis for architectural description languages.

KEYWORDS:

Architecture Description Languages, Architecture Representation Languages, Software Architecture, Software Design, Module Interconnection.

INTRODUCTION

As the size and complexity of software systems increase, the design and specification of overall system structure become a more significant issue than the choice of algorithms and data structures of computation. Structural issues include the gross organization of the system; the global control structure; the protocols for communication, synchronization, and data access; the assignment of functionality to design elements; the composition of design elements; scaling and performance; and selection among design alternatives. This is the software architecture level of design. Abstractly, software architecture involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns. In general, a particular system is defined in terms of a collection of components and interactions among those components. Such a system may in turn be used as a element in a larger system design. The use of software architectures is pervasive in the informal diagrams and idioms that people use to describe system designs[1].

Designers typically draw "architectural" diagrams consisting of boxes and connecting lines, and allude to common paradigms for describing their meaning. For example, the relation between entities might be described as an instance of a "client-server model a pipeline or a layered architecture. Unfortunately, diagrams and descriptions such as these are highly ambiguous. At best they rely on common intuitions and past experience to have any meaning at all. Moreover, system designers generally lack adequate concepts, tools, and decision criteria for selecting and describing system structures that fit the problem at hand [2]. It is virtually impossible to answer with any precision the many questions that arise during system design. What is a pipeline architecture, and when should one pick it over, say, a layered architecture? What are the consequences of choosing one structural decomposition over

another? Which architectures can be composed with others? How are implementation choices related to the overall performance of these architectures? And so on. The problem of describing structural decompositions more precisely has traditionally been addressed by modularization facilities of programming languages and module interconnection languages[3].

These notations typically allow an implementer to describe software system structure in terms of definition/use or import/export relationships between program units. This supports many features for programming-in-the-large, such as separate compilation, well-defined module interfaces, and module libraries. However, as we show later, such language support is inadequate for architectural descriptions. In particular, descriptions at the level of programming language modules provide only a low-level view of interconnections between components, in which the only directly expressible relationships between components are those provided by the programming language. Moreover, they fail to provide a clean separation of concerns between architectural level issues and those related to choose of algorithms and data structures. More recently, a number of component-based languages have been proposed and implemented[4].

These languages describe systems as configurations of modules that interact in specific, predetermined ways (such as remote procedure call, messages, or events or enforce specialized patterns of organization. While such languages provide new ways of describing interactions between components in a large system, they too are typically oriented around a small, fixed set of communication paradigms and programming-level descriptions or they enforce a very specialized single-purpose organization. This makes them inappropriate for expressing a broad range of architectural designs[5].

In this chapter we consider the need for new higher-level languages specifically oriented to the problem of describing software architecture. First, we show how ideas from "classical" language design apply to the task of describing software architectures. We then detail the characteristics such languages should have in the areas of composition, abstraction, reusability, configuration, heterogeneity, and analysis. Finally, we show how existing approaches fail to satisfy these properties, thus motivating the need for new language design. In taking this general point of view, the intention is not to propose a particular language indeed, we believe that no single language will be sufficient for all aspects of architectural description but rather to establish the framework within which architectural language design must take place [6].

The Linguistic Character of Architectural Description We now illustrate how the structure of the architectural task is amenable to treatment as a language problem and argue that the principles learned from the design of programming languages can serve us well in designing notations for software architecture. The argument is as follows:

i. Analysis of commonly-used architectures reveals common patterns, or idiomatic constructs.

ii. Those constructs rely on a shared set of common kinds of elements; similarly, they rely on a shared set of common intermodal connection strategies.

iii. Languages serve precisely the purpose of describing complex relations among primitive elements and combinations thereof.

iv. It makes sense to define a language when you can identify appropriate semantic constructs; we find an appropriate basis in the descriptions of architectures.

Common Patterns of Software Organization

Papers describing software systems often dedicate a section to the architecture of the system. This section typically contains a box-and-line diagram; usually boxes depict major components and lines depict some communication, control, or data relation among the components. The boxes and lines mean different things from one paper to another, and the terms in the prose descriptions often lack precise meaning. Nevertheless, important ideas are communicated by these descriptions. Some of the informal terms refer to common, or idiomatic, patterns used to organize the overall system. These are often widely used among software engineers in high-level descriptions of system designs. A number of the more pervasive patterns have been identified in descriptions of architectural idioms, and material based on these patterns is beginning to appear in courses on architectural design of software[7]. Among the more common architectural patterns are:

A. Pipes and filters Graph of incremental stream transformers. Examples: Unix pipes, signal processing.

B. Client-server Shared Services provided by request to distributed clients. Examples: File servers, distributed databases.

C. Hierarchical Layers System partitioned into layers, which act as virtual machines. Examples: OS kernels, ISO OSI.

D. Communicating processes System is composition of independent, concurrent processes. Interpreter Examples: Many distributed systems.

E. Execution engine bridge gap between the abstract programs and the machine on which they must run. Examples: Rule-based systems, blackboard shell.

Software developers would clearly benefit from having more precise definitions of these structures, including the forms in which they appear and the classes of functionality and interaction they provide. Initial steps toward this goal have recently appeared [8].

Common Components and Interconnections

In the diagrams of architectural descriptions, the boxes usually have labels that are highly specific to the particular system: "lexical analyser," "alias table," "requisition slip data file." The lines or sometimes adjacencies that represent interactions are similarly specific: "identifiers," "update requests," "inventory levels." Examination of these descriptions shows that if the specific functionality of the elements is set aside, the remaining structural properties often fall into identifiable classes [9]. For example, here are some of the classes of components that appear regularly in architectural descriptions:

A. (Pure) Computation Simple input/output relations, no retained state. Examples: math functions, filters, transforms.

B. Memory Shared collection of persistent structured data. Examples: database, file system, symbol table, hypertext.

C. Manager State and closely related operations. Examples: abstract data type, many servers.

D. Controller Governs time sequences of others' events. Examples: scheduler, synchronizer.

E. Link Transmits information between entities. Examples: communication link, user interface.

The interactions among components are also of identifiable kinds. Some of the most common are:

A. Procedure Call: Data flow Implicit triggering Message passing Shared data Single thread of control passes among definitions. Examples: ordinary procedure call (single name space), remote procedure call (separate name spaces).

B. Data Flow: Independent processes interact through streams of data; availability of data yields control. Examples: UNIX pipes.

C. Implicit Triggering: Computation is invoked by the occurrence of an event; no explicit interactions among processes. Examples: event systems, automatic garbage collection.

D. Message Passing: Independent processes interact by explicit, discrete handoff of data; may be synchronous or asynchronous. Examples: TCP/IP.

E. Shared Data: Components operate concurrently (probably with provisions for atomicity) on same data space Examples: blackboard systems, multiuser databases.

F. Instantiation: Instantiates uses capabilities of instantiated definition by providing space for state required by instance. Examples: use of abstract data types.

The significant thing about these classes of components and forms of interaction is that they are shared by many different architectural idioms that is, the higher-level idioms are composed from a common set of primitives.

