OULU UNIVERSITY OF APPLIED SCIENCES

Atte Hautamäki

DESIGNING SOFTWARE ARCHITECTURE FOR A TEST AUTOMATION SYSTEM

DESIGNING SOFTWARE ARCHITECTURE FOR A TEST AUTOMATION SYSTEM

Atte Hautamäki
Bachelor's Thesis
Spring 2020
Department of Information Technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Department of Information Technology, Software Development

Author: Atte Hautamäki
Title of the bachelor's thesis: Designing Software Architecture for a Test Automation System
Supervisor: Timo Vainio
Term and year of completion: Autumn 2020
Number of pages: 32

Software architecture has a massive effect on any software project's success. Early architectural decisions have a huge importance on the rest of the project, because making architectural changes later may be very difficult to accomplish. Software projects with inadequate software architecture tend to be slow to develop and eventually may even eventually fail. Therefore, it is very important from to use adequate resources on creating good software architecture.

The objective of this thesis was to design and document software architecture for a particular test automation system at Nokia. A prototype of the system existed before starting the thesis and it was used as a base for the design process.

This thesis briefly explains what software architecture is and why it is important. Then it continues to explain the process of designing and documenting software architecture for a particular test automation system. Details of the system are not elaborated, but the process is.

As a result of this thesis, reliable and well-documented software architecture for the project was created. The results can be utilized in the further development of the system.

Keywords: software development, software architecture, test automation

PREFACE

The thesis was done in Oulu mostly between May and June 2020. It was commisioned by Nokia as a part of a test automation project, to design and document software architecture for the said system.
Special thanks to Miika Tuuli and Ville Porthan, for their support and guidance during the process. Also special thanks to many individuals in the Oscar project at Nokia, who responded to many difficult questions swiftly and clearly.

Oulu, Finland, 2020
Atte Hautamäki

CONTENTS

VOCABULARY

AI	Artificial Intelligence
ASR	Architecturally Significant Requirements
DUT	Device Under Test
R&D	Research and Development

1 INTRODUCTION

The potential of automatable activities has been increasing in the 21st century at a very rapid rate, mainly thanks to many technological leaps taken in cloudification, robotics, and artificial intelligence.

Automation increases productivity, which is a key driver of economic growth and changes in living standards [1]. Therefore, the potential of automation should be harnessed wherever a valid justification for it can be made.

The power of automation is becoming more and more feasible for an increasing amount of solutions every day, because supporting tools, infrastructure, and understanding continue to improve at a rapid rate.

Despite the potential of automation already beign high, and despite it offers solutions to many problems, the potential is barely utilized, most likely because there is a lack of awareness, but also because it is often difficult to harness. Automation projects tend to be very software-centric, in which the projects' software architecture has an integral role in the projects' success.

The primary aim of the thesis was to design and document software architecture for a test automation system at Nokia. The system is supposed to automate parts of testing hardware equipment in a highly variable environment, where existing solutions or tools cannot be used. The results of this thesis can be will be utilized in developing the system further.

2 TELECOMMUNICATIONS EQUIPMENT AND TESTING

Telecommunications equipment refers to hardware products, such as base transceiver stations, that are used for telecommunications purposes. [2].

Telecommunication equipment has a vital role in modern societies, where everything and everyone are connected. Societies rely on these devices working perfectly around the clock, which, essentially, enable modern communication. The manufacturers of these products possess massive amounts of responsibility. It is essential for all societies that the key products in our infrastructures work flawlessly and are secure to use. To verify the flawlessness of these products, the devices need to be tested appropriately.

2.1 Hardware testing and verification

Testing the products is done in numerous different ways. The test automation system in the subject will focus on testing hardware products in research and development (R&D).

Hardware testing is one aspect of testing, which essentially focuses on ensuring that the system and every component in it are operating as they should and that the system is performing exactly by the requirements [3]. This is an essential part of the product's quality validation and verification, which occurs at various phases during the lifecycle of a product.

