Tampereen yliopisto

Ville Nyrhilä

TESTING AND SIMULATION MONITOR FOR TRAIN SYSTEMS

A case study in software engineering and design

"Faculty of Information Technology and Communication Sciences"
"Master's Thesis"
"Examiner: Hannu-Matti Järvinen"
"Examiner: Jukka Saari"
"November 2022"

TIIVISTELMÄ

"Ville Nyrhilä" : "Testing and Simulation Monitor for Train Systems"
Diplomityö
Tampereen yliopisto
"Tietotekniikan DI-ohjelma"
Marraskuu 2022

Nykyaikana junissa on sulautettuna suuri määrä erinäisiä laitteita, jotka tarkkailevat junan tilaa ja sääntelevät sitä erinäisin tavoin. Näiden laitteiden tilan seuraamiseen ja muokkaamiseen tarvittiin oma, erillinen työkalunsa. Kyseisen työkalun tuottaminen ja kuvailu oli tämän työn tavoite. Työkalun nimeksi annettiin ”Tobamon”.

Työn teoreettiseksi kehykseksi otettiin design science, joka on alun perin keksitty jo 1950-luvulla. Se on levinnyt ja tullut sovelletuksi monilla eri tekniikan aloilla. Kasvavissa määrin sitä käytetään myös ohjelmistotuotannon saralla. Design sciencen tarkempi määritelmä lainataan Hevner et co. paperista, jossa luetellaan suunnittelutieteen seitsemän perustavanlaatuista suuntaviivaa: Design as an Artifact, Problem Relevance, Design Evaluation, Research Contributions, Research Rigor, Design as a Search Process, ja Communication of Research.

Työssä esitellään käytetyt teknologiat. Tobamonin perustoiminnallisuus toteutettiin tässä työssä Pythonilla. Nettisivunäkymä, jossa monitoroitujen laitteiden tilat eri ajanhetkinä esitetään kaaviomuodossa, on toteutettu JavaScriptillä ja tämän erinäisillä kirjastoilla. GraphQl-kyselykieltä käytettiin kohdelaitteiden tilojen selvittämiseen. Postman-ohjelmaa käytettiin Tobamonin kehityksen aikana nettisivunäkymän käyttämien API-päätepisteiden testaamiseen.

Tämän jälkeen kuvaillaan alkuperäiset suunnitelmat Tobamonista, sekä sen kehittämisen vaiheita. Moni alun perin suunniteltu ominaisuus jouduttiin jättämään pois, sillä aika ei riittänyt. Käytettyjä työskentelytapoja kuvaillaan.

Kehityksen vaiheiden jälkeen kuvaillaan Tobamonin tässä työssä saavuttama lopullinen muoto. Sellaisenaan se kykenee yhdistämään palvelimeen, joka on junassa tai testipenkissä. Yhteyden kautta se pystyy tarkkailemaan ennalta konfiguroituja oheisia laitteita.

Viimeisenä muodostetaan yhteenveto, arvioidaan Tobamonin kehitystyön sujumista ja tuloksia. Mahdollisia jatkokehitysideoita harkitaan.

Avainsanat: Software Engineering, Design Science, monitori, juna, Teleste

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck –ohjelmalla.

ABSTRACT

"Ville Nyrhilä" : "Testing and Simulation Monitor for Train Systems"
"Master's Thesis"
Tampere University
"Master's Programme in Information Technology"
November 2022

In the modern times there are a lot of devices embedded within trains. These devices monitor the state of the train and modulate and modify it in various ways. A software was needed for monitoring and altering the state of said devices. The development of that tool was the goal of this work. “Tobamon” was given as the name of said software.

Design science was chosen as the theoretical framework for this work. Design science research was originally invented in the 1950s and has since come to be used in many types of engineering. It is increasingly being used in Software Engineering research as well. A more definition of design science is borrowed from a paper by Hevner et al., which introduces seven essential guidelines of design science: Design as an Artifact, Problem Relevance, Design Evaluation, Research Contributions, Re-search Rigor, Design as a Search Process, and Communication of Research.

In this work, the set of technologies used are introduced. The basic functionality of Tobamon was implemented using Python. The web page view, in which the states of monitored devices at given points in time are drawn up in a chart, was implemented with JavaScript and its assorted libraries. The GraphQL query language was used for finding out the states of the machines. Postman was used for testing the API endpoints in use of the web page view during development of Tobamon.

After this, the initial designs, and subsequent stages of the development of Tobamon are described. Many originally intended features had to be left out due to time constraints. Development techniques are described.

Following description of development, the form which Tobamon ultimately took is described in much detail. As it is, Tobamon can connect to a server on the train or test bench and monitor devices on board. The devices to be monitored can be configured in the configuration file of Tobamon.

Lastly, a conclusion is drawn, the development of Tobamon is assessed as well as the result. Potential future topics are considered.

Keywords: Software Engineering, Design Science, monitor, train, Teleste

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

FOREWORD

I wish to give thanks to my immediate family for all their encouragement and support. I wish to thank them for all the ways they tried to urge me to study when I was younger and more given to distractions. I wish to give thanks to my extended family as well. It has become a custom among us to value education and to pursue it, and so it has been for three generations now. I owe it to all of us, that we have developed this culture and these values. For my cherished nephew and niece, I hope you, your generation and all who come after you keep to the values and customs of our clan.

I want to express my deepest gratitude to Teleste for giving me this chance to create this thesis work. I wish to thank all my managers and coworkers for all the help they've provided and all their patience with this work. I will never forget this debt of gratitude I owe the company and apologize for any inconvenience it may have caused.

For the staff of the university in general, I also wish to give many thanks for their hard work and patience. Concerning the latter, I also wish to present an apology, for my overly long stay at the university which surely must have caused some frustration.

In the earliest parts of my studies, I was negligent and had my priorities in all the wrong places. I was very childish. Were it that I had had wisdom already then all these years ago, I would have seen to it that I graduated much sooner. Nevertheless, few go through life without any regrets or failures. I am therefore not bitter, that I previously could not see. I am grateful, that I was able to learn. What's important is that we learn from our errors and improve our ways.

I hope that this experience can serve as an example that youth inevitably give way to wisdom and that the scales of folly will fall off the eyes of those who are willing learn. I have seen in myself a tremendous growth from who I was. None of us is perfect. We all have the capacity to improve.

CONTENTS

1. INTRODUCTION 1
2. THEORETICAL FRAMEWORK 3
2.1 Design Science 3
2.2 Guidelines. 5
2.2.1 Design as an Artifact 7
2.2.2 Problem Relevance 7
2.2.3 Design Evaluation 7
2.2.4 Research Contributions 9
2.2.5 Research Rigor 10
2.2.6 Design as a Search Process 10
2.2.7 Communication of research 11
3. TOOLS AND METHODS USED 13
3.1 Python 13
3.2 JavaScript 13
3.3 Chart.js 14
3.4 GraphQL 14
3.5 Postman 14
3.6 Programming style 15
4. PLANS AND DEVELOPMENT 17
4.1 Initial meetings and provisional designs 17
4.2 Full concept 17
4.3 Automated testing 21
4.4 Auxiliary code 22
4.5 Communicating with device server 23
4.6 API endpoints 24
4.7 Problems in generating a visualization 27
4.7.1 The search for a library 27
4.7.2 Chart.js for development 28
5. IMPLEMENTATION 32
5.1 Tobamon Core 34
5.2 Listener 36
5.3 DataHolder 39
5.4 Visualizer 41
5.4.1 API implementation 41
5.4.2 The web page view 42
6. CONCLUSION 47
6.1 Design Science Assessment 47
6.2 Design and Implementation Assessment 47

6.3 Future Ideas.. 48
6.4 Concluding Words.. 48
REFERENCES.. 49

IMAGES

Image 1: Categories of design artifacts visualized. ... 4
Image 2: Test and Generate Cycle, based on Hevner et al. [13] ... 11
Image 3: Initial Plan of Tobamon ... 18
Image 4: Initial Plan of the Monitor element. ... 18
Image 5: Initial Plan of the Simulator element. ... 20
Image 6: Initial Plan of a Scripting System ... 20
Image 7: A testing strategy combining equivalence partitioning and fuzz testing. ... 22
Image 8: Screenshot of the Postman setup. ... 26
Image 9: Visualizer running with test data. ... 31
Image 10: Typical use environment for Tobamon. ... 32
Image 11: Tobamon Class Diagram. ... 33
Image 12: A 3D render representing a generic test bench setup. ... 34
Image 13: The Core sits in the center and coordinates the other components. ... 35
Image 14: Listener Class Diagram. ... 37
Image 15: Listener workflow. ... 38
Image 16: DataHolder saving data. ... 40
Image 17: Web page view script flow visual representation. ... 43
Image 18: Tobamon running with data from the device. ... 45
Image 19: Tobamon running with more dense data from an actual test device. ... 46

TABLES

Table 1. Design-science research guidelines by Hevner et al. [13] 6
Table 2: Design evaluation methods by Hevner et al. [13] 8
Table 3: Chart drawing libraries for Javascript compared.. 27
Table 4: API endpoints for the Visualizer. 41

ABBREVIATIONS AND MARKINGS

API	Short for “Application Programming Interface”. “*An interface that is defined in terms of a set of functions and procedures, and enables a program to gain access to facilities within an application.*” [8]
ASCII	Short for “American standard code for Information Interchange”. “*A standard character encoding scheme introduced in 1963 and used widely on many machines.*” [8]
PLC	Programmable Logic Controller, especially in the context of Allen-Bradley PLCs.
CIP	Common Industrial Protocol. “*CIP is a media independent protocol using a producer-consumer communication model, and is a strictly object oriented protocol at the upper layers.*” [6]
DOM	Document-Object Model. “*A standard interface for representing XML and HTML documents. The data is parsed into a tree of objects, which a programmer can navigate and manipulate.*” [8]
DSS	Decision Support Systems. Software created for the purpose of gathering and presenting relevant data to managers in an organization.
GUI	Graphical User Interface. ”*An interface between a user and a computer system that makes use of input devices other than the keyboard and presentation techniques other than alphanumeric characters.*” [8]
Github	A very popular providing Git repositories for its users.
HTTP	HyperText Transfer Protocol. “*An application-level protocol with the lightness and speed necessary for distributed collaborative hypermedia information systems.*” [8]
INI	A plain text file format used for configuring and initializing applications. File ending is “.ini”.
IP Address	The address of a device with which it can be connected to using the TCP/IP protocol.
IS	Information Systems. “*The branch of knowledge concerning the purpose, design, uses, and effects of information systems in organizations.*” [13]
ISO	International Organization for Standardization. “*-- the body responsible for all international data-processing standards, and many others.*” [8]
IT	Information Technology. “*Any form of technology, i.e. any equipment or technique, used by people to handle information.*” [8] Typically used to refer to the handling of information by means of computers and software.
JS	JavaScript. The most popular programming language for web pages.
JSON	JavaScript Object Notation. A popular format for storing or serializing objects in programming. Derived from the formation of objects in the JavaScript programming language.
MIT Licence	A popular legal licence for open source software. It is very permissive.
OBA	On-Board API. A Teleste API based on GraphQL.
TAM framework	Short for “Technology Acceptance Model”. A framework “*which predicts and explains why a given IT system might or might not be received by a given target audience.*” [13]
TCP/IP	Transmission Control Protocol/Internet Protocol. “*The obligatory standard to be used by any system connecting to the Internet.*” [8]

Tobamon	Short for “Teleste OBA Monitor”. A software to monitor, record and serve variables of devices on a train.
URL	Uniform Resource Locator. “*The address system used on the Internet, for example, to specify the location of documents in the World Wide Web.*” [8]
UTF-8	8-bit Unicode Transformation Format. “*A method of encoding Unicode codepoints using one-byte unsigned integers*”. [8]
dict	A Dictionary, a Python datastructure with key-value pairs.
flag	A boolean variable ; a kind of variable only capable of having one of two values: ‘true’ or ‘false’.

.

1. INTRODUCTION

This thesis describes a project which was commissioned by Teleste. In the project a software to monitor the state of various devices on a train was developed. Previously this had been achieved with older software, which had over time become impractical and hard to maintain.

Teleste is a company which produces data networks and infrastructure and provides both technology and whole solutions. The company was founded in 1954. It is headquartered in Finland. Teleste does manufacturing in Finland and China. Overall, it is a very large company, with offices in 20 countries. [1]

Teleste provides an integrated product, and their services enable building and running a networked society better. Solutions of Teleste bring secure television and broadband services to homes and public places, particularly to public transport. [1]

Keeping with the spirit of the modern times, trains too have an ever-increasing number of devices doing sundry things during a ride. Developers have a need to monitor and test each of these, which then necessitates tools and software. Here's a translated quote from a manager at Teleste describing the problem, the solution of which is the basis of this work:

"No complete system is available when developing a train system. The customer information system integrates to the other systems on-board the train through various interfaces. Towards this end of developing this component system and testing, it is necessary to be able to monitor and control said interfaces. There are currently in use disparate, different tools. In this [project], the purpose is to develop a single monitoring and simulation tool, which will fulfill the requirements of several projects. In addition, new features will be introduced especially into the monitoring of the interface." [36]

It follows that just as there is a need to monitor devices for testing purposes, there needs to be a means of simulating the subject in action. What use is it for a tester, if he must test a train line on a track that is several hundred kilometers long and with several stops, if he must always start from point zero? This cannot do. For this and other reasons, it is necessary for one to be able to alter the states of the devices.

A name was needed for the software developed in this work, so it came to be designated as “Tobamon”. It comes from the words “Teleste OBA Monitor”. “Tobamon” is also easier to pronounce than a typical alphabet-soup abbreviation. Henceforth in this text, the product will frequently also be called Tobamon, its proper name.

The goal of this work was the development of Tobamon, which is a computer program for monitoring devices on a train. Tobamon is highly configurable. Software Engineering is used as a framework for the development in this work. Additional perspective is borrowed from design science, in which the measure of what can be achieved is investigated by means of pushing the technology.