From Programming Languages to Software Architecture

One characterization of progress in programming languages and tools has been regular increases in abstraction level or the conceptual size of software designer's building blocks. To place the field of Software Architecture [10] into perspective let us begin by looking at the historical development of abstraction techniques in computer science.

i. High-level Programming Languages

When digital computers emerged in the 1950s, software was written in machine language; programmers placed instructions and data individually and explicitly in the computer's memory. Insertion of a new instruction in a program might require hand-checking of the entire program to update references to data and instructions that moved as a result of the insertion. Eventually it was recognized that the memory layout and update of references could be automated, and also that symbolic names could be used for operation codes, and memory addresses. Symbolic assemblers were the result. They were soon followed by macro processors, which allowed a single symbol to stand for a commonly-used sequence of instructions. The substitution of simple symbols for machine operation codes, machine addresses yet to be defined, and sequences of instructions was perhaps the earliest form of abstraction in software.

In the latter part of the 1950s, it became clear that certain patterns of execution were commonly useful indeed, they were so well understood that it was possible to create them automatically from a notation more like mathematics than machine language. The first of these patterns were for evaluation of arithmetic expressions, for procedure invocation, and for

loops and conditional statements. These insights were captured in a series of early high-level languages, of which Fortran was the main survivor.

Higher-level languages allowed more sophisticated programs to be developed, and patterns in the use of data emerged. Whereas in Fortran data types served primarily as cues for selecting the proper machine instructions, data types in Algol and it successors serve to state the programmer's intentions about how data should be used. The compilers for these languages could build on experience with Fortran and tackle more sophisticated compilation problems. Among other things, they checked adherence to these intentions, thereby providing incentives for the programmers to use the type mechanism [11]. Progress in language design continued with the introduction of modules to provide protection for related procedures and data structures, with the separation of a module's specification from its implementation, and with the introduction of abstract data types.

ii. Abstract Data Types

In the late 1960s, good programmers shared an intuition about software development: If you get the data structures right, the effort will make development of the rest of the program much easier. The abstract data type work of the 1970s can be viewed as a development effort that converted this intuition into a real theory. The conversion from an intuition to a theory involved understanding

A. The software structure (which included a representation packaged with its primitive operators),

B. Specifications (mathematically expressed as abstract models or algebraic axioms),

C. Language issues (modules, scope, user-defined types),

D. Integrity of the result (invariants of data structures and protection from other manipulation),

E. Rules for combining types (declarations),

F. Information hiding (protection of properties not explicitly included in specifications).

The effect of this work was to raise the design level of certain elements of software systems, namely abstract data types, above the level of programming language statements or individual algorithms. This form of abstraction led to an understanding of a good organization for an entire module that serves one particular purpose. This involved combining representations, algorithms, specifications, and functional interfaces in uniform ways. Certain support was required from the programming language, of course, but the abstract data type paradigm allowed some parts of systems to be developed from a vocabulary of data types rather than from a vocabulary of programming-language constructs [12].

iii. Software Architecture

Just as good programmers recognized useful data structures in the late 1960s, good software system designers now recognize useful system organizations. One of these is based on the theory of abstract data types. But this is not the only way to organize a software system.

Many other organizations have developed informally over time, and are now part of the vocabulary of software system designers. For example, typical descriptions of software architectures include synopses such as (italics ours):

A. Camelot is based on the client-server model and uses remote procedure calls both locally and remotely to provide communication among applications and servers.

B. Abstraction layering and system decomposition provide the appearance of system uniformity to clients, yet allow Helix to accommodate a diversity of autonomous devices. The architecture encourages a client server model for the structuring of applications.

C. We have chosen a distributed, object-oriented approach to managing information.

D. The easiest way to make the canonical sequential compiler into a concurrent compiler is to pipeline the execution of the compiler phases over a number of processors.

A more effective way split the source code into many segments, which are concurrently processed through the various phases of compilation by multiple compiler processes before a final, merging pass recombines the object code into a single program. Other software architectures are carefully documented and often widely disseminated. Examples include the International Standard Organization's Open Systems Interconnection Reference Model a layered network architecture, the NIST/ECMA Reference Model a generic software engineering environment architecture based on layered communication substrate and the X Window System a distributed windowed user interface architecture based on event triggering and call-backs.

We are still far from having a well-accepted taxonomy of such architectural paradigms, let alone a fully-developed theory of software architecture. But we can now clearly identify a number of architectural patterns, or styles, that currently form the basic repertoire of a software architect.

Common Architectural Styles

We now examine some of these representative, broadly-used architectural styles. To make sense of the differences between styles, it helps to have a common framework from which to view them. The framework we will adopt is to treat an architecture of a specific system as a collection of computational components or simply components together with a description of the interactions between these components the connectors. Graphically speaking, this leads to a view of an abstract architectural description as a graph in which the nodes represent the components and the arcs represent the connectors.

As we will see, connectors can represent interactions as varied as procedure call, event broadcast, database queries, and pipes. An architectural style, then, defines a family of such systems in terms of a pattern of structural organization. More specifically, an architectural style determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined. These can include topological constraints on architectural descriptions. Other constraints say, having to do with execution semantics might also be part of the style definition.

i. Pipes and Filters

In a pipe and filter style each component has a set of inputs and a set of outputs. A component reads streams of data on its inputs and produces streams of data on its outputs, delivering a complete instance of the result in a standard order. This is usually accomplished by applying a local transformation to the input streams and computing incrementally so output begins before input is consumed. Hence components are termed filters. The connectors of this style serve as conduits for the streams, transmitting outputs of one filter to inputs of

another. Hence the connectors are termed pipes. Among the important invariants of the style, filters must be independent entities: in particular, they should not share state with other filters.

Another important invariant is that filters do not know the identity of their upstream and downstream filters. Their specifications might restrict what appears on the input pipes or make guarantees about what appears on the output pipes, but they may not identify the components at the ends of those pipes. Furthermore, the correctness of the output of a pipe and filter network should not depend on the order in which the filters perform their incremental processing although fair scheduling can be assumed. Figure 1 illustrates this style. Common specializations of this style include pipelines, which restrict the topologies to linear sequences of filters; bounded pipes, which restrict the amount of data that can reside on a pipe; and typed pipes, which require that the data passed between two filters have a well-defined type.

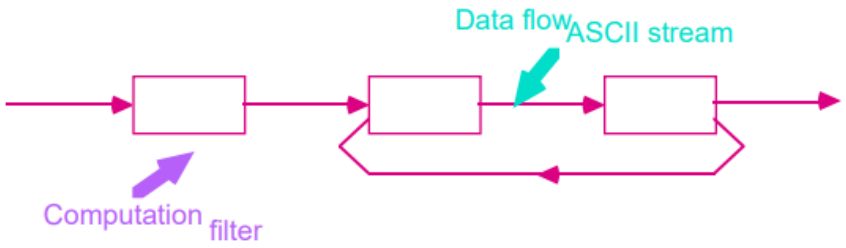

Figure 1: Represented that the Pipes and Filters.

A degenerate case of a pipeline architecture occurs when each filter processes all of its input data as a single entity. In this case the architecture becomes a batch sequential system. In these systems pipes no longer serve the function of providing a stream of data, and therefore are largely vestigial. Hence such systems are best treated as instances of a separate architectural style. The best-known examples of pipe and filter architectures are programs written in the Unix shell. Unix supports this style by providing a notation for connecting components represented as Unix processes and by providing run time mechanisms for implementing pipes.