Testing as early as possible is very beneficial because the earlier a fault is found in a product, the faster and cheaper it is to correct, this ultimately decreases R&D and manufacturing costs, which in turn improve the products' time-to-market.

2.1.1 Automating Hardware Testing

Automating hardware testing in R&D is becoming increasingly important, partly because the market demands more, which means that products must be

developed faster. While the products must be developed faster, the products must also have more features and be more performant, which among other things means that the devices are becoming increasingly complex. This in turn means that more matters need to be tested, and thus, ultimately, testing the products without harnessing the full power of automation could become a burden that is unsustainable for the business. [4].

While automation is challenging in an environment where variability is high, automation means and understanding have increased a lot in recent years. Therefore test automation at a comprehensive scale in hardware R&D is very feasible and should be employed as soon as possible. [4].

3 TEST AUTOMATION SYSTEM

3.1 Project Purpose

Because manual testing is slow and time-consuming and the means for automated testing exists, testing products manually, even in R&D will likely be unsustainable in the future. Because competition becomes tougher all the time, and products are becoming increasingly complex, thus, test automation and automation, in general, should be employed wherever possible. [4].

The main purpose is to create a test automation system for hardware R&D that will be used in testing and verifying hardware products in development. The system is expected to increase testing coverage and decrease the time required for testing, which will give the hardware designers more time to do other important things. [4].

3.2 Challenges

Due to the complexity and variability in the hardware R&D testing processes, creating a high-scale automation system has been very challenging in the past. However, thanks to the advances in automation technologies in recent years, it is possible and feasible to automate many time-consuming sections of the testing processes completely. This is also why the system is planned to be developed and taken into use in parts, so among other things, the benefits of the system can be seen as early as possible. [4].

Testing in hardware in R&D is difficult. The environment is constantly changing and the products are very complex. Therefore, testing requires a person who is very familiar and experienced with the products being tested and the whole testing environment.

The tests are performed according to test plans, which have been designed and created before the testing phase begins. The test plans, among other things, state what should be tested and why. However, due to the nature of the

environment, where variability is high, the tests can be rarely performed in the same way as they had been done previously. This partly makes it challenging to perform tests and even more challenging to automate them. Nevertheless, flexible automation means exist and can be utilized.

3.3 Automated Testing System

Due to the system being under development, only a summary of the system could be disclosed without the explanation becoming too vague.

In essence, the system is designed to be very flexible and highly modifiable, so that it is possible to meet the requirements in a highly varying environment. The system will utilize a versatile testing software system that is easy to extend and modify, and it will utilize an open-source testing sequencer software.

3.3.1 System Background

As a base, the system will extend and modify an existing automation project created at Nokia for testing products in the production.

While the purpose of the projects is slightly different, it is possible to harness a lot of existing functionality, knowledge, and many other things, which will accelerate and assist the project's development.

Before this thesis had been started, a feasibility study for the system had been concluded and a prototype of the system had been built. The prototype system had demonstrated that automating even parts of the testing process would be extremely feasible and the system would be very useful in the future.

4 INTRODUCTION TO SOFTWARE ARCHITECTURE

4.1 What is software architecture?

Defining software architecture is difficult, primarily because the border between software architecture, design, and other areas of software engineering is very blurry. Nevertheless, there have been multiple attempts to define it. Firtz Solms and others in their research paper called, “What is Software Architecture?” proposed the following definition: “Software architecture is defined as the software infrastructure within which application components providing user functionality can be specified, deployed, and executed” [5].

The currently active software architecting standard, ISO/IEC 42010:2011 – defines architecture as: "The fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution.” [6].

Software architecture is also thought of as a discipline of designing the high-level structures for software systems, structures that meet the technical and operational requirements of a system and affect the system the most. [7].

Fundamentally, software architecture represents a common abstraction that can be used as a basis for mutual understanding of a system. [8].

The difficulty in defining *what* software architecture is is that software architecture often includes more than only designing and producing technical blueprints for a system. The border of software architecture is very blurry and depending on the context, it can go anywhere from choosing programming quirks to system design, to worrying about the economics and managing of a system during its lifetime.