In Section 2, the intellectual discipline of design science (DS) research is explained in detail. Section 3 describes technologies and techniques used in this project. In Section 4, the development of Tobamon is described. Section 5 documents the form of the complete Tobamon.

2. THEORETICAL FRAMEWORK

The theoretical approach pursued in this paper is software engineering (=SE), specifically through a paradigm known as Design Science (=DS). DS was invented by R. Buckminster Fuller and he introduced it in 1957 at a presentation to the Royal Architectural Society of Canada [11]. By today, design science is well established discipline that has been around for decades and has been applied to many fields of engineering.

In this thesis, a definition of Design Science is borrowed from Hevner et al., who's paper is itself in Information Systems research (=IS). IS is a general field researching the innovation and application of systems of information management [13].

Although Hevner's paper is not in the field of software engineering, it is used here because it is concise, clear, and has been very influential. Design science was chosen because of the nature of the work. What then is DS?

2.1 Design Science

Design science pursues the advancement of engineering knowledge by deliberately trying to innovate. These innovations define ideas, practices, technical capabilities, and products. The purpose of design science is to solve real-world problems. For example, to try to come up with a new design pattern or try an existing design pattern in a novel environment. DS is meant to produce generalized design knowledge, not ad-hoc solutions to specific instances. The goal is to improve information systems, how they can be better analyzed, designed, implemented, managed, and made use of. [9][13]

The principles and methodology of DS have become widely used in many fields of engineering, including mechanical, civil, and architectural engineering. Even though for the longest time DS was not recognized as an applicable paradigm for software engineering, many SE papers de facto do line up with the DS paradigm. [9][25]

Design science was chosen as it is the most appropriate conceptual whole in which to develop a product. One is presented with a problem, for which a software solution is expected to be feasible and effective. Designing and implementing such a solution provides a stimulating and educative undertaking.

Design artifacts are a central concept to design science. IT artifacts can be such things as vocabulary and symbols, grouped together as "constructs". They can be abstractions or representations, grouped together as "models". They can be algorithms or practices,

mutually called as “methods”. IT artifacts can also refer to implemented and prototype systems, known collectively as “instantiations”. As such, instantiations are fundamentally more concrete than the other types of artifacts. Most DS papers in SE are in the category of instantiations. [9][13][25] The division of design artefacts into categories is represented in Image 1.

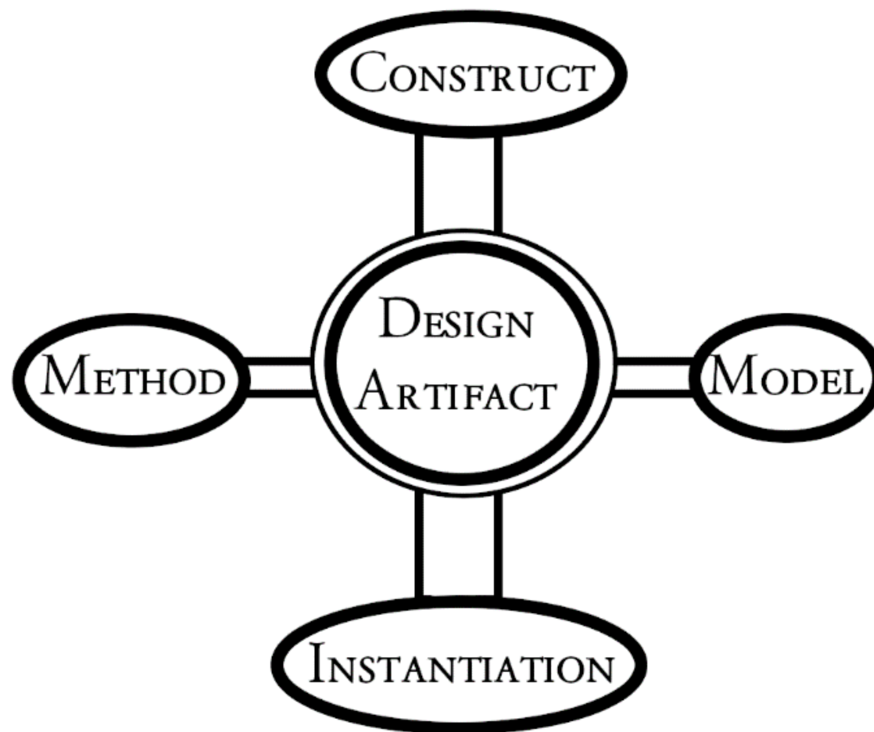

Image 1: Categories of design artifacts visualized.

Design science has been used to identify and handle a great number of design activities. Many have been reduced to routine, but some are such things that are known as “wicked problems”. Enumerated among these are:

“* unstable requirements and constraints based upon ill-defined environmental contexts

* complex interactions among subcomponents of the problem and its solution

* inherent flexibility to change design processes as well as design artifacts (i.e., malleable processes and artifacts)

* a critical dependence upon human cognitive abilities (e.g., creativity) to produce effective solutions

* a critical dependence upon human social abilities (e.g., teamwork) to produce effective solutions”. [13]

Hevner et co. use the term “wicked problem”, but do not explain what it means, apparently assuming it common knowledge. Although the meaning of the term might be intuitively obvious, for the benefit of the reader, an explanation will be provided here.

The term “wicked problem” was coined by Horst Rittel and Melvin Webber in their 1973 paper: “Dilemmas in a General Theory of Planning”. The text is long, ponderous and contains many interesting contemplations on the state of the world in 1973. [32] For the benefit of the reader, an essential points’ list of the traits of wicked problems is provided here:

1. “There is no definitive formulation of a wicked problem”
2. “Wicked problems have no stopping rule”
3. “Solutions to wicked problems are not true-or-false, but good-or-bad”
4. “There is no immediate and no ultimate test of a solution to a wicked problem”
5. “Every solution to a wicked problem is a ‘one-shot operation’; because there is no opportunity to learn by trial-and-error, every attempt counts significantly”
6. “Wicked problems do not have an enumerable (or an exhaustively describable) set of potential solutions, nor is there a well-described set of permissible operations that may be incorporated into the plan”
7. “Every wicked problem is essentially unique”
8. “Every wicked problem can be considered to be a symptom of another problem”
9. “The existence of a discrepancy representing a wicked problem can be explained in numerous ways. The choice of explanation determines the nature of the problem’s solution”
10. “The planner has not right to be right or wrong”. [32]

To boil the points down: one can never grasp the true form of the problem. One can never know if one has successfully dealt with the problem or not. One does not get to practice first. There are no do-overs, and one are stuck with the errors of the solution for life. This is not unlike how it has been noted that design knowledge holistic, which means the full character and context of neither the problem nor solution can ever be known, and unseeable contextual factors are always present [9].

2.2 Guidelines.

In their 2004 paper Hevner et al. provided a description of design science in seven guidelines [13]. It provides an outline for design science research, which has from then on

gone to influence the application of DS in other fields of technology. A reproduction of the outline is provided in the form of Table 1.

Table 1. Design-science research guidelines by Hevner et al. [13]

Design-Science Research Guidelines	
Guideline	Description
Guideline 1: Design as an artifact	Design-science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation.
Guideline 2: Problem Relevance	The objective of design-science research is to develop technology-based solutions to important and relevant business problems.
Guideline 3: Design Evaluation	The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods.
Guideline 4: Research Contributions	Effective design-science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies.
Guideline 5: Research Rigor	Design-science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact.
Guideline 6: Design as a Search Process	The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment.
Guideline 7: Communication of Research	Design-science research must be presented effectively both to technology-oriented as well as management-oriented audiences.

Of these seven, Engström et al. considered relevance, rigor, and novelty as the most important for assessing DS in SE [9]. In the sections below, the guidelines are described in further detail, each under their own heading.

2.2.1 Design as an Artifact

Design science must produce an artifact. The artifact must tackle a specific, clearly defined problem. The artifact must be documented extensively, succinctly but unambiguously. Without sufficient description it will not be possible to make use of it in practical applications. [9][13]

Artifacts themselves are rarely complete products. Rather, they are innovations on ideas, practices, etc. History is replete with examples of artifacts enabling further innovations or completely unprecedented paths of development. [13]

Do other, comparable rules exist that ought to be considered in a similar situation? This refers to the aspect of novelty, which important for DS applied to software engineering. Artifact instantiation can demonstrate that a novel path for innovation exists, where before such could only have been a matter of speculation. [9][13]

2.2.2 Problem Relevance

For the research, the goal is to solve business problems by seeking out knowledge and information that make solving said problems possible. This goal is prosecuted by means of attempting to solve the problems by means of inventing new artifacts. Problems could well be defined in a formalized manner as the difference between the initial situation and the desired end state. [13]

The prosecution of design science research is relevant chiefly to a community of concerned parties. The research must necessarily concern itself with the problems and opportunities afforded by the interaction of people, organizations, and IT. [13] In SE, the relevance ought to be determined from the perspective of the intended user of the artifact and from the perspective of the research community. [9]

2.2.3 Design Evaluation

The efficacy, utility and quality of an artifact must be evaluated rigorously, and the means of evaluation must be performed well [13]. No product, no design, no idea is worth any faith unless its value can be demonstrated. A demonstration is only as good as its rigor. Not all means of evaluation are made equal. Good evaluation methods are well thought-out, and their execution is carried out with discipline. Only good evaluation can discern true quality.

Evaluation is the process of ascertaining a thing's importance, quality, or value [25]. Evaluation must be based on needs imposed by the business environment. The appropriate metrics need to be defined well. The appropriate data may also need to be collected and analyzed. Design is an iterative process, where the different phases of design and development feed into each other. [13]

Mijač proposed a set of 7 guidelines for selecting a means of evaluating artifact instantions. The guidelines were synthesized from DS literature:

1. Use established frameworks for design science research.
2. Use existing frameworks for design of evaluation.
3. Consider evaluating commonly evaluated artifact properties when designing evaluation.
4. Consider commonly used evaluation methods when designing evaluation.
5. Consider commonly used evaluation compositional styles when designing evaluation.
6. Use appropriate frameworks for performing particular evaluation methods.
7. Consider using established software quality models and metrics to evaluate instantiations. [25]

The means usable for evaluation must inevitably be drawn from what is known. These are summarized in Table 2. Despite a rigorous methodology existing, style and artistic expression are inevitably a part of design. The assessment of the quality of style is however a subjective matter. [13]

Table 2: Design evaluation methods by Hevner et al. [13]

Design Evaluation Methods	
1. Observational	Case study: Study artifact in depth in business environment.
	Field study: Monitor use of artifact in multiple projects.
2. Analytical	Static Analysis: Examine structure of artifact for static qualities (e.g., complexity).
	Architecture Analysis: Study fit of artifact into technical IS structure.
	Optimization: Demonstrate inherent optimal properties of artifact or provide optimality bounds on artifact behaviour.

	Dynamic Analysis: Study artifact in use for dynamic qualities (e.g., performance).
3. Experimental	Controlled Experiment: Study artifact in controlled environment for qualities (e.g., usability).
	Simulation – Execute artifact with artificial data.
4. Testing	Functional (Black Box) Testing: Execute artificial interfaces to discover failures and identify defects.
	Functional (White Box) Testing: Perform coverage testing of some metric (e.g., execution paths) in the artifact implementation.
5. Descriptive	Informed Argument: Use information from the knowledge base (e.g., relevant search) to build a convincing argument for the artifact's utility.
	Scenarios: Construct detailed scenarios around the artifact to demonstrate its utility.

2.2.4 Research Contributions

The research conducted must provide new value to the discipline of design science. One would think the fruit of the research need not strictly be a novel innovation. While review of the existing body of knowledge is another important linchpin of science, the accent in design science should be on innovation. [9][13]

The broad categories of innovations in design science are of three varieties:

- Design Artifacts, solutions to previously unsolved problems. Examples include prototype systems.
- Foundations, which refers to more abstract artifacts, such as constructs. Examples include modeling formalisms.

- Methodologies. The innovative use and development of means of evaluation. [13]

The criteria for evaluating whether research has contributed to design science depend on “representational fidelity and implementability”. Artifacts must be faithful representations of the actual business environments used in the research.

Hevner et co. differentiated design research from routine design. Routine design is when puts existing knowledge to use in organizational problems. The contrast to DS is that the latter attempts to solve previously unsolved problems in innovative or unique ways, or previously solved problems in novel and more efficient ways. One innovates and the other applies previous innovations. [13]

2.2.5 Research Rigor

Research conducted haphazardly will yield fruit of no value. Only by taking on sufficient discipline and precision may true findings be uncovered. Likewise, only with sufficient care can the attained knowledge be appraised. This is the nature of things. [13]

Mathematical formalizations are often depended upon in design science, though the IT environment does not always lend itself to such. Excessively over-eager as well as under-eager formalization may occur. Research rigor comes from the proper and effective use of the foundations of the IS knowledge base and research methodologies. [13] For SE, rigor refers to which extent the research is built on prior, existing design knowledge and the consideration of alternative solutions. [9]

Metrics are essential for the evaluation of artifacts. The true worth of artifacts in human-machine interaction needs to be appraised with behavioral science research. Appropriate means must be used for appropriate ends. [13]

2.2.6 Design as a Search Process

Design is typically an iterative process, with trial-and-error involved. As one comes up with an initial plan, the next natural step is to start implementing it. This is the initial plan and the initial product. Initial designs are often imprecise and need to be sharpened repeatedly to reach a better design. [13][25] It is often easier to improve on an existing design than to spawn an immaculate design fully formed as though Athena from Zeus’ forehead. To quote Larry Niven:

“Everybody talks first draft.” [23]

The repetition of design processes becomes something of a virtuous cycle. As designs reach an acceptable level of confidence, they can be put to test. Testing will reveal inadequacies, which will give direction for further, improved design. This cyclical process of design is illustrated in Image 2.