As another well-known example, traditionally compilers have been viewed as a pipeline system though the phases are often not incremental. The stages in the pipeline include lexical analysis, parsing, semantic analysis, and code generation. Other examples of pipes and filters occur in signal processing domains, functional programming, and distributed systems. Pipe and filter systems have a number of nice properties. First, they allow the designer to understand the overall input/output behaviour of a system as a simple composition of the behaviours of the individual filters. Second, they support reuse: any two filters can be hooked together, provided they agree on the data that is being transmitted between them. Third, systems can be easily maintained and enhanced: new filters can be added to existing systems and old filters can be replaced by improved ones. Fourth, they permit certain kinds of specialized analysis, such as throughput and deadlock analysis. Finally, they naturally support concurrent execution.

Each filter can be implemented as a separate task and potentially executed in parallel with other filters. But these systems also have their disadvantages. First, pipe and filter systems often lead to a batch organization of processing. Although filters can process data incrementally, since filters are inherently independent, the designer is forced to think of each filter as providing a complete transformation of input data to output data. In particular, because of their transformational character, pipe and filter systems are typically not good at

handling interactive applications. This problem is most severe when incremental display updates are required, because the output pattern for incremental updates is radically different from the pattern for filter output. Second, they may be hampered by having to maintain correspondences between two separate, but related streams. Third, depending on the implementation, they may force a lowest common denominator on data transmission, resulting in added work for each filter to parse and unparsed its data. This, in turn, can lead both to loss of performance and to increased complexity in writing the filters themselves.

ii. Data Abstraction and Object-Oriented Organization

In this style data representations and their associated primitive operations are encapsulated in an abstract data type or object. The components of this style are the objects or, if you will, instances of the abstract data types. Objects are examples of a sort of component we call a manager because it is responsible for preserving the integrity of a resource (here the representation). Objects interact through function and procedure invocations. Two important aspects of this style are:

A. That an object is responsible for preserving the integrity of its representation,

B. That the representation is hidden from other objects. Figure 2 illustrates this style.

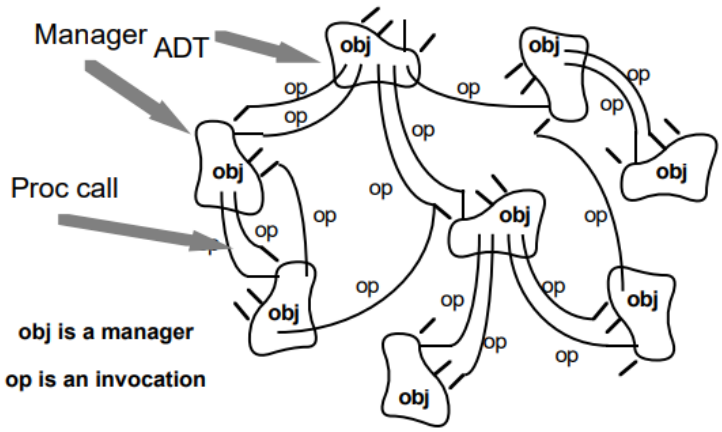

Figure 2: Represented that the Abstract Data Types and Objects [13]**.**

The use of abstract data types, and increasingly the use of object-oriented systems, is, of course, widespread. There are many variations. For example, some systems allow "objects" to be concurrent tasks; others allow objects to have multiple interfaces.

Object-oriented systems have many nice properties, most of which are well known. Because an object hides its representation from its clients, it is possible to change the implementation without affecting those clients. Additionally, the bundling of a set of accessing routines with the data they manipulate allows designers to decompose problems into collections of interacting agents.

The most significant is that in order for one object to interact with another via procedure call it must know the identity of that other object. This is in contrast, for example, to pipe and filter systems, where filters do need not know what other filters are in the system in order to interact with them. The significance of this is that whenever the identity of an object changes it is necessary to modify all other objects that explicitly invoke it. In a moduleoriented language this manifests itself as the need to change the "import" list of every module that uses the changed module. Further there can be side effect problems: if A uses object B and C also uses B, then C's effects on B look like unexpected side effects to A, and vice versa.

iii. Event-based, Implicit Invocation

Traditionally, in a system in which the component interfaces provide a collection of procedures and functions, components interact with each other by explicitly invoking those routines. However, recently there has been considerable interest in an alternative integration technique, variously referred to as implicit invocation, reactive integration, and selective broadcast. This style has historical roots in systems based on actors, constraint satisfaction, daemons, and packet-switched networks.

The idea behind implicit invocation is that instead of invoking a procedure directly, a component can announce one or more events. Other components in the system can register an interest in an event by associating a procedure with the event. When the event is announced the system, itself invokes all of the procedures that have been registered for the event. Thus an event announcement implicitly causes the invocation of procedures in other modules. For example, in the Field system, tools such as editors and variable monitors register for a debugger's breakpoint events. When a debugger stops at a breakpoint, it announces an event that allows the system to automatically invoke methods in those registered tools. These methods might scroll an editor to the appropriate source line or redisplay the value of monitored variables. In this scheme, the debugger simply announces an event, but does not know what other tools are concerned with that event, or what they will do when that event is announced.

Architecturally speaking, the components in an implicit invocation style are modules whose interfaces provide both a collection of procedures (as with abstract data types) and a set of events. Procedures may be called in the usual way. But in addition, a component can register some of its procedures with events of the system. This will cause these procedures to be invoked when those events are announced at run time. Thus, the connectors in an implicit invocation system include traditional procedure call as well as bindings between event announcements and procedure calls.

The main invariant of this style is that announcers of events do not know which components will be affected by those events. Thus, components cannot make assumptions about order of processing, or even about what processing, will occur as a result of their events. For this reason, most implicit invocation systems also include explicit invocation that is normal procedure call as a complementary form of interaction. Examples of systems with implicit invocation mechanisms abound. They are used in programming environments to integrate tools, in database management systems to ensure consistency constraints, in user interfaces to separate presentation of data from applications that manage the data, and by syntax-directed editors to support incremental semantic checking [14].

One important benefit of implicit invocation is that it provides strong support for reuse. Any component can be introduced into a system simply by registering it for the events of that system. A second benefit is that implicit invocation eases system evolution. Components may be replaced by other components without affecting the interfaces of other components in the system. In contrast, in a system based on explicit invocation, whenever the identity provides some system function is changed, and all other modules that import that module must also be changed. The primary disadvantage of implicit invocation is that components relinquish control over the computation performed by the system. When a component announces an event, it has no idea what other components will respond to it. Worse, even if it does know what other components are interested in the events it announces, it cannot rely on the order in which they are invoked. Nor can it know when they are finished. Another problem concerns

exchange of data. Sometimes data can be passed with the event. But in other situations event systems must rely on a shared repository for interaction[15].

In these cases global performance and resource management can become a serious issue. Finally, reasoning about correctness can be problematic, since the meaning of a procedure that announces events will depend on the context of bindings in which it is invoked. This is in contrast to traditional reasoning about procedure calls, which need only consider a procedure's pre- and post-conditions when reasoning about an invocation of it.

iv. Layered Systems

A layered system is organized hierarchically, each layer providing service to the layer above it and serving as a client to the layer below. In some layered systems inner layers are hidden from all except the adjacent outer layer, except for certain functions carefully selected for export. Thus in these systems the components implement a virtual machine at some layer in the hierarchy. (In other layered systems the layers may be only partially opaque.) The connectors are defined by the protocols that determine how the layers will interact. Topological constraints include limiting interactions to adjacent layers. Figure 3 illustrates this style.