Ultimately, "Architecture is about the important stuff. Whatever that is.", once said author and computer scientist Ralph Johnson. [9].

4.2 What is good software architecture?

Software is a highly malleable entity, meaning that it can be modified easily compared to, for example, a hardware product or to a building. Nevertheless, as the software of a system grows, so do its dependencies and, thus, it tends to become more and more challenging to modify and add new functionality, especially if its architecture is not well designed. Essentially, the better the architecture, the more adaptable the system is for changes and additions.

However, defining what is good architecture is difficult because there is not only one particularly good architecture. After all, all architectures are just products of their context, and good architecture in one environment might not be good in another one. For example, if one considers some websites architecture for just ten years ago. In the past ten years, cloudification, virtualization, and automation in software have taken over. It is a completely different environment. Most likely many good architectures for just ten years ago would not be considered good today because the context and the environment have changed so drastically. [10].

4.2.1 Internal Quality

Good architecture leads to a higher internal quality, which in turn leads to a higher productivity in a team, and vice-versa. A system with a great business potential, but improper software architecture may become unjustifiable quickly. A system with bad architecture would likely display a very high number of bugs and fixing them would be slow and challenging, without any other obvious reasons. On top of this, adding new and modifying existing features would also continue to become slower over time. [9].

Although bugs are a normal part of the software development process, too many of them without any obvious reasons, work as a great warning sign alerting that the systems architecture might not be good enough.

Good architecture, however, comes at a cost and in some cases, it is not viable to invest at all in designing and creating an especially good one. On top of proper, extensive design and documentation, good architecture requires the architectural principles being followed and updated, which in some short-lived projects can be impractical. For example, a one-time marketing product, which after some marketing camping would be ditched, would not require especially good architecture simply because the system does grow great enough to suffer from a bad one.

The larger and longer-term a project is, the more essential it is to have good architecture. This makes perfect sense if you compare software architectures to buildings architectures, which are analogous [11].

5 INTERFACES AND DECOUPLING

This section explains the role of interfaces and briefly touches decoupling and cohesion because they have such an integral role in software architecture.

The definition of the work *interface* in the Oxford Pocket Dictionary of Current English is the following*: "a point where two systems, subjects, organizations, etc., meet and interact:" [12].* In other words, interfaces are communication protocols used all over for interaction purposes.

5.1 Interfaces in Software

Interfaces provide a way to achieve better software architecture, by offering a straight easy to use way to decouple software components. Most object-oriented principle programming languages have interfaces as a part of the language, which, are just: *"a contract between the consumer and provider.". [19].*

The following code snippet example figure 1, written in C# illustrates such a contract. It is not important to understand the syntax, but it is important to understand that interfaces in programming also are just contracts that all participating parties agree to follow.

```
public interface IStandardElectricalSystem
{
    void GetEnergy();
}

public class Phone
{
    public void PluginToWall(IStandardElectricalSystem electricalSystem)
    {
        electricalSystem.GetEnergy();
    }
}

public class SchoolBuilding : IStandardElectricalSystem
{
    public void GetEnergy()
    {
        throw new NotImplementedException();
    }

    public SchoolBuilding()
    {
        new Phone().PluginToWall(this);
    }
}

public class OfficeBuilding : IStandardElectricalSystem
{
    public void GetEnergy()
    {
        throw new NotImplementedException();
    }

    public OfficeBuilding()
    {
        new Phone().PluginToWall(this);
    }
}
```

FIGURE 1. Using an interface in C#

In the above code example, the *Phone* object does not rely on any single object to work. It relies on the *IStandardElectricalSystem* interface. The classes using

the *Phone class* must provide an implementation of the interface. The advantage is that the *Phone* does not need to know of any other classes or objects. Thus, it does not rely on them, meaning that the *Phone* class does not know about the *Building* classes and vice-versa, yet both offer the same functionality to the *Phone* consumer.

5.2 Decoupling and cohesion

Generally speaking, good software architecture has low-cohesion and high decoupling. This essentially means that software components should not rely on other components, or in other words, if some component breaks, others should not be affected. Therefore, it is important to minimize the cohesion between components in a system.