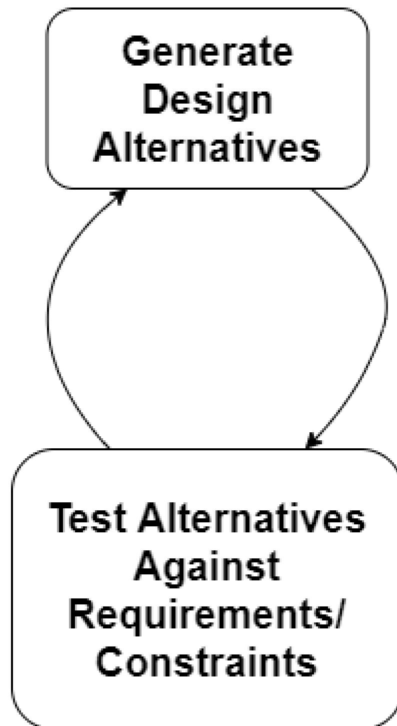

Image 2: Test and Generate Cycle, based on Hevner et al. [13]

Design can be characterized as the search for the right means to produce an effective solution to a problem. Research in design science often start with a simplified subset of of the pertinent means and ends. The other way is to break the problem down into its smaller constituent problems. Those means which can achieve the desired end state make up the set of possible design solutions. [13]

However, design presents many wicked problems for which no definitively correct solution may possibly be described. Under such circumstances, instead of perfect solutions, it is adequate and necessary to go for satisfactory solutions. Instead of describing the complete set of possible designs, describe at least subset of that set. The measure of a good design solution is not always a given, and several means of evaluating it exist. [13]

2.2.7 Communication of research

The research of design science needs to be presented to both technology-oriented and management-oriented audiences. Technology-oriented audiences need to be able to im-

plement and use the artifact in the appropriate business environment. Management-oriented audiences need information to gauge whether their organization ought to commit resources to the procurement of the artifact for use. [13]

3. TOOLS AND METHODS USED

An assortment of technologies, otherwise known as a ‘tech stack’, was used to realize Tobamon. These are described in this section.

3.1 Python

Python is a popular and easy-to-use programming language. It is increasingly used as an educational language to teach students the basics of programming. This in turn has boosted its prominence in the industry, as so many students come out capable of using it. Development on Python is frequently very rapid due to its human-intuitive grammar and features. [30]

There also exists a rich and mature ecosystem of extra modules and libraries for it, accessible through ‘pip’, Python’s own package manager. Pip can be used to download non-standard Python libraries from the online repository of Pip. All the standard library of Python, conversely, comes prepackaged with Python itself. [30]

The reasons Python was chosen as the back-end language of this project are the author’s own familiarity with the language, and the fact that it was already being used to some extent in Teleste. The notable maturity and great selection of libraries of the language were also factors.

3.2 JavaScript

JavaScript, often shortened to ‘JS’ or ‘js’, has been for the longest time the one and only programming language of the Internet. It is in JavaScript that the behavior of websites is programmed in. [15]

This language also has a very robust and mature third-party library ecosystem. Packages can be imported in several ways, though principally either by a direct call to a hosting server, such those provided by CDON or Google, or by downloading them using ‘npm’, a command-line package manager. In the latter means, the developer must serve these libraries from their own server for their web page. [15][24]

In recent times some of the more performance-intensive tasks of JavaScript have been taken over by WebAssembly, a likewise browser-based programming language which is more efficient thanks to the nature of its implementation. It is a compilation target for languages such as C/C++ and Rust, etc. [38]

WebAssembly is largely not yet used in developing Web UIs. There are not many precedents nor examples for this kind of use. JavaScript on the other hand is frequently used in this way with tutorials and examples aplenty. Therefore the decision to use JavaScript was easy to make.

A somewhat deserved ill reputation exists for JavaScript, as compared to language like Python. JavaScript can be much less intuitive. [7] Nonetheless, as it is inevitably the programming language of choice for web-based user interfaces, it was used for this project as well.

3.3 Chart.js

Chart.js is a third-party module for JavaScript. It is a very popular tool for creating and displaying interactive charts and graphs on a web page. [5] Many alternatives exist, but this one deemed most convenient and most feasible, hence why it was chosen. A more detailed account of Chart.js' choosing for this project can be read in Section 5.4.2.

3.4 GraphQL

GraphQL is a common framework for creating and querying APIs [12]. Such an API was used for some Teleste projects, which is why it also came to be used in Tobamon. It is a very handy and intuitive language for constructing queries and returns data in a JSON format.

The website of GraphQL gives a more detailed description, which will be described here, but not as direct quote. It is a query language for APIs as well as a runtime for fulfilling those queries with data from one's API. Queries resemble the JSON data which they return. Giving the name of a heading or tag returns the data in the exact part of the JSON structure. In this way, GraphQL returns multiple resources with a single request. [12]

3.5 Postman

Postman is an application used for sending specific HTTP requests to chosen URLs. It is frequently used for testing API endpoints during software development. By no means is it the only tool usable to this end. Alternatives such as cUrl exist as well. However, Postman provides an intuitive and easy-to-use graphical user interface out of the box. [28]

Postman is an API platform for building and using APIs. It makes every step of API production and maintenance simpler. Postman includes a comprehensive set of API tools

which help accelerate the API lifecycle, an API repository for storing and working with all of one's API artifacts, workspaces for organizing distinct projects and their APIs, and intelligent features for improving API operations. [28]

3.6 Programming style

All programmers naturally tend to develop habits and personal preferences about the form and shape of their code. In a world where every computer program is only ever used and maintained by its very creator, this would not cause issues.

However, in the real world, projects maintained by a single developer are passed on to another as the first one either retires, is moved to another project, or moves to work for another company. Writing code in a very personal - or worse, a haphazard – style makes it very obtuse to other programmers. This can cause the loss of a significant number of working hours into interpretation and refactoring. In the worst case, the source code may have to be scrapped altogether and a new project will have to started to replace it. These were considerations in the development of Tobamon as well.

The responsible and considerate way to write code is to make use of what are called "style guides". It is not something that strictly speaking influences how the code works during runtime, but rather has everything to do with the human needs of programmers.

Some programming languages have several style guides, recommended by different organizations. This is the case of the C language. Some programming languages don't have an official one, but one or more unofficial ones. This is the case of JavaScript. Lastly, there are such programming languages for which no style guide exists, official or unofficial.

A good explanation on the use and purpose of programming style guides is available on Medium.com [22], written by one Bradley Nice. In the text, he reiterates a lot of what was said above, but also elaborates further. A point from the text that was not noted here as exempli gratia: "*A particular programming style may be different from coding conventions, or even designed around a specific language or even program.*" [22] For a series of articles that go into further detail about how to design a style guide, a recommendation must be given for the writings of one Brennan Angel on pullrequest.com [3].

The style guide for Python is known as "PEP 8" [35]. It is an exhaustive compilation of good style rules for orderly and aesthetic Python code and includes a few small text paragraphs discussing the principles of the style decisions as well. PEP 8 is perpetually open for amendments, though the whole has largely been set for some time.

As the name implies, it is a part of a collection of articles which include suggestions for the language itself or for the style conventions, as well general articles about the design of the Python language. Among these texts, PEP 20 “The Zen of Python” [27] deserves mention. It is very short and is here quoted in whole:

“Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

There should be one-- and preferably only one --obvious way to do it.

Although that way may not be obvious at first unless you're Dutch.

Now is better than never.

Although never is often better than *right* now.

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea -- let's do more of those!” [27]

The rules and suggestions given in the above quote largely boil down to not making the code any more complex than it must be, to not leave things unambiguous. Readability is strongly recommended, even though what one finds readable is a subjective experience.

4. PLANS AND DEVELOPMENT

Before starting out on the project, a broad concept outline was sketched for the product: A program for monitoring and simulating, altering the state of devices on a train. More deliberation would be needed however, as well as additional input for the requirements.

4.1 Initial meetings and provisional designs

At the outset, there was some deliberation to properly assess what it would be necessary for the application to do. A few of ideas were concocted. One such idea was to make use of a Python module called “pycomm3”. It is a library for communicating with Allen-Bradley PLCs using Ethernet/IP. Pycomm3 is a continuation of ‘pycomm’, which was a library for Python 2. [26]

For a long time, using pycomm3 was considered for Tobamon, but the idea was eventually scrapped. It would have been used to communicate with devices that use the CIP protocol. CIP stands for “Common Industrial Protocol”. It is an object-oriented, that is widely used in many industries. [6] Due to time constraints and other concerns, it was decided to limit this project to only make use of GraphQL.

Another, more significant suggestion was to containerize the program, using Docker or perhaps some alternative technology. Containerizing software is frequently a good way to greatly boost the portability of a program, though it is not a cure-all by any means. Eventually this too had to be discarded for time concerns.

4.2 Full concept

With the task was set, it was surmised that initial concept was much too specific and narrow in its scope. It was envisioned as being exclusive to a certain project. It was only upon further deliberation, that the final concept of the product was conceived. The software was to be much more generalized and highly configurable.

From there on, designing, and implementing Tobamon in more detail was very straightforward. What was needed was a modular system, where components could be switched out, recycled, and maintained separately.

The first, full concept of the program was quite extensive in its scope. It included two principal components: a monitor and a simulator, as well as an interface for both to allow

using both through Robot Framework (shortened to RF). RF is a popular testing automation framework for the Python programming language [33]. Configuration files would be used to adapt Tobamon to whichever project. This initial design is illustrated in Image 3.

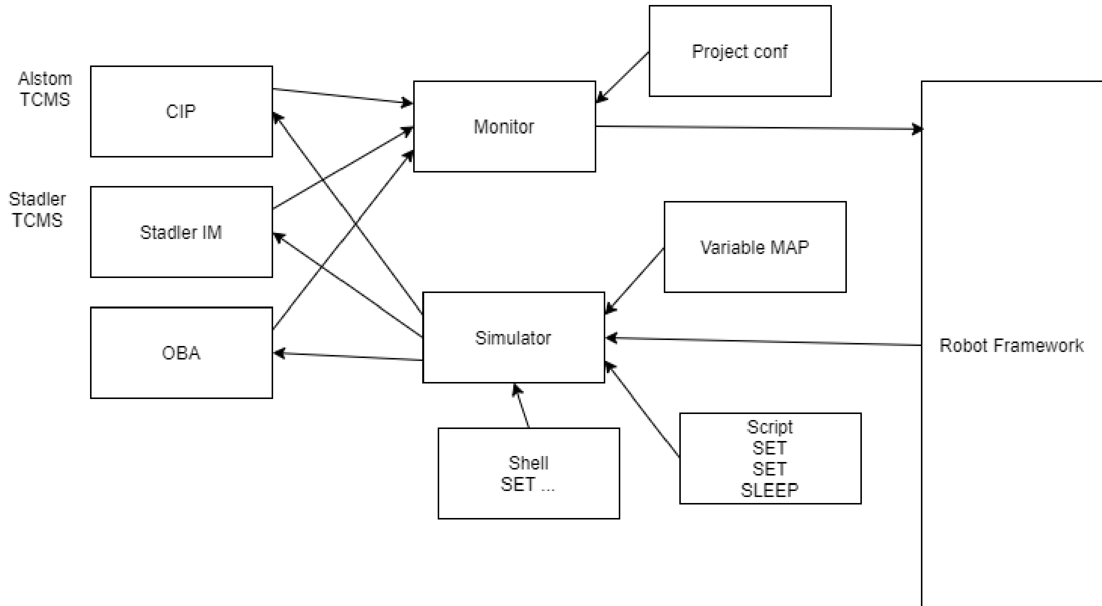

Image 3: Initial Plan of Tobamon

The monitor element would observe any, and all devices described in the configuration file. As the devices under surveillance typically have a myriad of different variables at run-time of which only a small fraction is ever of interest to the tester, the configuration file includes a section for specifying which items are to be actively monitored. The initial design of the monitor element is given in Image 4.

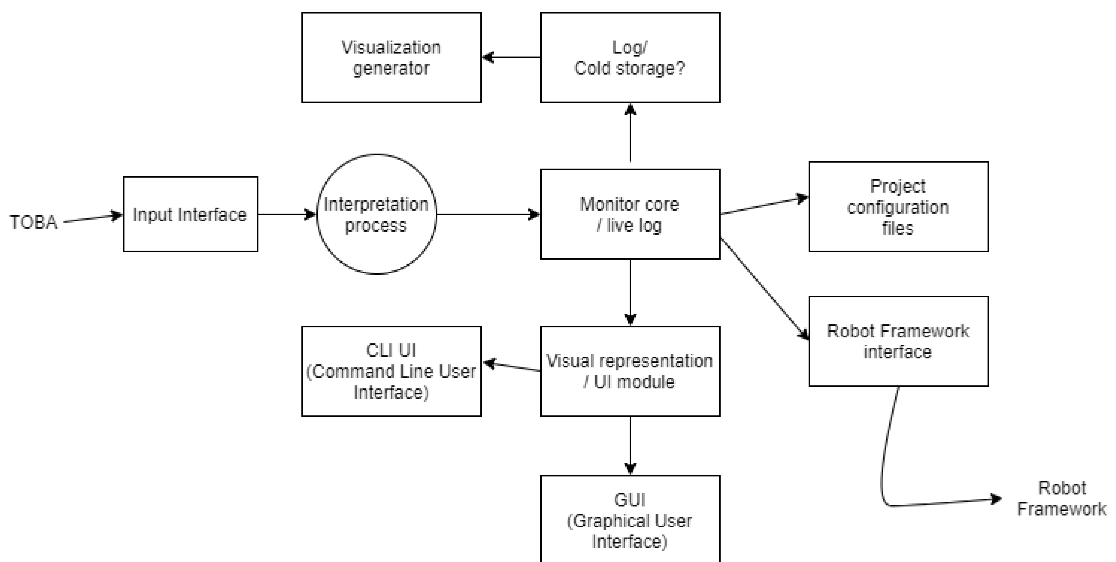

Image 4: Initial Plan of the Monitor element.