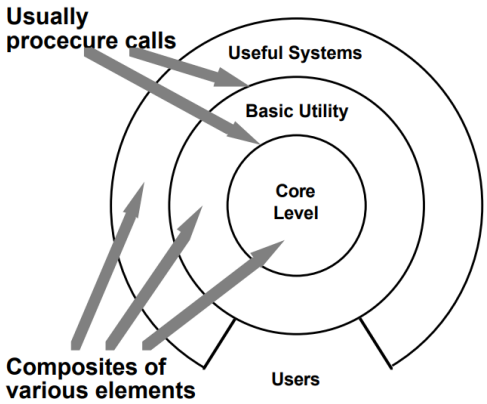

Figure 3: Represented that the Layered Systems[16]**.**

The most widely known examples of this kind of architectural style are layered communication protocols. In this application area each layer provides a substrate for communication at some level of abstraction. Lower levels define lower levels of interaction, the lowest typically being defined by hardware connections. Other application areas for this style include database systems and operating systems.

Layered systems have several desirable properties. First, they support design based on increasing levels of abstraction. This allows implementers to partition a complex problem into a sequence of incremental steps. Second, they support enhancement. Like pipelines, because each layer interacts with at most the layers below and above, changes to the function of one layer affect at most two other layers. Third, they support reuse. Like abstract data types, different implementations of the same layer can be used interchangeably, provided they support the same interfaces to their adjacent layers. This leads to the possibility of defining standard layer interfaces to which different implementers can build but layered systems also have disadvantages. Not all systems are easily structured in a layered fashion and even if a system can logically be structured as layers, considerations of performance may require closer coupling between logically high-level functions and their lower-level implementations. Additionally, it can be quite difficult to find the right levels of abstraction. This is particularly true for standardized layered models. One notes that the communications

community has had some difficulty mapping existing protocols into the ISO framework: many of those protocols bridge several layers.

In one sense this is similar to the benefits of implementation hiding found in abstract data types. However, here there are multiple levels of abstraction and implementation. They are also similar to pipelines, in that components communicate at most with one other component on either side. But instead of simple pipe read or write protocol of pipes, layered systems can provide much richer forms of interaction. This makes it difficult to define system independent layers since a layer must support the specific protocols at its upper and lower boundaries. But it also allows much closer interaction between layers, and permits two-way transmission of information.

v. Repositories

In a repository style there are two quite distinct kinds of components: a central data structure represents the current state, and a collection of independent components operate on the central data store. Interactions between the repository and its external components can vary significantly between systems.

The choice of control discipline leads to major subcategories. If the types of transactions in an input stream of transactions trigger selection of processes to execute, the repository can be a traditional database. If the current state of the central data structure is the main trigger of selecting processes to execute, the repository can be a blackboard. Figure 4 illustrates a simple view of a blackboard architecture.

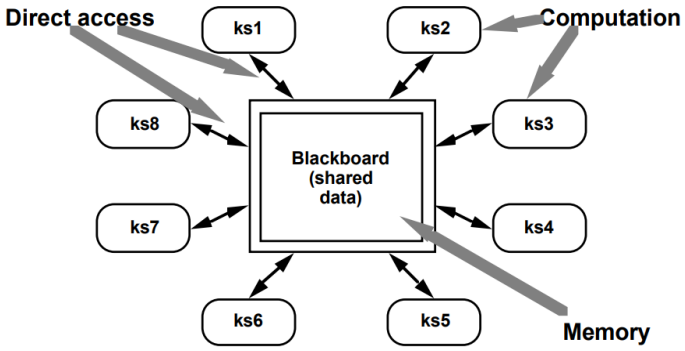

Figure 4: Represented that the Blackboard [17]**.**

a. **The Knowledge Sources:** separate, independent parcels of application dependent knowledge. Interaction among knowledge sources takes place solely through the blackboard.

b. **The Blackboard Data Structure:** problem-solving state data, organized into an application-dependent hierarchy. Knowledge sources make changes to the blackboard that lead incrementally to a solution to the problem.

Control driven entirely by state of blackboard. Knowledge sources respond opportunistically when changes in the blackboard make them applicable.

In the diagram there is no explicit representation of the control component. Invocation of a knowledge source is triggered by the state of the blackboard. The actual locus of control, and hence its implementation, can be in the knowledge sources, the blackboard, a separate module, or some combination of these. Blackboard systems have traditionally been used for applications requiring complex interpretations of signal processing, such as speech and pattern recognition. They have also appeared in other kinds of systems that involve shared

access to data with loosely coupled agents. There are, of course, many other examples of repository systems. Batch sequential systems with global databases are a special case. Programming environments are often organized as a collection of tools together with a shared repository of programs and program fragments. Even applications that have been traditionally viewed as pipeline architectures, may be more accurately interpreted as repository systems.

DISCUSSION

Requirements of analysis address the ability to support automated and non-automated reasoning about architectural descriptions. Different architectures permit different kinds of analysis, and it should be possible to tailor the kind of analysis to the kind of architecture. This goes beyond the current support for analysis, which primarily consists of type checking. When a designer uses a certain set of architectural elements to construct a system it is often because this choice enables analysis of specialized properties of that system. For example, in a pipe and filter architecture, it is possible to analyse properties of throughput, investigate questions of deadlock and resource usage, or infer the input-output behaviour of a system from that of the component filters. It should be possible to tailor special-purpose analysis tools and proof techniques to these architectures. Existing module connection languages provide only weak support for analysis. At best they provide some form of type checking across component boundaries. They rarely permit more semantically-based properties to be analysed or even expressed. It is possible to add specification of input-output behaviour and reason via procedure call proof rules. However, for many other forms of interaction, such as event broadcast, there are currently no corresponding systems of specification and analysis. The need for enhanced forms of analysis are particularly important for architectural formalisms, since many of the interesting architectural properties are dynamic ones. For example, if a connector is associated with a particular protocol, it should be possible to reason about whether the use of that connector is correct in its context of use. Similarly, issues such as timing, performance, and resource usage may play a significant part in reasoning about whether a given architectural description is adequate. The variability of kinds of analyses that one might want to perform on an architectural description argue strongly that no single semantic framework will suffice. Instead, it must be possible to associate specifications with architectures as they become relevant to particular components, connectors, and patterns.

CONCLUSION

The compiler example illustrates how SADL can be used to describe simple textbook" architectures. Each level of detail is described by a SADL architecture specification, employing constructs from the relevant architectural styles. Multiple levels of description are linked by mappings which associate objects declared at one level with corresponding objects at the next level, creating a refinement hierarchy. A systematic way of creating consistent refinement hierarchies is to develop them transformational by applying verified refinement patterns. Our experience suggests that simple, but real, architectures can be generated using a small number of patterns. If you understand the compiler example, you understand SADL well enough to use it for describing simple architectures. SADL has been applied to larger examples, requiring parameterized specifications and other advanced features. For example, SADL has been used to formalize the open Distributed Transaction Processing (DTP) reference architecture [4] at multiple levels of detail. The highest level of the Sadl DTP hierarchy corresponds to the informal data flow diagram in the X/Open documents, a lower level corresponds to the C procedure call model documented in the hundreds of pages of X/Open documents, and still lower levels detail various implementation strategies. The architectures are linked by mappings to form a tree of alternative implementations of the

Open specification. This example illustrates the feasibility of using SADL for real world architectural specification. Recently, we have demonstrated how our research on architecture hierarchy can serve as a basis for secure system design [3]. Our approach was demonstrated by incorporating security directly into the Open DTP architecture. It was necessary to build an architecture hierarchy containing four secure SADL architectures related by formal proofs of the kind described in our earlier papers.