Interfaces provide one good solution for achieving higher decoupling and lower cohesion, which ultimately helps the architect to develop a lower separation of concerns, and that is the key in all good software architectures.

6 REQUIREMENTS

Software system requirements are very important, as the system is built on them, but it is also very important to realize that software requirements tend to change a lot. As mentioned earlier, good software architecture is adaptable for changes. Therefore the architecture must evolve along with new and changing requirements, meaning that software architecture is often an iterative process.

The architecture of the system must evolve with its requirements, but it is also equally important to realize that not all requirements affect the architecture of a system equally [13]. Some requirements are very significant for the architecture of a system, while others are not. The ones which are, are called architecturally significant requirements (ASR).

All system requirements can be divided into two distinct categories. Named ambiguously by IEEE, the categories are called *functional* and *non-functional* requirements. Non-functional requirements are often also called "quality attributes" [14].

6.1.1 FUNCTIONAL REQUIREMENTS

A functional requirement focuses on "*what*" the system should do. A few examples from the system in the thesis:

- The system should be able to perform a test automatically according to given parameters.
- The system shall generate a test report from every test.
- The system should notify the user with the test results

6.1.2 NON-FUNCTIONAL REQUIREMENTS

A non-functional requirement focuses on "*how*" the system should behave. Non-functional requirements specify the system's quality chraracteristics.

A few examples from the test automation system are:

- The system shall be easy to use.
- The tests shall be generic so they can be used for different products.
- The system shall be reliable.
- The system shall be secure.

6.1.3 ARCHITECTURALLY SIGNIFICANT REQUIREMENTS

Architecturally significant requirements are a subset of requirements. Requirements determine and shape software architecture [13]. The architect shall give special attention to architecturally significant requirements because they largely define the architecture and do not tend to change much during the lifetime of a project.

Some common characterizes to ASRs are [13].

- The requirement is associated with high business value and/or technical risk.
- The requirement is a concern of a particularly important stakeholder.
- The requirement has a first-of-a-kind character
- The requirement has caused budget overruns or client dissatisfaction in a previous project with a similar context.

ASRs include the “ilities” of a system. They are something that the architect should be concerned about in the system all the time.

The most notable “ilities” include [14], are shown below (figure 2).

accessibility	accountability	accuracy	adaptability
administrability	affordability	agility	audibility
autonomy	availability	compatibility	composability

configurability	correctness	credibility	customizability
debuggability	degradability	determinability	demonstrability
dependability	deployability	discoverability	distributability
durability	effectiveness	efficiency	evolvability
extensibility	failure transparency	fault-tolerance	fidelity
flexibility	inspectability	installability	integrity
interchangeability	interoperability	learnability	localizability
maintainability	manageability	mobility	modifiability
modularity	observability	operability	orthogonality
portability	precision	predictability	process capabilities
producibility	provability	recoverability	relevance
reliability	repeatability	reproducibility	resilience
responsiveness	reusability	robustness	safety
scalability	seamlessness	self-sustainability	serviceability
securability	simplicity	stability	standards compliance
survivability	sustainability	tailorability	testability
understandability	upgradability	usability	vulnerability

FIGURE 2. Ilities

There are hundreds of quality attributes, some of which the architect should focus on and perform trade-offs. The architect must figure out the most important quality attributes of the system and focus on them. It is also important to realize that most quality attributes also affect other quality attributes, and often in counter to each other. For example, it may be very difficult to create a highly performant and scalable application [10].

All quality attributes could be added to every software project, but the architect should only focus on the most important ones for that particular project. They guide the architectural design and influence architectural decisions during the design process [10].

7 DESIGNING THE ARCHITECTURE

This section briefly explains the design process for the beforementioned test automation system in the subject. However, no details could be shared because the system is still under an active research and development phase.

The system would continue to be developed according to the outcomes of this process.

The main purpose of the process was to:

- Critically re-evaluate the architecture of the existing system
- Re-design what is necessary and design sections of the architecture that had not been completed yet.
- Document the systems architecture clearly and understandably so that it can be used when developing the system further.