Such was the initial concept. During development it would continue to change its shape somewhat. The principal components of the Monitor came to be as follows:

- Monitor Core, which is the heart of the application and brings the other components together. This corresponds to the “Monitor core / live log” box in Image 4. The live log functionality was moved to the DataHolder component.
- The DataHolder, the task of which is to store and curate the data. This component corresponds mostly to the ”Log/ Cold storage?” item in Image 4, but with the live log functionality combined into it as well.
- The Visualizer, which would serve an API for displaying the data. It would also serve a web page which displays the data in a line chart form. This component corresponds to the “Visualization generator” and “Visual representation / UI module” items in Image 4. The “CLI UI (Command Line User Interface)” and “GUI (Graphical User Interface)” items were replaced by a browser-based UI which was rolled into the Visualizer.
- The Listener, which opens a connection to a server on a train and queries it regularly. Alternatively, it subscribes, if such a functionality is available. This item corresponds to “Input Interface” and “Interpretation process” items in Image 4.
- The “Project configuration files” came to be integrated as they were conceived in Image 4.
- The “Robot Framework Interface” eventually had to be scrapped due to time constraints.

These are of course only the briefest descriptions of the components. A more detailed description is available in Section 5.

The simulator component would connect to the device and manipulate specific variables on it. Like the monitor element, it would be configured to only alter variables which are specifically allowed by the simulator component’s own configuration file. Image 5 presents an initial design sketch of the simulator component.

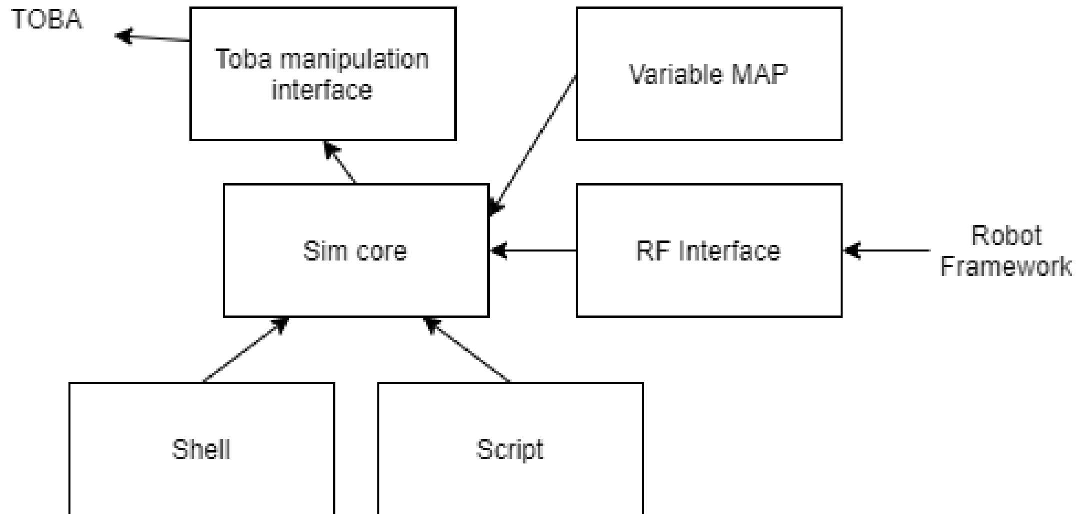

Image 5: Initial Plan of the Simulator element.

The third envisioned major feature was the interface to use Tobamon through Robot Framework, enabling scripting for the product. Using the simulator component, Robot Framework manipulate the devices' state, and through the monitor component, it would observe the consequences. This functionality existed only as a very rough sketch, which is illustrated by Image 6.

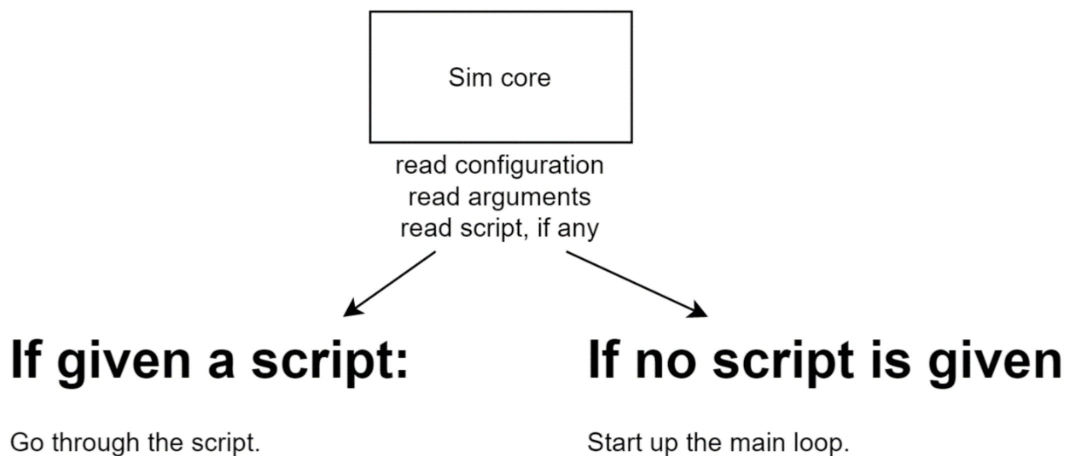

Image 6: Initial Plan of a Scripting System

As progress on the project proceeded, it became clear which intended features would be included in the final product. First to go was the Robot Framework interface, second was the Simulator element itself. Certain technical difficulties in the implementation also took up disproportionate lengths of time.

4.3 Automated testing

Testing is important and automated tests make everything a lot easier. It is instrumental in creating a routine of maintenance for code and contributes to the transferability of code. Writing automated tests saves time and it enables a programmer to keep working more effectively. [17] Towards this end, automated testing was implemented in Tobamon. Unit tests were developed for each of the components of the Monitor.

What is a unit test? It's an automated test that verifies some limited unit of code, does it fast, and does it in an isolated manner. There are different schools of thought as to what should constitute an isolatable unit, but in Tobamon the components of the Monitor were treated as just such units. Unit tests are divided into test cases. Test cases which pertain to the same unit are collected into test suites. [14][17]

A test case is a set of inputs, the conditions under which the code is run, and the expected results of the run. If the actual results match the expected results, the test case has been run successfully. [14][17]

In this project, a hybrid strategy of equivalence partitioning and fuzzing was pursued where possible. Equivalence partitioning, also called equivalence class partitioning, is a type of testing where the unit is given input from the one end of its range of acceptable values, then something from the opposite end, and something in the appropriate range [14][21].

Equivalence partitioning was supplemented by a limited implementation of fuzz testing. Fuzz testing, or fuzzing, is a testing technique which gives random and inappropriate data as an input to the unit being tested [20]. For functions with no parameters, a single unit test would be created as a bare minimum. This hybrid strategy was conceived to force the creation of a sufficient number of basic tests.

The general idea was to give as input something that ought to be too little, something that ought to be too much, something to represent typical, valid input, i.e., “just right”. Lastly, one function test with parameter input where its data type or its content would be totally invalid for the function. In this way, the function would be covered from a few broad vectors of error as well as to check for normal behavior. This strategy is illustrated in Image 7.

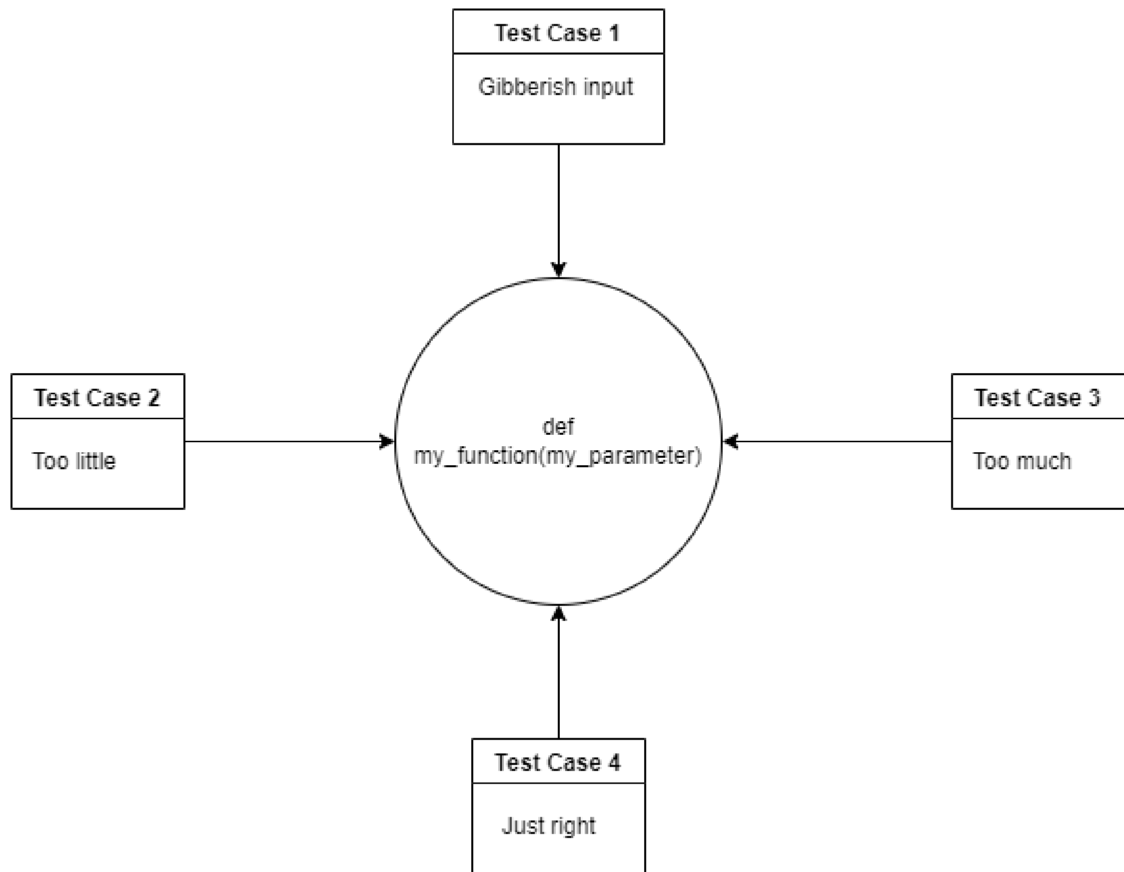

Image 7: A testing strategy combining equivalence partitioning and fuzz testing.

4.4 Auxiliary code

During the project it was necessary to create several smaller programs to test out certain ideas for Tobamon or to generate test data. In the former category, there are many examples, with the most prominent one being the attempts at implementing communication with an OBA server.

The latter group was developed to generate a set of data to use for testing while developing the components, and to determine the proper format of the data. Before it was possible to implement the Listener, which would have provided data to work with, it was necessary to have some data to test out the DataHolder and the Visualizer.

The first data generator that was created gives out a JSON file, which contains an array of coordinates. Both the X and the Y coordinates are floating point numbers. The code below code was called “BigTestFileCreator”. It proved to be a good stress test for the system, as well.

```
import random
import json

from sys import getsizeof
```

```
def create_entry(x_base = 0):
    return {"x": round(x_base + float(random.randint(1, 99)) / 100, 3),
            "y": round(random.random() * random.randint(1, 100), 3)}


if __name__ == '__main__':
    my_dict = {}
    length_of_tables = 100
    # Generate labels
    for i in range(0, 10):
        my_dict["item" + str(i)] = []

    # Generate random contents to each
    for key in my_dict:
        new_item = {"x": 1.0, "y": 2.0}
        for i in range(1, length_of_tables):
            new_item = create_entry(new_item["x"])
            my_dict[key].append(new_item)
        print("Done with key " + key)

    with open("BIG_TESTING_FILE.json", encoding='utf-8', mode='w') as out-
put_file:
        json.dump(my_dict, output_file, ensure_ascii=False, indent=4)

    print(getsizeof(my_dict))
```

First, the code instantiates a Python Dictionary (typically referred to as “dict”) and an integer variable called “length_of_tables”, which can be any positive integer. It then inserts ten keywords into the dict, with names spanning from “item1, item2… item10”. The value behind each keyword is an empty Python List (typically merely called “list”).

For each of those keywords, it creates as many entries into the corresponding list as has been configured into the variable “length_of_tables”. Each entry is an (x,y)-coordinate, and each entry is derived from the previous one by multiplication with random integers. The first entry is derived from a seed value of “{"x": 1.0, "y": 2.0}”.

At the end of its run, BigTestFileCreator writes all its data to a JSON file. In the above example, this is given as “"BIG_TESTING_FILE.json"”.

Apart from the test file generators, a great number of very small files were created to prototype certain features of Tobamon. Some small files were created to try out certain Python features to make sure their behavior was correctly understood by the author.

4.5 Communicating with device server

There was initially some confusion about which protocol the Listener should initially be built for. One alternative that was considered was the CIP protocol. It was left out due to concerns about it being rather complicated. The second choice was GraphQL, a popular query language for APIs.

To make use of GraphQL in Python, it was necessary to choose a client for it. The Python PIP package catalogue contains several modules for making use of GraphQL, some more mature and robust than others.

The maturity of the Python ecosystem provides for some pitfalls as well. Because there had been a messy move from Python 2 to 3 [18], there are libraries and implementations of various things separately for the two versions of Python. Many a promising library was in fact built for the outdated Python 2, and thus not usable for this project.

A helpful little tutorial by one Melvynn Fernandez provided one alternative. It guides the reader into communicating with GraphQL using the Python module 'requests' [31]. An initial attempt was prototyped using this tutorial as an example. This method was straightforward, and it could successfully perform queries and mutations. The former means a singular one-time request of selected data from the server, the latter denotes an attempt to alter data on the server [10].