REFERENCES

[1] C. Y. Tsai, "Improving students' understanding of basic programming concepts through visual programming language: The role of self-efficacy", *Comput. Human Behav.*, 2019, doi: 10.1016/j.chb.2018.11.038.

[2] H. Baumeister, F. Hacklinger, R. Hennicker, A. Knapp, en M. Wirsing, "A Component Model for Architectural Programming", *Electron. Notes Theor. Comput. Sci.*, 2006, doi: 10.1016/j.entcs.2006.05.015.

[3] N. Lazebna, Y. Fedorova, en M. Kuznetsova, "Scratch language of programming vs English language: Comparing mathematical and linguistic features", *EUREKA, Phys. Eng.*, 2019, doi: 10.21303/2461-4262.2019.00982.

[4] P. Mardziel en N. Vazou, "PLAS 2019 - ACM SIGSAC workshop on programming languages and analysis for security", 2019. doi: 10.1145/3319535.3353561.

[5] S. B. Jain, S. G. Sonar, S. S. Jain, P. Daga, en R. S. Jain, "Review on Comparison of different programming language by observing it's advantages and disadvantages", *Res. J. Eng. Technol.*, 2020.

[6] W. Van Der Vegt, W. Westera, E. Nyamsuren, A. Georgiev, en I. M. Ortiz, "RAGE Architecture for Reusable Serious Gaming Technology Components", *Int. J. Comput. Games Technol.*, 2016, doi: 10.1155/2016/5680526.

[7] M. Kargar, A. Isazadeh, en H. Izadkhah, "Multi-programming language software systems modularization", *Comput. Electr. Eng.*, 2019, doi: 10.1016/j.compeleceng.2019.106500.

[8] S. Ahmad, Z. Hashim, en S. A. Asmai, "A study on reuse-based requirements engineering by utilizing knowledge pattern", *Int. J. Adv. Sci. Eng. Inf. Technol.*, 2020, doi: 10.18517/ijaseit.10.1.10168.

[9] A. Dimopoulos, "Educational Leadership Effectiveness. Is it a Matter of a Leader's Characteristics, Behaviors, or Leadership Style?", *J. Econ. Manag. Sci.*, 2020, doi: 10.30560/jems.v3n1p13.

[10] R. Wilhelm, "Foundations of programming languages", *Form. Asp. Comput.*, 2020, doi: 10.1007/s00165-021-00561-4.

[11] G. O. Barnett en R. A. Greenes, "High-level programming languages", *Comput. Biomed. Res.*, 1970, doi: 10.1016/0010-4809(70)90010-8.

[12] "Application of High Level Programming Language (Visual Basic): A Review", *J. Sci. Technol.*, 2020, doi: 10.46243/jst.2020.v5.i5.pp90-109.

[13] J. Kook en J. H. Chang, "A high-level programming language implementation of topology optimization applied to the acoustic-structure interaction problem", *Struct. Multidiscip. Optim.*, 2020, doi: 10.1007/s00158-021-03052-5.

[14] D. Krpan, S. Mladenovic, en G. Zaharija, "Mediated transfer from visual to high-level programming language", 2017. doi: 10.23919/MIPRO.2017.7973531.

[15] K. Daungcharone, P. Panjaburee, en K. Thongkoo, "Implementation of mobile game-transformed lecturebased approach to promoting C programming language learning", *Int. J. Mob. Learn. Organ.*, 2020, doi: 10.1504/IJMLO.2020.106168.

[16] J. Ogala, B. Ogala, en J. Onyarin, “Comparative Analysis of C, C++, C# and JAVA Programming Languages”, *Glob. Sci. Journals*, 2020.
[17] J. Beal, T. Lu, en R. Weiss, “Automatic compilation from high-level biologically-oriented programming language to genetic regulatory networks”, *PLoS One*, 2011, doi: 10.1371/journal.pone.0022490.

CHAPTER 20

DISCERNING COUPLING IN SOFTWARE ARCHITECTURE

Mr. Surendra Mehra, Associate Professor, Department of Computer Science, Jaipur National University, Jaipur, India, Email Id-surendra.mehra@jnujaipur.ac.in

ABSTRACT:

The architecture of a software platform is the collection of significant design choices. Over a system's existence, in reality, new architectural choices are introduced, and old methods are changed or reversed. The above choices often depart from the carefully researched aim of the architect, and software systems frequently show increased architectural degeneration over time. The results of such careless design choices are referred to as architectural structure. There hasn't been a thorough investigation of the attributes or patterns associated with this behavior. Instead, both practitioners and researchers had to depend on tradition and their own, intrinsically restricted experience when discussing architectural scents and their detrimental impacts. We describe the reasoned approach we used to examine the characteristics and effects of environmental structure in this research.

KEYWORDS:

Architecture Description Languages, Architecture Representation Languages, Software Architecture, Software Design, Module Interconnection.

INTRODUCTION

One of the most difficult tasks an architect will face is untangling the various forces and trade-offs at play in distributed architectures. People who provide advice constantly extol the benefits of “loosely coupled” systems, but how can architects design systems where nothing connects to anything else? Architects design fine-grained micro services to achieve decoupling, but then orchestration, transactionality, and synchronicity become huge problems. Generic advice says “decouple,” but provides no guidelines for how to achieve that goal while still constructing useful systems. Architects struggle with granularity and communication decisions because there are no clear universal guides for making decisions no best practices exist that can apply to real-world complex systems. Until now, architects lacked the correct perspective and terminology to allow a careful analysis that could determine the best set of trade-offs on a case-by-case basis. Why have architects struggled with decisions in distributed architectures? After all, we’ve been building distributed systems since the last century, using many of the same mechanisms (message queues, events, and so on). Why has the complexity ramped up so much with micro services? The answer lies with the fundamental philosophy of micro services, inspired by the idea of a bounded context. Building services that model bounded contexts required a subtle but important change to the way architects designed distributed systems because now transactionality is a first-class architectural concern [1]–[3] . In many of the distributed systems architects designed prior to micro services, event handlers typically connected to a single relational database, allowing it to handle details such as integrity and transactions. Moving the database within the service boundary moves data concerns into architecture concerns.

When architects look at entangled problems, they struggle with performing trade-off analysis because of the difficulties separating the concerns, so that they may consider them

independently. Thus, the first step in trade-off analysis is untangle the dimensions of the problem, analyzing what parts are coupled to one another and what impact that coupling has on change. For this purpose, we use the simplest definition of the word coupling:

i. Coupling

Two parts of a software system are coupled if a change in one might cause a change in the other.

Often, software architecture creates multidimensional problems, where multiple forces all interact in interdependent ways. To analyse trade-offs, an architect must first determine what forces need to trade off with each other.