7.1.1 Standards and guidelines

During the design phase, the I*SO/IEC/IEEE 42010:2011 Systems and software engineering* was used as a guide. The architecture does not claim conformance to the standard, largely because the prototype had been developed far enough, hence some activities defined in the standard would be less valuable to follow. It is however possible with relative ease to extend and modify the documentation to conform to the standard in the future if found appropriate.

7.2 Architecture Design Methods

The architecting activities largely ensued a software architecting model designed by Christine Hofmeister, Philippe Kruchten, ectara. [15].

The model includes three separate activities throughout which the architecture is created. The activities do not proceed sequentially. The design activities happen iteratively and at different stages of the software development life cycle,

as well as over the evolution of a system [15].

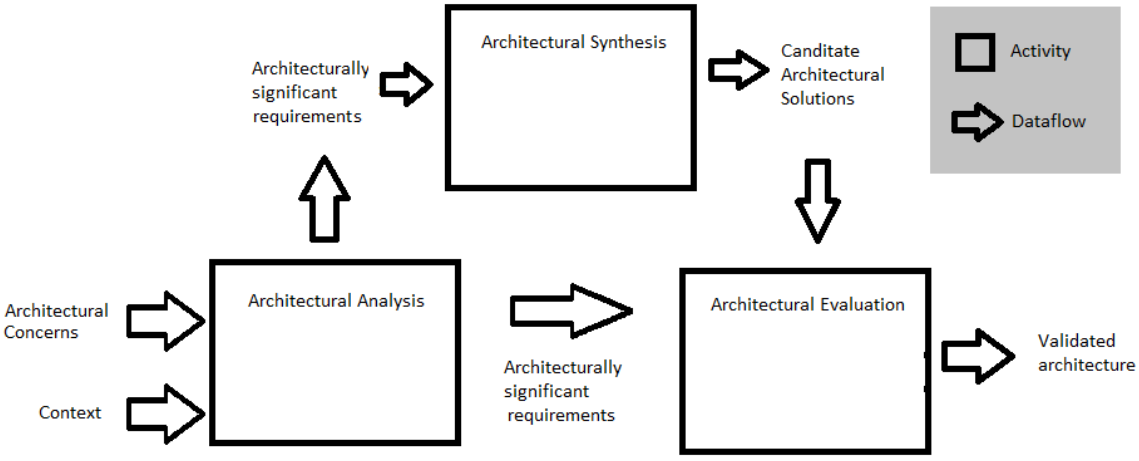

Image 3. Architectural Design Activities

The architectural process followed these activities.

7.2.1 Architectural analysis

The first main activity is to determine the requirements of the system and the environment the system will operate in. The outputs of this process are called architecturally significant requirements. Deciding whether a requirement is architecturally significant is often based on judgment [16].

During the process, the ASRs were straightforward to discover, as the author had been working on existing prototype system before starting the thesis, and therefore, was very well aware of the systems requirements.

The ASRs for the analysis were mostly confirmed from Use Case diagrams, entailing the most important stakeholders' aims. Further, ASRs were confirmed by discussing with the stakeholders about the system and by going through the systems requirements documents. Finally, the ASRs were analyzed and validated during the architectural evaluation activity.

7.3 Architectural synthesis

In this activity, the design is created and improved, based on the discovered architectural requirements by the analysis.

Different types of architectural diagrams of the system were drawn and documented, which depict different aspects of the system and therefore offer the ability to spot flaws in the design. The diagrams were mostly drawn in the UML language.

The following architectural diagrams were drawn and elaborated on. Some included multiple views in different contexts.

- System Context Diagram, which shows **the relationship** between external components and hardware, which in turn helps to focus attention on external factors and events that should be considered [17].
- Behavioural Diagram, which shows how the system behaves and interacts with itself and other entities in **run-time**. It shows the step-by-step activity of the system [18].
- Deployment diagram, which shows the **static view** of the system's run time configuration. In other words, it displays the hardware, and the software running on that hardware, and the middleware connecting them [18].