However, the idea of using the 'requests' module eventually had to be scrapped. This was due to concerns about 'requests' not being able to make use of the subscription feature of GraphQL. The search had to continue.

For a preferable alternative, the website of GraphQL [12] itself was perused. There a catalogue of sundry libraries and modules for several programming languages found. Under Python itself, there were several entries.

There are many modules for implementing servers and many again to use as GraphQL clients. First on the latter list was GQL, which seemed rather prominent and mature. It was also apparent that it was being actively maintained. The latest release (time of writing: 3.11.2021) was only four days old. [39].

4.6 API endpoints

One of the requirements of the monitor element of Tobamon was a system for representing data visually to a user. At the very beginning, a few options were considered: creating a terminal program for printing out the data, creating a native desktop graphical user interface (shortened to GUI), or to develop a browser-based graphical user interface (shortened to Web GUI). After a little deliberation with coworkers, the browser-based solution turned out to be the most preferred option.

To develop the visualization, a means of reading the data would be needed. For a self-hosted web page to be able to read the data, an API would have to be created for it. Towards implementing an API, the Flask web development framework was chosen.

Using Flask, creating API endpoints was easy. For testing the API endpoints, much use was made of an application known as Postman. Postman is introduced in more detail in Section 3. The set-up of Postman testing calls used in this project can be seen in Image 8.

In the created set of API calls for testing, there was a division into two groups: calls for the value of a variable at a given index and calls for the values of a variable from a given index onwards. During the development and design, certain test files wound up getting a few test calls specifically for themselves. More on the functionality of the finalized API in Section 5.4.

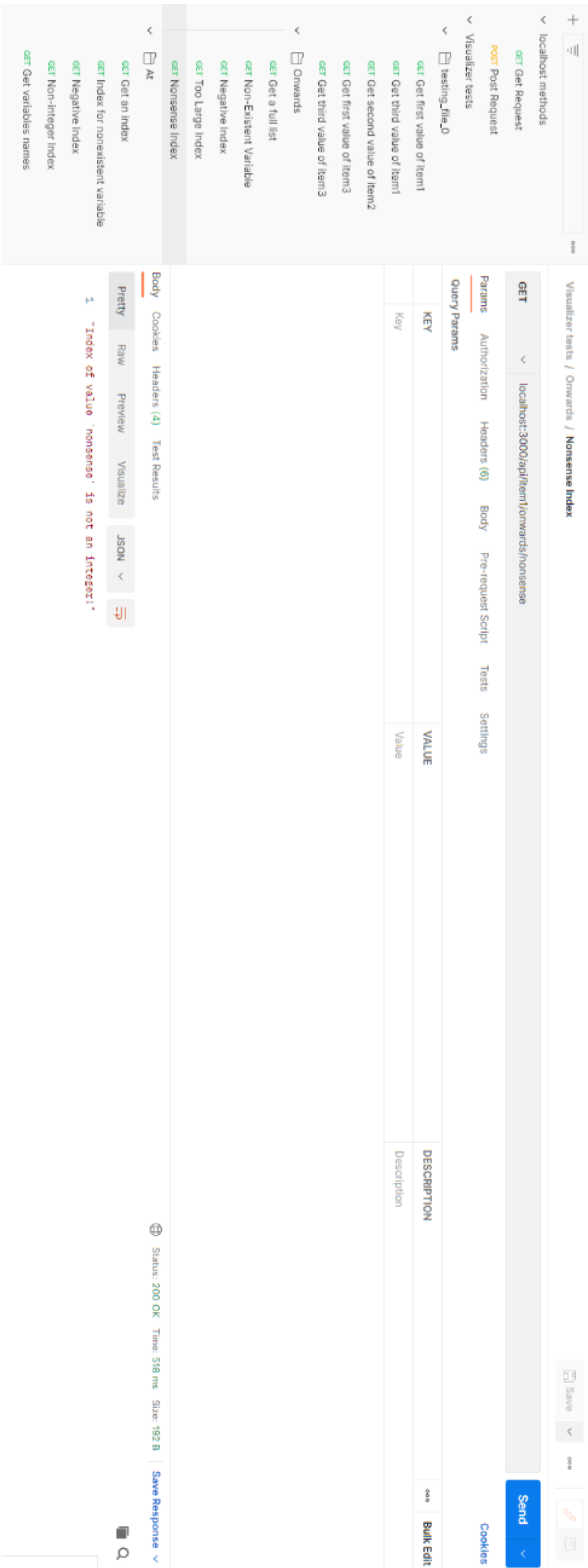

Image 8: Screenshot of the Postman setup.

4.7 Problems in generating a visualization

A browser-based solution was chosen as the means of viewing data, as it was the most modern and most intuitive solution. The choice of which library to use for that was not easy. There are many JavaScript libraries for generating chart representation of data, and many of them were tried.

4.7.1 The search for a library

Initial scouting revealed a few potential libraries for developing visualization. All of them had official tutorials, and these were made use of to measure their usability for the Tobamon project. Some of them were of completely original origin, but one was a wrapper written around an existing framework. All except one failed to produce a working software when the official tutorial was followed. The libraries are briefly summarized in Table 3.

Table 3: Chart drawing libraries for Javascript compared.

Library name	Licence	Nature of implementation	Dependencies	Tutorial done successfully?
Chart.js	MIT Licence	Direct implementation	None.	Yes.
Vue-Chartjs	MIT Licence	Wrapper	Chart.js	No.
Vue3Charts	MIT Licence	Direct Implementation	None.	No
ApexCharts	MIT Licence	Direct Implementation	None.	No

The first choice was a library called Chart.js [5]. It is the most obvious, most intuitive, and most largely used library for visualizing information in JavaScript. Each library was tried, but eventually Chart.js would prove to be the chart library of choice. Following its tutorial produced a working program, but despite this, Chart.js was initially dismissed due to a misunderstanding by the author. The misunderstanding was that Chart.js did not seem to support real-time changes to the data. This was eventually found not to be the case.

Several wrappers for Chart.js had been produced in Vue, itself a very common and potent visualization library, mostly intended for single-page applications. Vue allows binding values in the JavaScript engine to elements in the DOM and from then on, it makes

possible the real-time manipulation of these values. The changes are updated immediately to the bound element in the DOM. [37]

First on the list of Vue wrappers for Chartjs was “Vue-Chartjs”. The home page and tutorial are generally very high quality. [16] It was even more peculiar, that following the tutorial did not produce a working program. It is not plainly obvious, for which version of Vue this library is built for. The version of Chart.js used in the tutorial is Chart.js 2.7.

The second candidate for generating data visualization with Vue was “Vue3Charts”. It is built with Vue 3. The tutorial and documentation on Vue3Charts are a lot sparser than Vue-Chartjs, implying that it might not be as mature a product. The version number at the time of writing (25.10.2021) is “1.0.18”, implying some degree of maturity. [2] Following and trying to implement the tutorial code for this library did not produce a successful prototype.

The third option was a library called ApexCharts. Based on the navigation bar on the website, ApexCharts includes embedded analytics, possibly making it much more potent and useful than the others. [4] Following the tutorial did not produce a working product.

Despite the official tutorials and examples being followed to the greatest possible degree of accuracy, none of the Vue-based charting libraries seemed to work. Some tutorials however were obviously written for an older iteration of Vue and/or Chart.js.

Many tutorials were written with the assumption of developing in a Linux-based environment. Development of Tobamon was conducted in a Windows environment. The Linux ecosystem remains the OS of choice for programmers.

Why did following so many of the tutorials fail? It could well be, that the fault was entirely with the author and not the libraries themselves. On the other hand, the project was being pressed against time. An inordinate amount of time had already been spent on the chart libraries. It was simply not sensible to spend much more.

It was after these other libraries were dismissed one after another that Chart.js was given another view. It was then found that real-time manipulation of the chart was in fact supported. Development on the visualization could finally commence.

4.7.2 Chart.js for development

Chart.js came to be the library of choice, but it was not without its faults. It seemed initially that in Chart.js it is not in fact possible to set the labels on horizontal nor vertical lines with arbitrary steps. It is thus also not possible to add new values between such steps. It seemed all new steps were necessarily also the lines on which each step of the chart

is divided into. This all makes it impractical, or at least not ideal, to add visualization for several different variables into the same chart.

A solution was found after a little searching: it appeared that someone else had had the same misunderstanding about the features of this library. An issue about this had been posted on the Github repository of Chart.js [34]. Reading the post, it turns out it was a duplicate issue. Someone else still had misunderstood the workings of the labels [29].

The basic line chart in Chart.js, presented in the tutorial, did necessarily place the points X-coordinate on top of the X-axis labels. However, this is not necessary for scatter diagrams. A scatter diagram and a line diagram are not necessarily mutually exclusive in Chart.js. While a line diagram could not be converted to a scatter diagram, the latter could easily be configured to draw connecting lines between the points. With scatter diagrams, it became possible to create the intended kind of chart.

There were points during development, that access to the company test bench was not available. In its place, another means of testing the Visualizer was needed. A test file generator was described in Section 4.5. It was adequate to allow developing the cooperation of the Core, the DataHolder and the Visualizer.

However, because the generator merely generated floating point numbers, it was not a sufficient representation of the kind of data Tobamon would be receiving. Neither was it the form in which Tobamon would be storing them. For each value received from the connected device, the time at which that value came about would be important as well.

It was necessary to develop another test data generator, one which would give a datetime value as the x-coordinate. This then provoked another question: “What format of datetime should it use?” There are a number of different time, date, and datetime formats. Examples include the ISO format, the Python format, among others. After some experimentation, the Python format for datetime notation was chosen for use.

The datetime generator was called “DatetimeListGenerator”, and its code was thus:

```
import json
import random
from datetime import datetime, timedelta


def generate_list(number_of_entries: int, now) -> list:
    output_list = []
    for i in range(1, number_of_entries + 1):
        new_value = round(random.random(), 3) + random.randint(0, 100)
        new_time = now + timedelta(seconds=i)
        new_time = new_time.strftime("%Y-%m-%d %H:%M:%S.%f")
        new_time = new_time[:-3]
        new_entry = {
            "x": new_time,
```

```
                "y": new_value
            }
            output_list.append(new_entry)
        return output_list


    if __name__ == '__main__':
        now = datetime.now()
        number_of_entries = 10

        my_object = {}
        for i in range(1, number_of_entries + 1):
            my_object["item" + str(i)] = generate_list(number_of_entries, now)

        with open('datetime_test_file_0.json', 'w', encoding='utf-8') as out-
put_file:
            json.dump(my_object, output_file, ensure_ascii=False, indent=4)
```

The code is somewhat alike the earlier generator. Once again, a "number_of_entries" is configured. This time, the datetime value of the very moment of running is also instantiated, along with a dict called "my_object". Then, for as many times as is the value of ”number_of_entries", an entry is created into "my_object". The keywords are once again "item1, item2… item10" and for each, a list is generated using the "generate_list()" function.

The function takes "number_of_entries" and "now" as its input arguments. Within, It enter a for-loop which iterates as many times as is the value of "number_of_entries". At each iteration, it increments the value of now, and saves the new value into a variable called "new_time". It generates a random value for y. and for x it assigns the "new_time" value. After the iteration is done, it returns the generated list of values.

Back in main, the loop therein will generate lists for each of the keywords in the manner described above. It then writes the generated values into a JSON file, in this case called "'datetime_test_file_0.json'". Image 9 shows Tobamon Visualizer component running on generated test data.

In Image 9, there are ten sets of data from a programmatically generated test file: "item1" to "item9" and "Booleans". Data sets prefixed with "item" contain floating point values as their Y-coordinates. The X-coordinate is a time value with an accuracy up to a hundredth of a second. It includes the date to ensure that if a test is left running past midnight, the later values do not start appearing before earlier ones in the chart. The dataset labeled "Boolean" is otherwise similar, but the Y-coordinate is of the Boolean data type. That is, it can have one of two potential values: "True" or "False". The charting library represents these as 0 and 1.