Thus, here's our advice for modern trade-off analysis in software architecture:

A. Find what parts are entangled together.

B. Analyse how they are coupled to one another.

C. Assess trade-offs by determining the impact of change on interdependent systems.

While the steps are simple, the hard parts lurk in the details. Thus, to illustrate this framework in practice, we take one of the most difficult and probably the closest to generic problems in distributed architectures, which is related to micro services:

Determining the proper size for microservices seems a pervasive problem too small services create transactional and orchestration issues, and too-large services create scale and distribution issues.

To that end, the remainder of this book untangles the many aspects to consider when answering the preceding question [4] . We provide new terminology to differentiate similar but distinct patterns and show practical examples of applying these and other patterns.

However, the overarching goal of this book is to provide you with example-driven techniques to learn how to construct your own trade-off analysis for the unique problems within your realm. We start with our first great untangling of forces in distributed architectures: defining architecture quantum along with the two types of coupling, static and dynamic.

Architecture

The term quantum is, of course, used heavily in the field of physics known as quantum mechanics. However, the authors chose the word for the same reasons physicists did. Quantum originated from the Latin word quantus, meaning "how great" or "how many." Before physics co-opted it, the legal profession used it to represent the "required or allowed amount" (for example, in damages paid). The term also appears in the mathematics field of topology, concerning the properties of families of shapes [5], [6]. Because of its Latin roots, the singular is quantum, and the plural is quanta, similar to the datum data symmetry. An architecture quantum measures several aspects of both topology and behaviour in software architecture related to how parts connect and communicate with one another:

A. Architecture Quantum

An architecture quantum is an independently deployable artefact with high functional cohesion, high static coupling, and synchronous dynamic coupling. A common example of an architecture quantum is a well-formed micro service within a workflow.

B. Static Coupling

Represents how static dependencies resolve within the architecture via contracts. These dependencies include operating system, frameworks, and/or libraries delivered via transitive dependency management, and any other operational requirement to allow the quantum to operate.

C. Dynamic Coupling

Represents how quanta communicate at runtime, either synchronously or asynchronously. Thus, fitness functions for these characteristics must be continuous, typically utilizing monitors. Even though both static and dynamic coupling seem similar, architects must distinguish two important differences. An easy way to think about the difference is that static coupling describes how services are wired together, whereas dynamic coupling describes how services call one another at runtime. For example, in a microservices architecture, a service must contain dependent components such as a database, representing static coupling the service isn't operational without the necessary data. That service may call other services during the course of a workflow, which represents dynamic coupling. Neither service requires the other to be present to function, except for this runtime workflow. Thus, static coupling analyses operational dependencies, and dynamic coupling analyses communication dependencies.

Independently Deployable

Independently deployable implies several aspects of an architecture quantum each quantum represents a separate deployable unit within a particular architecture. Thus, a monolithic architecture one that is deployed as a single unit is by definition a single architecture quantum. Within a distributed architecture such as micro services, developers tend toward the ability to deploy services independently, often in a highly automated way. Thus, from an independently deployable standpoint, a service within a microservices architecture represents an architecture quantum. Making each architecture quantum represent a deployable asset within the architecture serves several useful purposes. First, the boundary represented by an architecture quantum serves as a useful common language among architects, developers, and operations.

Each understands the common scope under question: architects under stand the coupling characteristics, developers understand the scope of behaviour, and the operations team understands the deployable characteristics. Second, the architecture quantum represents one of the forces architects must consider when striving for proper granularity of services within a distributed architecture. Often, in microservices architectures, developers face the difficult question of what service granularity offers the optimum set of trade-offs. Some of those trade-offs revolve around deploy ability: what release cadence does this service require, what other services might be affected, what engineering practices are involved, and so on [7] .

Architects benefit from a firm understanding of exactly where deployment boundaries lie in distributed architectures. We discuss service granularity and its attendant trade-offs in Chapter 7. Third, independent deploy ability forces the architecture quantum to include common coupling points such as databases. Most discussions about architecture conveniently ignore issues such as databases and user interfaces, but real-world systems must commonly deal with those problems.

Thus, any system that uses a shared database fails the architecture quantum criteria for independent deployment unless the database deployment is in lockstep with the application.

Many distributed systems that would otherwise qualify for multiple quanta fail the independently deployable part if they share a common database that has its own deployment cadence. Thus, merely considering the deployment boundaries doesn't solely provide a useful measure. Architects should also consider the second criteria for an architecture quantum, high functional cohesion, to limit the architecture quantum to a useful scope.

High Functional Cohesion

High functional cohesion refers structurally to the proximity of related elements: classes, components, services, and so on. Throughout history, computer scientists defined a variety of cohesion types, scoped in this case to the generic module, which may be represented as classes or components, depending on platform. From a domain standpoint, the technical definition of high functional cohesion overlaps with the goals of the bounded context in domain-driven design: behaviour and data that implements a particular domain workflow. From a purely independent deploy ability standpoint, a giant monolithic architecture qualifies as an architecture quantum. However, it almost certainly isn't highly functionally cohesive, but rather includes the functionality of the entire system [8], [9] . The larger the monolith, the less likely it is singularly functionally cohesive. Ideally, in a microservices architecture, each service models a single domain or workflow, and therefore exhibits high functional cohesion. Cohesion in this context isn't about how services interact to perform work, but rather how independent and cou□ pled one service is to another service. Any of the monolithic architecture styles will necessarily have a quantum of one, as illustrated in Figure 1.

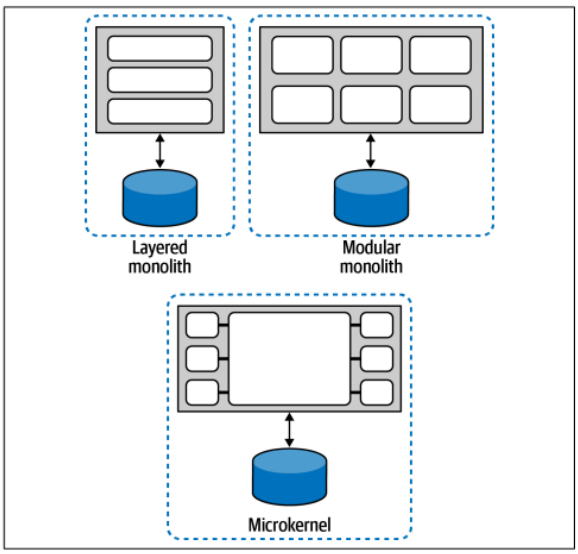

Figure 1: Represented that the Monolithic Architectures.

As you can see, any architecture that deploys as a single unit and utilizes a single database will always have a single quantum. The architecture quantum measure of static coupling includes the database, and a system that relies on a single database cannot have more than a single quantum. Thus, the static coupling measure of an architecture quantum helps identify coupling points in architecture, not just within the software components under development. Most monolithic architectures contain a single coupling point typically, a database that makes its quantum measure one. Distributed architectures often feature decoupling at the component level; consider the next set of architecture styles, starting with the service-based architecture shown in Figure 2.