Additional diagrams of the system, which were included in the architectural documentation, were parts of the system that were complex and needed clarification. These parts were mostly associated with the networking aspects of the system.

7.4 Architectural Evaluation

The purpose of this activity is to determine how well the current design satisfies the discovered ASRs. This activity usually happens after an architectural decision has been made.

During the process, the evaluation mostly happened in the following ways.

- Analyzing architectural diagrams and validating them against the ARSs
- Confirming that the architecture follows common architectural guidelines
- Asking peers to review the architecture after important architectural decisions.

7.5 Architecture Evolution

This activity mostly includes critical supporting activities, such as documentation and architectural decision recording.

As a result of the process, ultimately only three architectural documents were created which were adequate to describe the architecture without causing any maintainability burden. The following documents were created:

- A presentation, which introduced the system and its high-level architecture on a general level to all stakeholders.
- Solution architecture document, which included a brief description of the purpose and scope of the system. The document was much more technical and was aimed at developers. It included all the architectural diagrams with explanations and other architecturally significant details of the system.
- Architectural decisions document, which largely goes in hand with the solution architecture document. It included all the architectural decisions, which were documented using the Alexandrian Form.

7.6 Results of the process

The main purpose of the process was essential to critically evaluate the existing architecture of the system, to improve, re-design, if necessary, and to document it. The architecture of the system was documented together with architectural decisions and reasonings.

The existing prototype had a simple architecture, and a new significantly superior architecture was not discovered that should have been applied to the existing system. There is always a cost of change and the alternative architectures found as a result of the process were not sufficient to satisfy that.

The primary aims were met. The architectural documentation will greatly in developing the system further and make it easier to communicate about the system with others. Finally, now that the architecture has been evaluated and redesigned where necessary, there is a great confidence that it will last.

One worry and issue during the process was: *How to avoid over-documenting and only documenting the relevant information?* In a world where agile methods and agility has taken over, this felt like a very important question to ask.

Documentation quickly becomes old if not maintained, but if some parts of the documentation do not add value to the project, should it be actively maintained? Thankfully, the documented parts of the system stayed very relevant during the thesis. Documenting also the lower-level implementation details would unavoidably have had to be updated often and thus, it might have become a burden that does not necessarily add much value to the project itself. This ,however, largely depends on the scale and complexity of the system. The larger and more complex the system is, the more comprehensive documentation it requires.

8 CONCLUSION

Good software architects are technically very competent, meaning that they have deep pockets filled with the most recent technical information, but also, importantly, they have very good social and communication skills.

The ability to communicate the architecture of a system and its architectural decisions to different stakeholders effectively and convincingly is essential. This means that the architect must see how the project fits in the bigger picture, and see how it will perform in the future.

8.1 Results

The produced documentation certainly made it easier to communicate and reason the architecture of the system and the architectural decisions made to people involved. Additionally, as the software architecture of the project is well documented, tested, it can be trusted to advance with the development.

Slightly surprisingly though, the high-level architecture of the existing system barely changed. There is always a cost of change, and the alternative design options that were considered were not substantial enough. Partly because the system had quite modest architectural requirements, and also because architectures are a product of their context, and the context had not changed.

8.2 Expectations

Initially, I certainly expected the subject to be very challenging and took it with great pleasure as a great learning opportunity to expand my existing skillset. However, I did not expect that creating a clear plan for the architecting process was as challenging and time-demanding as they were.

In summary, software architecture is somewhat ambiguous and is an enormous software engineering topic, and learning even the basics takes time. Furthermore, the industry has been moving towards agile software development

in recent years quickly, and software architecture is in that mix. Therefore, all architecting documentation had to be taken with a grain of salt.

Overall, doing the thesis, while challenging, was very rewarding and has given me a new way of looking at software systems. There is still certainly a lot to learn about software architecting, and software engineering in general, but this gave me a great push towards becoming a better one.

9 REFERENCES

[1] OECD (2010), "Multi-factor productivity (Edition 2013)", OECD Productivity Statistics [Online]. Available: https://doi.org/10.1787/data-00495-en [Accessed: 19-Sep-2020].