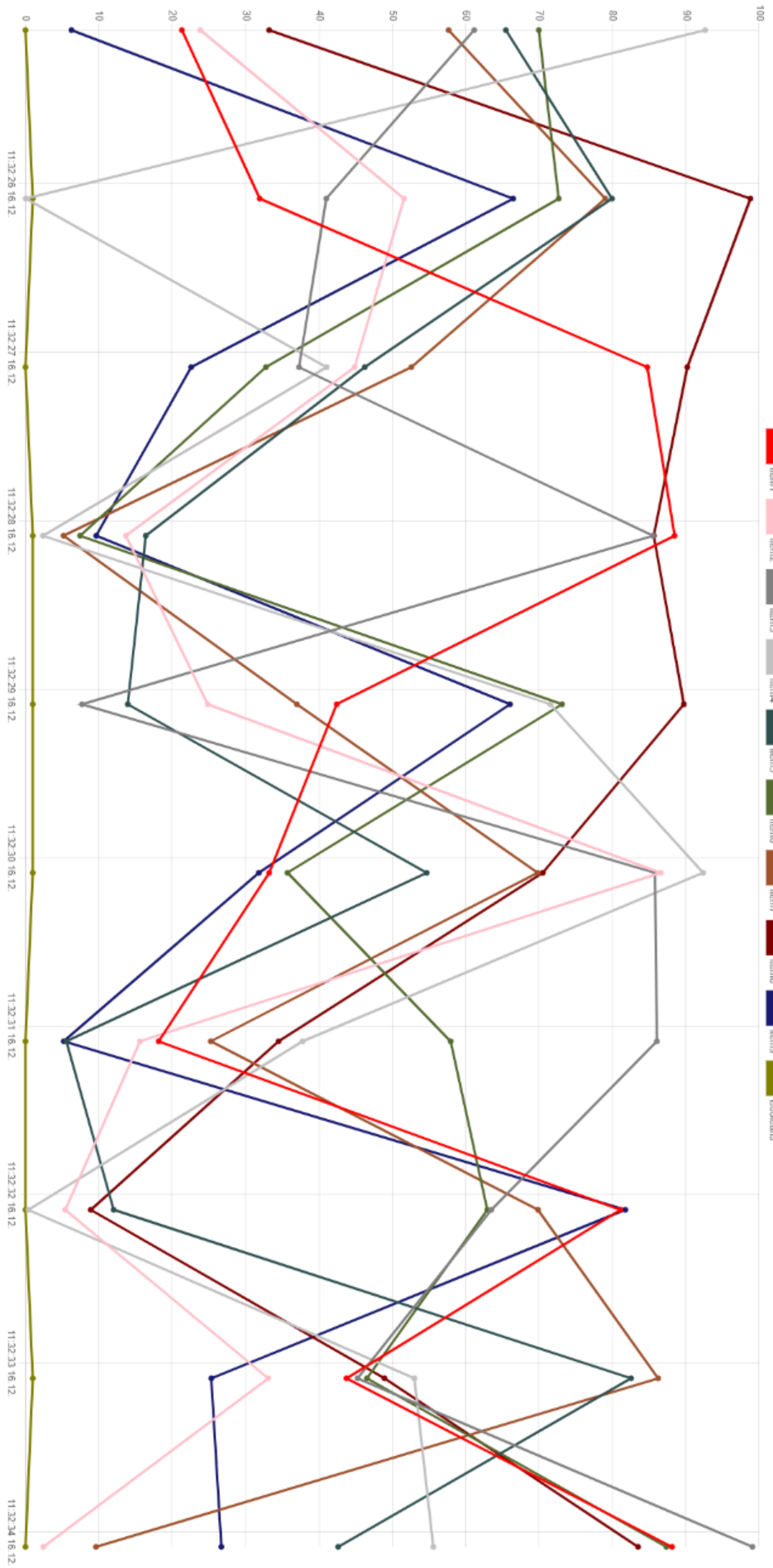

Image 9: Visualizer running with test data.

5. IMPLEMENTATION

‘The application, which is known as Tobamon, is a software program used for monitoring the states of devices on a train or a test bench substituting for a train. The latter is the more typical use. The devices and their variables which are to be monitored can be set with the configuration file of the program.

In a typical testing setting involving a test bench, a computer is connected to the test bench using an Ethernet cable. Other connections can be used if they are available. An abstract representation of a typical testing environment is provided in Image 10. Testing environments are further illustrated by Image 12.

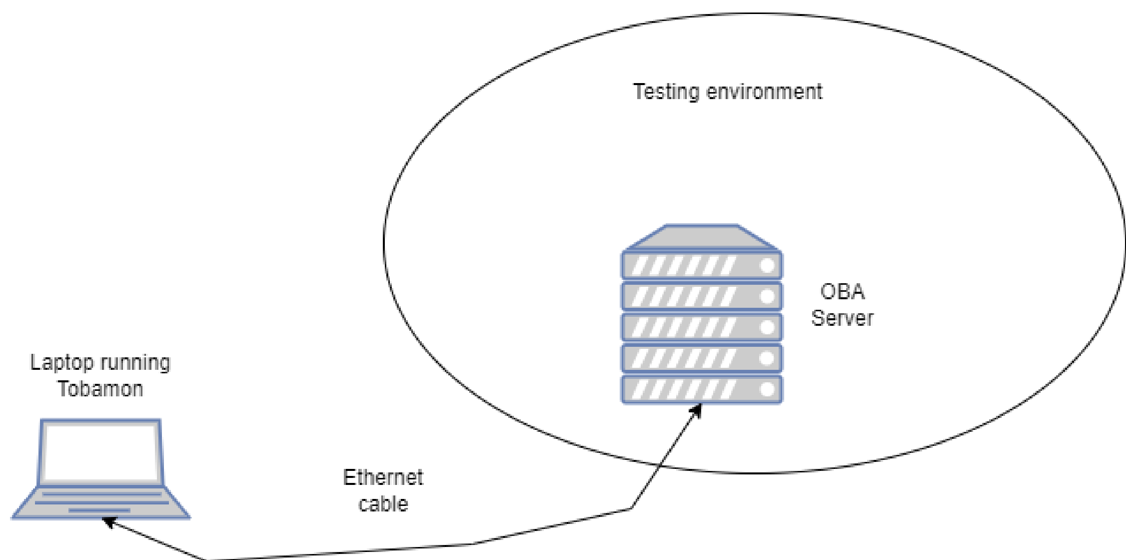

Image 10: Typical use environment for Tobamon.

The application is made up of three separate components in addition to a core component, which connects all the others. Some of the components run within their own dedicated process, while the others run in the same one as the core. Some of the components run within their own threads inside the core process. The three components apart from the core are: the Listener, the DataHolder and the Visualizer. All come together to perform the intended actions of Tobamon. The class structure of Tobamon is presented in Image 11.

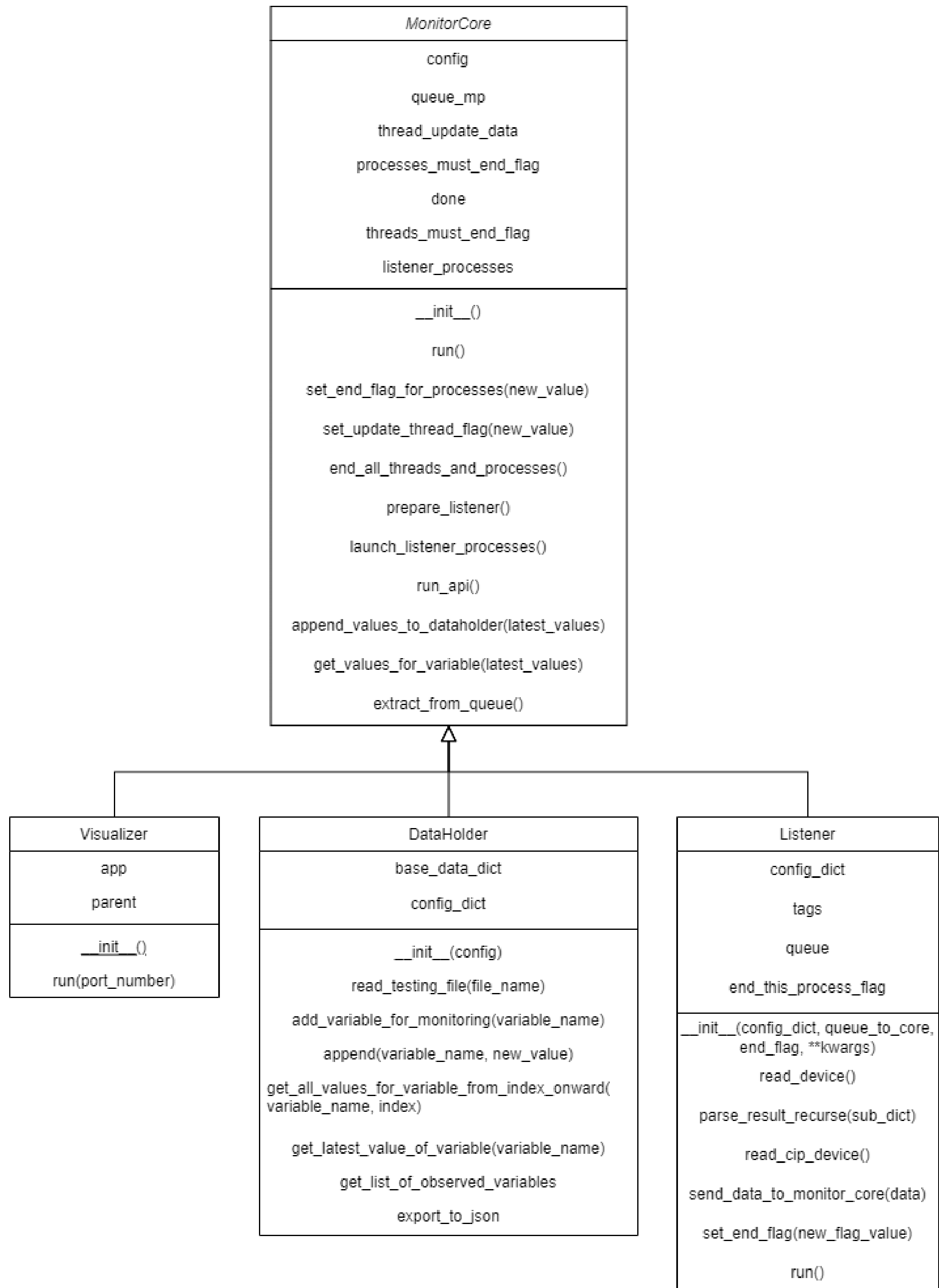

Image 11: Tobamon Class Diagram.

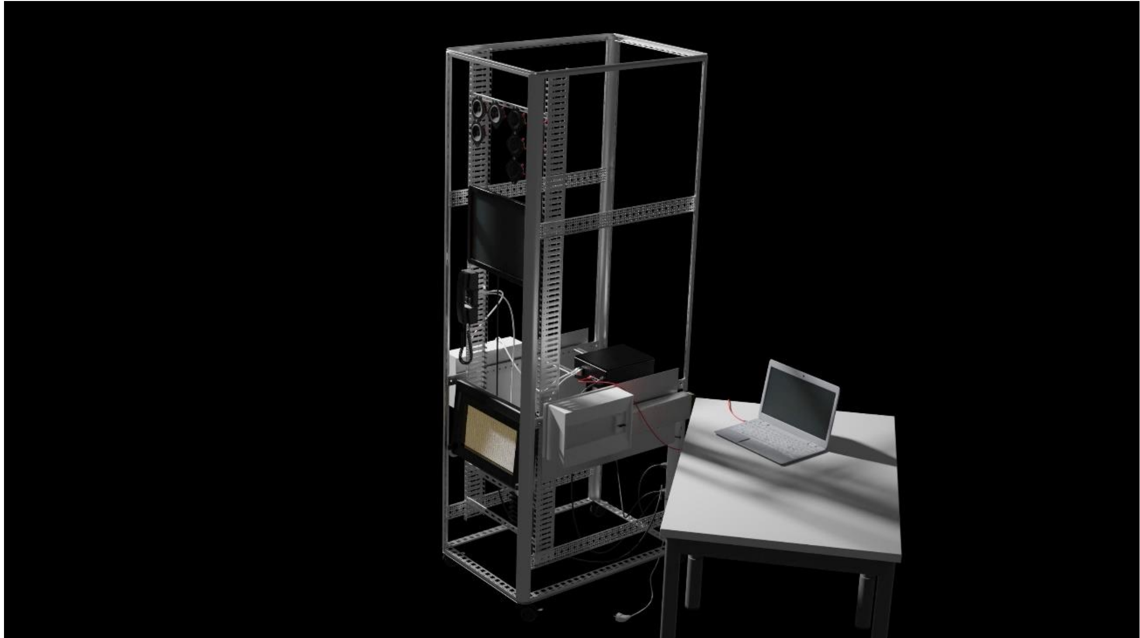

Image 12: A 3D render representing a generic test bench setup.

5.1 Tobamon Core

At the heart of Tobamon is the component simply called the “Core”. This component brings together all the other components of Tobamon and coordinates their behaviour. For example, when the Visualizer needs data, it cannot access the Dataholder directly, but does so through an interface provided by the core. Thereby, Tobamon is designed according to a design pattern known as the “Mediator” [19]. The shape of Tobamon is illustrated in Image 13.

What is the Mediator pattern? The Mediator pattern is means to allow sets of objects to communicate data in some way, when they are otherwise self-contained and mutually separate wholes. Certain mutually related functions and functionalities are packaged into objects, which represent such wholes. Those objects are solely responsible for their given task. Communication is given as the task of a separate object. [19] In Tobamon, that object handling the communication is referred to as the Core.

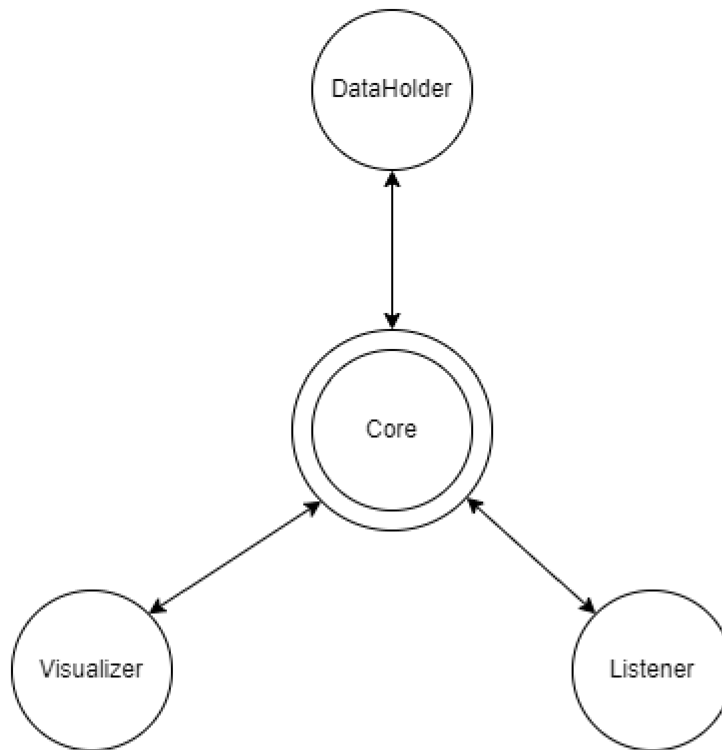

Image 13: The Core sits in the center and coordinates the other components.

The Core runs in the main process of the program, but it initiates a few other processes and threads. For each Listener configured for running, the Core initiates a separate, dedicated process. For getting data from the Listeners, it instantiates a data structure of shared memory called “Queue”, which is part of the Python ‘multiprocessing’ library. Note that the implementation described in this paper only uses one Listener instance at a time. This is only so to keep the scale of the program manageable for a thesis work, but in real usage this limit would not be necessary. The Core also creates a separate thread for retrieving data from the Queue and gives it onwards to the DataHolder.