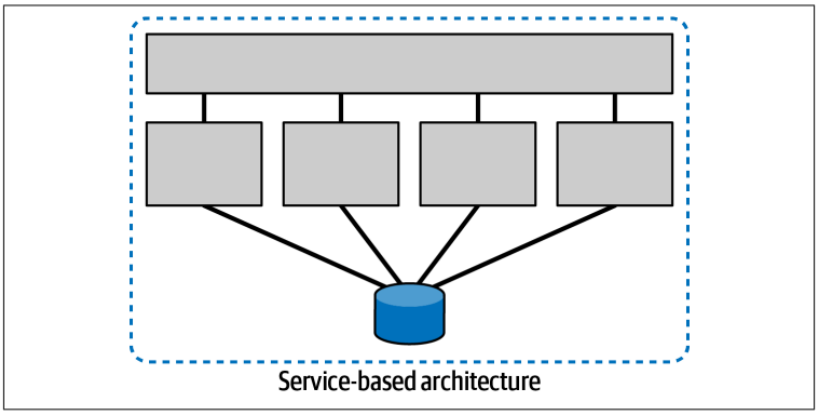

Figure 2: Represented that the Architecture Quantum for a Service-based Architecture.

While this individual services model shows the isolation common in microservices, the architecture still utilizes a single relational database, rendering its architecture quantum score to one.

So far, the static coupling measurement of architecture quantum has evaluated all the topologies to one. However, distributed architectures create the possibility of multiple quanta but don't necessarily guarantee it. For example, the mediator style of event driven architecture will always be evaluated to a single architecture quantum, as illustrated in Figure 3. Even though this style represents a distributed architecture, two coupling points push it toward a single architecture quantum: the database, as common with the previous monolithic architectures, but also the Request Orchestrator itself any holistic coupling point necessary for the architecture to function forms an architecture quantum around it.

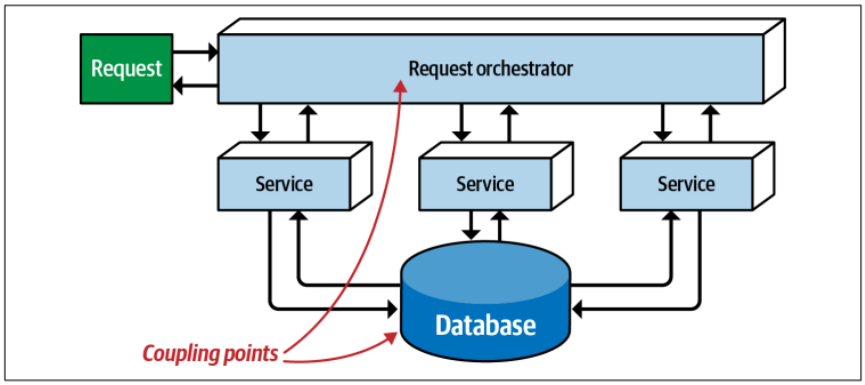

Figure 3: Displayed that the A Mediated EDA has a Single Architecture Quantum.

Broker event-driven architectures without a central mediator are less coupled, but that doesn't guarantee complete decoupling. This broker-style event driven architecture without a central mediator is nevertheless a single architecture quantum because all the services utilize a single relational database, which acts as a common coupling point. The question answered by the static analysis for an architecture quantum is, "Is this dependent of the architecture necessary to bootstrap this service?" Even in the case of an event-driven architecture where some of the services don't access the database, if they rely on services that do access the database, then they become part of the static coupling of the architecture quantum.

However, what about situations in distributed architectures where common coupling points don't exist? Consider the event-driven architecture. The architects designed this event-driven system with two data stores, and no static dependencies between the sets of services. Note

that either architecture quantum can run in a production-like ecosystem. It may not be able to participate in all workflows required by the system, but it runs successfully and operates sends requests and receives them within the architecture. The static coupling measure of an architecture quantum assesses the coupling dependencies between architectural and operational components. Thus, the operating sys tem, data store, message broker, container orchestration, and all other operational dependencies form the static coupling points of an architecture quantum, using the strictest possible contracts, operational dependencies.

The micro services architecture style features highly decoupled services, including data dependencies. Architects in these architectures favour high degrees of decoupling and take care not to create coupling points between services, allowing each individual service to each form its own quanta as display in Figure 4.

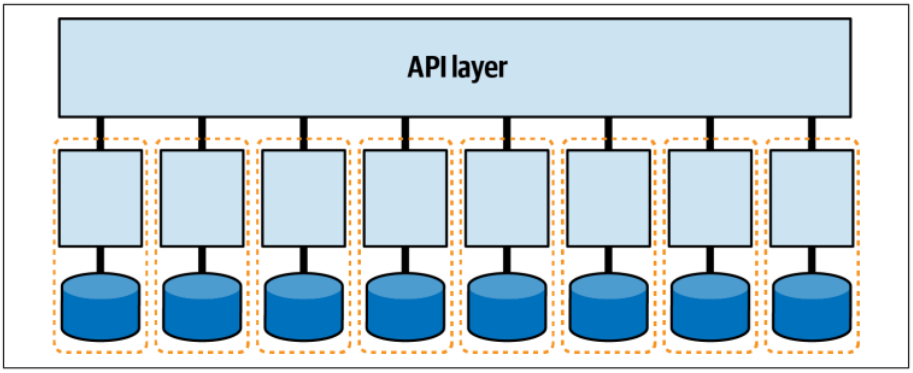

Figure 4: Represented that the Micro services may form their Own Quanta.

Each service acting as a bounded context may have its own set of architecture characteristics one service might have higher levels of scalability or security than another. This granular level of architecture characteristics scoping represents one of the advantages of the micro services architecture style. High degrees of decoupling allow teams working on a service to move as quickly as possible, without worrying about breaking other dependencies.

User interfaces create coupling points between the front and back end, and most user interfaces won't operate if portions of the backend aren't available. Additionally, it will be difficult for an architect to design different levels of operational architecture characteristics (performance, scale, elasticity, reliability, and so on for each service if they all must cooperate together in a single user. Architects design user interfaces utilizing asynchronicity that doesn't create coupling between front and back. A trend on many micro services projects is to use a micro frontend framework for user interface elements in a microservices architecture. In such an architecture, the user interface elements that interact on behalf of the services are emitted from the services themselves. The user interface surface acts as a canvas where the user interface elements can appear, and also facilitates loosely coupled communication between components, typically using events.

Dynamic Quantum Coupling

The last portion of the architecture quantum definition concerns synchronous cou pling at runtime in other words, the behaviour of architecture quanta as they interact with one another to form workflows within a distributed architecture.

The nature of how services call one another creates difficult trade-off decisions because it represents a multidimensional decision space, influenced by three interlocking forces:

D. Communication refers to the type of connection synchronicity used: synchronous or asynchronous.

E. Consistency describes whether the workflow communication requires atomicity or can utilize eventual consistency.

F. Coordination describes whether the workflow utilizes an orchestrator or whether the services communicate via choreography.

Communication

When two services communicate with each other, one of the fundamental questions for an architect is whether that communication should be synchronous or asynchronous. Synchronous communication requires the requestor to wait for the response from the receiver.

The calling service makes a call using one of a number of protocols that support synchronous calls, such as gRPC and blocks does no further processing until the receiver returns a value or status indicating a state change or error condition. Asynchronous communication occurs between two services when the caller posts a message to the receiver (usually via a mechanism such as a message queue) and, once the caller gets acknowledgment that the message will be processed, it returns to work. If the request required a response value, the receiver can use a reply queue to asynchronously notify the caller of the result.