[2] “What is Telecommunications Equipment? - Definition from Techopedia,” Techopedia.com. [Online]. Available: https://www.techopedia.com/definition/30400/telecommunications-equipment. [Accessed: 19-Sep-2020].

[3] “Hardware testing,” *Hardware Verification, Testing and Maintenance -*. [Online]. Available: http://aceproject.org/main/english/et/ete05a.htm. [Accessed: 28-Sep-2020].

[4] Employees, Digital Engineers, 2019 – 2020. Oulu: Nokia Solutions and Networks, MN RF RD OUL BB 2 SG. Conversation between June 2019 to September 2020.

[5] F. Solms, “What is software architecture?” Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference on - SAICSIT 12, 2012.

[6] Institute of Electrical and Electronics Engineers, Inc., International Standard 42010:2011(E), Systems and software engineering — Architecture description, ISO/IEC/IEEE, 2011-12-01.

[7] P. Clements, F. Bachmann, L. Bass; D. Garlan; J. Ivers; R. Little; P. Merson; R. Nord; J. Stafford, Documenting Software Architectures: Views and Beyond, Second Edition, Boston: Addison-Wesley, 2010

[8] Bass, Clements and Kazman, Software Architecture in Practice (3rd Edition), Addison-Wesley Professional, 2012.

[9] M. Flower, “Software Architecture Guide,” *martinfowler.com*, 17-May-2017. [Online]. Available: https://martinfowler.com/architecture/. [Accessed: 28-Sep-2020].

[10] N. Ford and M. Richards. "What is Software Architecture?" O’Reilly Media, Inc. [Video file]. Nov. 2017. Available: https://learning.oreilly.com/videos/software-architecture-fundamentals/9781491998991/9781491998991-video316989. [Accessed: 26-Nov-2020].

[11] D. E. Perry and A. L. Wolf, “Foundations for the study of software architecture,” Softw. Eng. Notes, vol. 17, no. 4, pp. 40–52, 1992.

[12] “interface,” Oxfordlearnersdictionaries.com. [Online]. Available: https://www.oxfordlearnersdictionaries.com/definition/english/interface_1. [Accessed: 26-Sep-2020].

[13] Wikipedia contributors, “Architecturally significant requirements,” *Wikipedia, The Free Encyclopedia*, 20-Jun-2020. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Architecturally_significant_requirements&oldid=963521005. [Accessed: 26-Sep-2020].

[14] Wikipedia contributors, “List of system quality attributes,” *Wikipedia, The Free Encyclopedia*, 20-Aug-2020. [Online]. Available: https://en.wikipedia.org/w/index.php?title=List_of_system_quality_attributes&oldid=974015228. [Accessed: 26-Sep-2020].

[15] C. Hofmeister, P. Kruchten, R. L. Nord, H. Obbink, A. Ran, and P. America, “A general model of software architecture design derived from five industrial approaches,” J. Syst. Softw., vol. 80, no. 1, pp. 106–126, 2007.

[16] “Concept: Architecturally significant requirements,” Liu.se. [Online]. Available:

https://www.ida.liu.se/~TDDC88/openup/core.tech.common.extend_supp/guidances/concepts/arch_significant_requirements_1EE5D757.html. [Accessed: 26-Sep-2020].

[17] “Ideal modeling & diagramming tool for Agile team collaboration,” Visual-paradigm.com. [Online]. Available: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/behavior-vs-structural-diagram/). [Accessed: 26-Sep-2020].

[18] “UML 2 Deployment Diagrams: An Agile Introduction,” Agilemodeling.com. [Online]. Available: http://www.agilemodeling.com/artifacts/deploymentDiagram.htm. [Accessed: 26-Sep-2020].

[19] Wikipedia contributors, “Open service interface definitions”, *Wikipedia, The Free Encyclopedia*, 20-Jun-2020. [Online]. Available: https://en.wikipedia.org/wiki/Open_service_interface_definitions. [Accessed: 8-Oct-2020].