The configuration file of Tobamon is read by the Core, and it provides the appropriate sections to the other components. The file is parsed using Python standard library’s ‘configparser’ module. Thus, the format of the configuration closely resembles that of Microsoft Windows INI files. An example of the contents of the file is given here as thus:

```
[DEFAULT]
DevelopmentMode=True

[VISUALIZER]
PortNumber=3000

[DATAHOLDER]
ReadTestingFile=True
TestFileName=datetime_test_file_4.json

[OBA]
```

```
; Listener
IpAddress=http://10.59.1.136:8888
CheckingInterval=100
Tags=tag1,tag2,tag3,tag4
Command: ”””subscription {
            tag1
            tag2
            tag3
            tag4
         }
    „„„
```

The first section is the “default”, into which general configurations of Tobamon in general and Core specifically are to be given. There is a separate section for the Visualizer, which must contain an entry for the port on which the Visualizer is supposed to run. 3000 is a good default value. Then there’s the section for the DataHolder. It can be set to read a testing file, which can be specifically designated. This is also useful for taking records of previous test runs and visualizing them.

Past these are the sections for the devices which Tobamon is to connect to and which it is supposed to monitor for the tags. Configuration includes the IP address of the target device. “CheckingInterval” designates the interval period in milliseconds, between which the Listener is supposed to query the target device, if subscriptions are not available. The tags designate those variables on the device, which the tester wants to monitor. “Command” contains that command which is sent to the target device and is used to either query or subscribe for the tags.

5.2 Listener

The Listener, as its name implies, listens constantly to that device which is configured in the configuration file. The specific configuration of a Listener instance is given as a section of the main configuration file. Any section apart from “DEFAULT”, “DATAHOLDER” and “VISUALIZER” will be instantiated as a Listener process.

The Listener class is implanted as inheriting Process class from the Python library “multiprocessing”. Therefore, the Listener can be called Listener processes or Listener objects interchangeably. The Listener also inherits the abstract ListenerInterface class. The class diagram of the Listener is provided in Image 14.

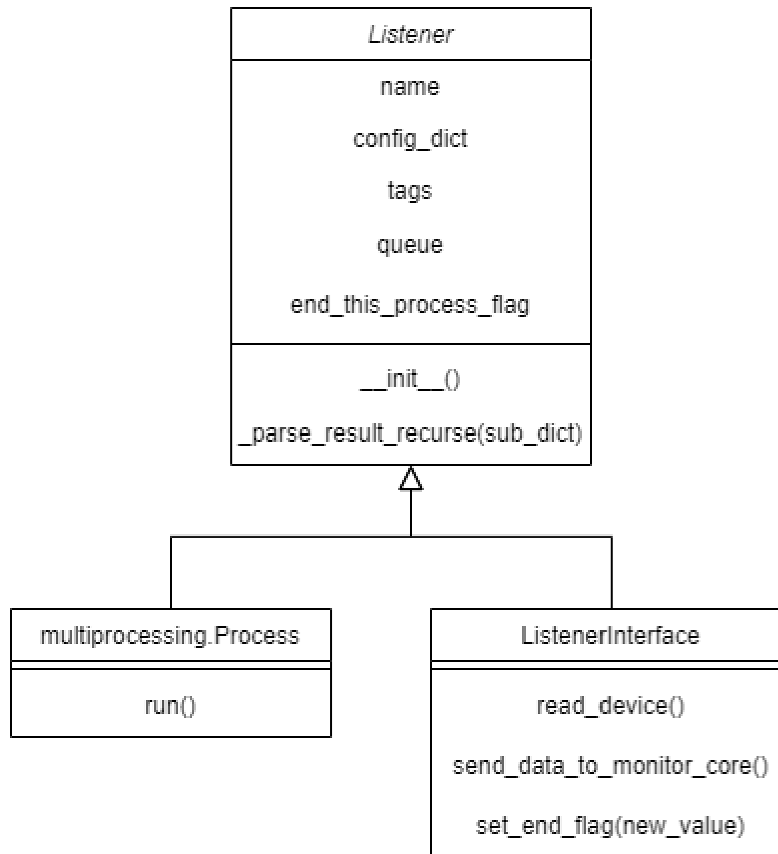

Image 14: Listener Class Diagram.

All Listeners are initiated from the core component process. The number of Listener instances can be set by creating new sections for connections to be listened to. That version of Tobamon which is described in this paper only supports one Listener connection at a time. Listeners are given their section from the configuration file as their configuration settings. This includes an IP address, possibly also a port number, a list of tags and a query string. The tags in the list are the labels of the devices and variables which Listener is set to listen for.

The query string contains a command which the Listener gives to its target connection. The command in the query instructs the connected device on which data to send to the Listener. The Listener has list of tags which is used to parse the desired data from the reply to this query.

Active connections are not the only alternative, as some connection types support subscriptions. The implementation of Tobamon described in this paper exclusively uses GraphQL connections over TCP/IP to its set devices. GraphQL supports subscription queries. Subscription is preferred instead of active pinging, as it can reduce unnecessary network traffic. In a subscription, the GraphQL server sends data to the subscribing service whenever a change occurs in the tag which is being subscribed for.

Upon receiving data, the Listener parses it, adds a timestamp, and sends the product to the DataHolder. This data transfer is sent by means of a special data structure called “Queue”, provided by the ‘multiprocessing’ library of Python. This is necessary, because the Listener instances run within their own processes with their own self-contained memory, which is not shared with the core process, which too has its own memory space. A Queue object provides a means of sharing memory across processes. This process is illustrated in Image 15.

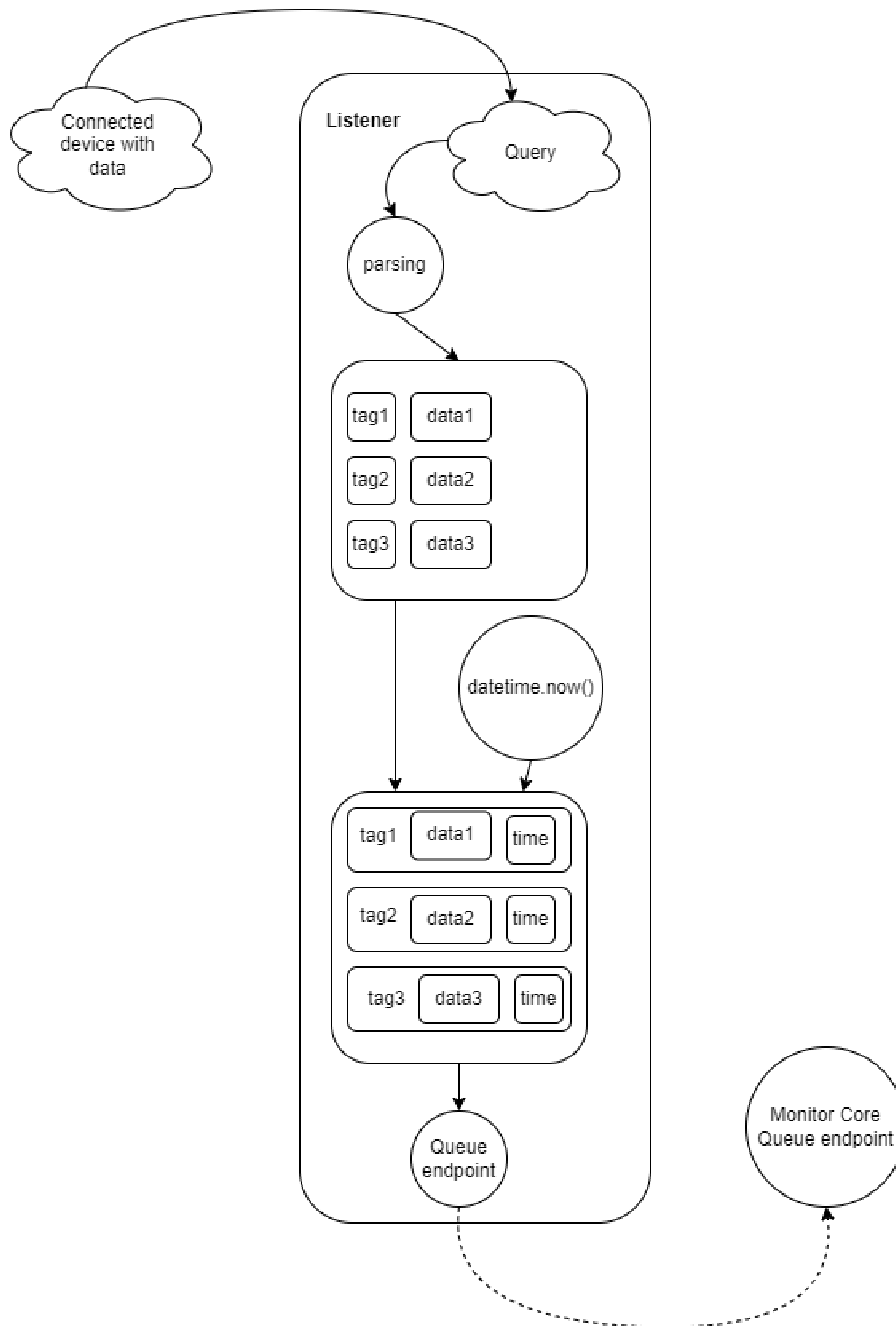

Image 15: Listener workflow.

As a means for the core process to control Listener processes, these share a common flag called the “end_all_processes_flag”. This is implemented by means of another special data structure from the “multiprocessing” library called “Value”. A Value can be instantiated as any primitive data structure. In this case, it is set to be a Boolean value.

5.3 DataHolder

The DataHolder acts as the curator and archivist of the data collected by the application through the Listener processes. Tobamon only ever has one DataHolder instance, and it is contained within the core process. The DataHolder gets a section all for itself from the Tobamon configuration file passed unto it as its configuration settings.

The DataHolder also receives certain information from the sections for the devices to be monitored. For each device to be connected, a list of tags is included. These tags are also given to the DataHolder, which generates an entry for each of them. Note again, that within the implementation of Tobamon described in this paper, only one device is included.

Data is organized into entries, which are implemented as Python dict datatypes, commonly also called “dictionaries”. A Python dictionary is a kind of data structure which is organized as key-value pairs. Each tag that Tobamon is set to monitor for is given as a keyword in the dictionary. Each entry then contains a list, which is an array-like data structure in Python. This arrangement is illustrated in Image 16.

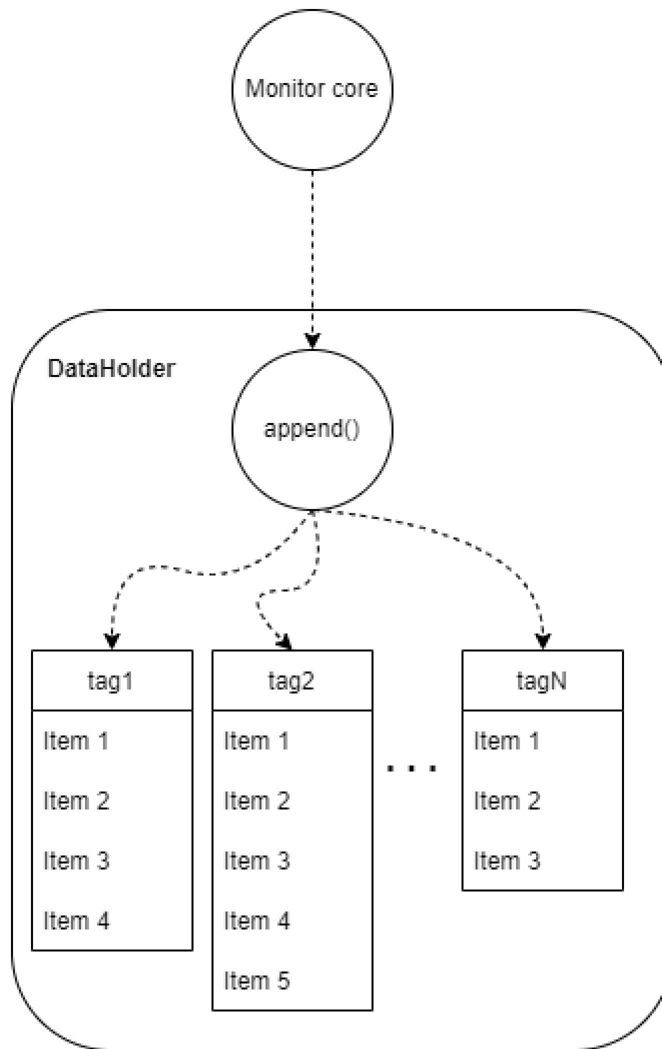

Image 16: DataHolder saving data.

The DataHolder is fed new values from the core object, which receives them from Listener processes as described in Section 5.2. The DataHolder does not do any formatting or mutation to the data, but only checks that they are of the same data type as previous values for that variable.

The DataHolder can then retrieve any parcels of data stored within it, if requested. This is for the benefit of the Visualizer which will be described in more detail in 5.4. The data which is put out will be given to the Core, which will deliver it wherever it is needed.

If so desired, the DataHolder can be set to export its contents in JSON format. This created file can later be used as input for the DataHolder when not connected to a listenable device and not doing monitoring. In this way, it is possible for Tobamon to generate a visualization of past testing runs.

5.4 Visualizer

The Visualizer takes the data from DataHolder via the Core, provides an API to retrieve them and can be used to generate a visual representation of it. Tobamon only ever has one Visualizer instance. This component serves an API for reading the data, but no means for putting data in.

5.4.1 API implementation

The API provides four endpoints. They are presented in Table 4.

Table 4: API endpoints for the Visualizer.