The caller posts a message to a message queue and continues processing until notified by the receiver that the requested information is available via return call. Generally, architects use message queues to implement asynchronous communication, but queues are common and create noise on diagrams, so many architects leave them off, as shown in the lower diagram. And, of course, architects can implement asynchronous communication without message queues by using a variety of libraries or frameworks. Each diagram variety implies asynchronous messaging; the second provides visual shorthand and less implementation detail.

Architects must consider significant trade-offs when choosing how services will com□ municate. Decisions around communication affect synchronization, error handling, transnationality, scalability, and performance. The remainder of this book delves into many of these issues.

i. Consistency

Consistency refers to the strictness of transactional integrity that communication calls must adhere to. Atomic transactions all-or-nothing transactions requiring consistency during the processing of a request) lie on one side of the spectrum, whereas different degrees of eventual consistency lie on the other side. Transnationality having several services participate in an all-or-nothing transaction is one of the most difficult problems to model in distributed architectures, resulting in the general advice to try to avoid cross-service transactions. We discuss consis□ tency and the intersection of data and architecture.

ii. Coordination

Coordination refers to how much coordination the workflow modelled by the communication requires. Simple workflows a single service replying to a request don't require special consideration from this dimension. However, as the complexity of the workflow grows, the greater the need for coordination. These three factors communication, consistency, and coordination all inform the important decision an architect must make. Critically, however,

architects cannot make these choices in isolation; each option has a gravitation effect on the others. For example, trans actionality is easier in synchronous architectures with mediation, whereas higher levels of scale are possible with eventually consistent asynchronous choreographed systems.

DISCUSSION

A variety of academic studies argue that a relationship exists between the structure of an organization and the design of the products that this organization produces. Specifically, products tend to "mirror" the architectures of the organizations in which they are developed. This dynamic occurs because the organization's governance structures, problem solving routines and communication patterns constrain the space in which it searches for new solutions. Such a relationship is important, given that product architecture has been shown to be an important predictor of product performance, product variety, process flexibility and even the path of industry evolution. We explore this relationship in the software industry. Our research takes advantage of a natural experiment, in that we observe products that fulfill the same function being developed by very different organizational forms [10] . At one extreme are commercial software firms, in which the organizational participants are tightly-coupled, with respect to their goals, structure and behavior. The mirroring hypothesis predicts that these different organizational forms will produce products with distinctly different architectures. Specifically, loosely-coupled organizations will develop more modular designs than tightly-coupled organizations. We test this hypothesis, using a sample of matched-pair products. We find strong evidence to support the mirroring hypothesis. In all of the pairs we examine, the product developed by the loosely-coupled organization is significantly more modular than the product from the tightly-coupled organization. We measure modularity by capturing the level of coupling between a product's components. The magnitude of the differences is substantial—up to a factor of six, in terms of the potential for a design change in one component to propagate to others. Our results have significant managerial implications, in highlighting the impact of organizational design decisions on the technical structure of the artifacts that these organizations subsequently develop.

CONCLUSION

This study makes an important contribution to the academy and practicing managers. We find strong evidence to support the hypothesis that a product's architecture tends to mirror the structure of the organization in which it is developed. In all the pairs we examine, the loosely-coupled organization develops a product with a more modular design than that of the tightly-coupled organization. Furthermore, the open-source software product with the highest propagation cost comes from an organization that is more tightly-coupled than the typical open-source project. The results have important implications, in that we show a product's architecture is not wholly determined by function, butis influenced by contextual factors [11], [12] . The search for a new design is constrained by the nature of the organization within which this search occurs. The differences in levels of modularity within each product pair are surprisingly large, especially given each matches products of similar size and function. We find products vary by up to a factor of six, in terms of the potential for a design change in one component to propagate to other system components. Critically, these differences are not driven by differences in the number of direct dependencies between components–in only three of the pairs does the tightly-coupled organization produce a design with significantly higher density. Rather, each direct dependency gives rise to many more indirect dependencies in products developed by tightly-coupled organizations, as compared to those developed by loosely-coupled organizations. The mirroring phenomenon is consistent with two rival causal mechanisms. First, designs may evolve to reflect their development environments. In tightly-

coupled organizations, dedicated teams employed by a single firm and located at a single site develop the design. Problems are solved by face-to-face interaction, and performance "tweaked" by taking advantage of the access that module developers have to information and solutions developed in other modules. Even if not an explicit managerial choice, the design naturally becomes more tightly-coupled. By contrast, in loosely coupled organizations, a large, distributed team of volunteers develops the design. Face-to-face communications are rare given most developers never meet. Hence fewer connections between modules are established. The architecture that evolves is more modular as a result of the limitations on communication between developers.

REFERENCES:

[1] N. Medvidovic and R. N. Taylor, "A classification and comparison framework for software architecture description languages," *IEEE Trans. Softw. Eng.*, 2000, doi: 10.1109/32.825767.

[2] E. M. Dashofy, A. Van Der Hoek, and R. N. Taylor, "A comprehensive approach for the development of modular software architecture description languages," *ACM Trans. Softw. Eng. Methodol.*, 2005, doi: 10.1145/1061254.1061258.

[3] J. Li, N. T. Pilkington, F. Xie, and Q. Liu, "Embedded architecture description language," *J. Syst. Softw.*, 2010, doi: 10.1016/j.jss.2009.09.043.

[4] S. Strohmeier and F. Gross, "Designing an architecture description language for nontechnical actors and purposes (NOTE-ADL)," *J. Enterp. Inf. Manag.*, 2020, doi: 10.1108/JEIM-06-2018-0120.

[5] T. Papapostolu, "μσADL: An architecture description language for MicroServices," in *Advances in Intelligent Systems and Computing*, 2020. doi: 10.1007/978-3-030-25629-6_138.

[6] F. D. McKenzie, M. D. Petty, and Q. Xu, "Usefulness of software architecture description languages for modeling and analysis of federates and federation architectures," *Simulation*, 2004, doi: 10.1177/0037549704050185.

[7] R. Kassem, M. Briday, J. L. Béchennec, G. Savaton, and Y. Trinquet, "Harmless, a hardware architecture description language dedicated to real-time embedded system simulation," *J. Syst. Archit.*, 2012, doi: 10.1016/j.sysarc.2012.05.001.

[8] M. Guessi, E. Cavalcante, and L. B. R. Oliveira, "Characterizing Architecture Description Languages for Software-Intensive Systems-of-Systems," in *Proceedings - 3rd International Workshop on Software Engineering for Systems-of-Systems, SESoS 2015*, 2015. doi: 10.1109/SESoS.2015.10.

[9] F. Oquendo, "π-ADL: an Architecture Description Language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures," *ACM SIGSOFT Softw. Eng. Notes*, 2004, doi: 10.1145/986710.986728.

[10] N. Medvidovic and R. N. Taylor, "A framework for classifying and comparing architecture description languages," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1997. doi: 10.1007/3-540-63531-9_7.

[11] J. O. Filho, S. Masekowsky, T. Schweizer, and W. Rosenstiel, "CGADL: An architecture description language for coarse-grained reconfigurable arrays," *IEEE Trans. Very Large Scale Integr. Syst.*, 2009, doi: 10.1109/TVLSI.2008.2002429.

[12] E. Woods and R. Bashroush, "Modelling large-scale information systems using ADLs - An industrial experience report," *J. Syst. Softw.*, 2015, doi: 10.1016/j.jss.2014.09.018.