	Method	URI Path	Description
1	GET	/	Get web page with chart. Use with browser.
2	GET	/api/variables	Get a list of all the variables Tobamon is configured to monitor for.
3	GET	/api/{variable name}/starting/{index}	Get all values of the variable designated in the URL, from that index onward which is designated in the URL.
4	GET	/api/{variable name}/at/{index}	Get the particular value of the variable designated in the URL, from that index which is designated in the URL.

This component is implemented with a combination of Python and JavaScript. The former provides the back end, the latter the front-end. The Python back end is implemented using a framework called Flask. The front-end makes use of certain JavaScript libraries provided by npm. These technologies were introduced in more detail in Section 3.

In its typical state of operation, Tobamon has the Visualizer serve an API. This API provides not only the endpoints for querying the data, but by requesting the root URL, '/', it serves a web page which provides a graphical representation of the values stored by the DataHolder. The chart is provided by means of a JavaScript library known as Chart.js. More on this and other dependencies was explained in Section 4.

The API provides an endpoint for getting a list of the tags which Tobamon is set to monitor for. This is made use of in the web page visualization, but more on that in Section 5.4.2.

The third endpoint of the API, and the most frequently used one, is “/api/{variable name}/starting/{index}”. It returns all values for a given tag starting from the given index all the to the latest value in the correspond list. The “{variable name}” in the URL designates which tag is being queried for. The “{index}” designates the index.

The last API endpoint is “/api/{variable name}/at/{index}”. This is rather like the third endpoint, with the difference being that this one returns only one value for the tag which is found at the given index.

5.4.2 The web page view

There is a web page view provided with the Tobamon Visualizer component. It is available from the API at endpoint “/”, the base endpoint. The web page is implemented using HTML and various JavaScript libraries.

The working order of the web page is thus: first the page itself is loaded with all necessary static files. Then, the script of the page requests the API for all tags which are being monitored. The call is implemented using the Fetch API and is sent to the endpoint: “/api/variables”. The script will wait for the tags to be retrieved. For each tag, it will create a dataset which it will insert into the Chart object. The name of the tag is given as the label of the dataset. This process is represented in Image 17.

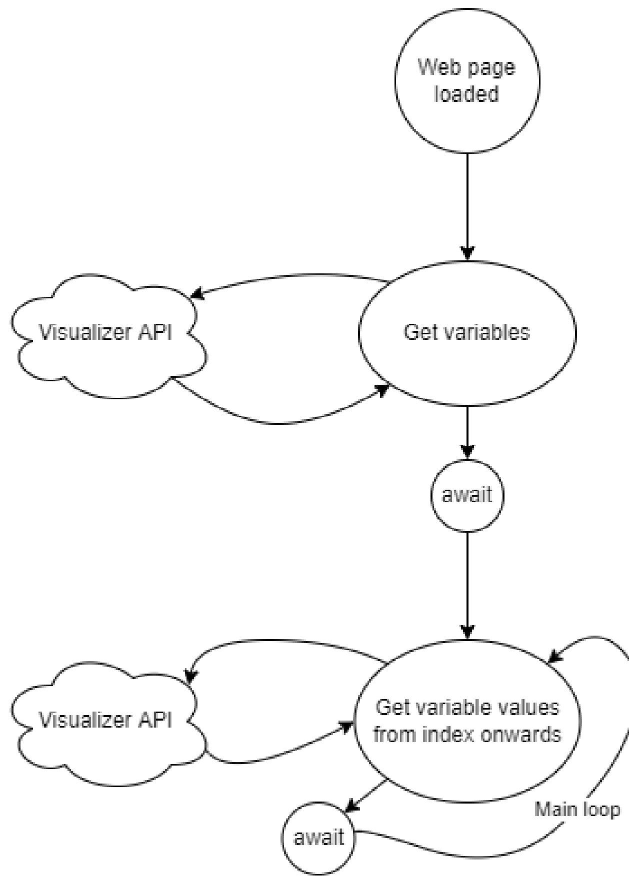

Image 17: Web page view script flow visual representation.

After the datasets have been initialized, the script enters a infinite loop, where at each iteration it sends a request to the API for each tag that is being monitored for. This call is implemented using the Fetch API. This call is sent to the endpoint: "/api/{variable name}/starting/{index}". An asynchronous Promise is collected towards each request, and at the end of the loop iteration, the code waits for each of them to resolve or reject.

The request for the values of a variable is packaged with an integer value called "index". When the web page view is making this request, it gives the size of the array within the dataset as that index. Thus, the web page view will only receive the latest values since it was last updated during the previous iteration of the loop. In this way, the code of the web page view avoids asking for duplicates of those values it has already received from the API.

For each dataset, there is a Boolean flag called 'underUpdate'. When a dataset receives data from the API, the value of its 'underUpdate' flag is set as 'true'. Hence, when the update loop encounters this dataset, it will not send another request for data. This too avoids duplication. After the update procedure for a dataset is complete, the 'underUpdate' flag is set back to 'false'. From thereon again, the main loop will enter update procedures for the dataset upon encountering it.

In Image 18 and Image 19, there are demonstrations of the complete Tobamon running on data from actual test devices. The X-axis represents a date and a time, with the latter being accurate to a hundredth of a second. The Y-axis represents numerical values from the queried devices on the test bench. Frequently these are floating point numbers, but integers and Boolean values are also possible.

The data in images 18 and 19 is retrieved from actual test devices, but they were induced by means of auxiliary code. The devices and variables given to use for testing Tobamon did not change frequently enough, a separate script was written to change their values arbitrarily, at arbitrary intervals. Nevertheless, the images demonstrate Tobamon running successfully on test devices.

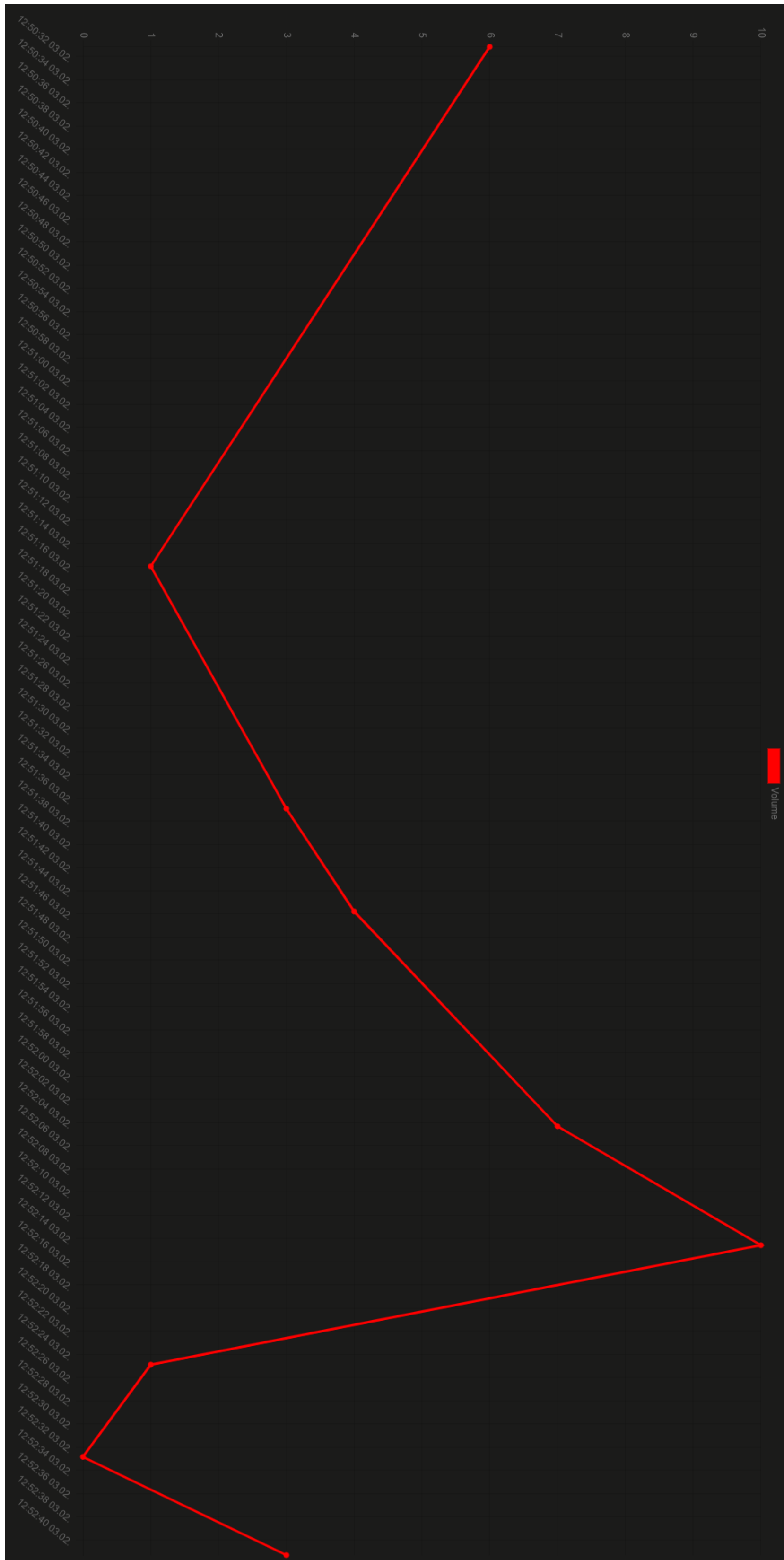

Image 18: Tobamon running with data from the device.

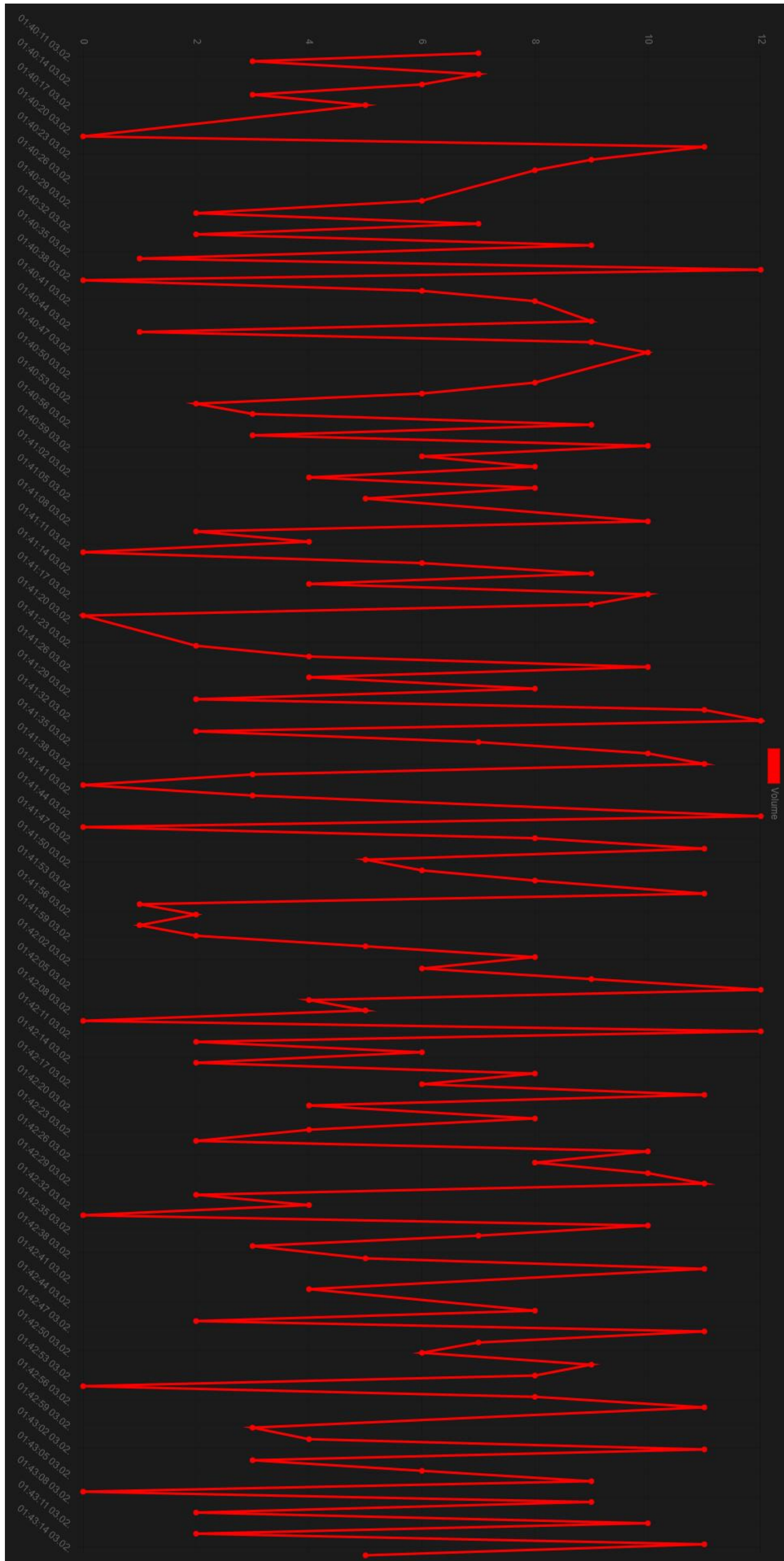

Image 19: Tobamon running with more dense data from an actual test device.

6. CONCLUSION

The Tobamon project is complete. Was it successful or not? This needs to be considered from a couple of angles: adherence to design science principles (laid out in Section 2) and success at fulfilling the initial plans.

6.1 Design Science Assessment

The relevance of this work to its own framework of DS can be reviewed by its adherence to the seven guidelines of design science.

The first guideline, “Design as an Artifact”, has been satisfied. An instantiation has been produced in the form of Tobamon. The second guideline “Problem Relevance” is met, as the project has been created as a commission by Teleste. The third guideline “Design Evaluation” is satisfied by the extensive testing and experimentation Tobamon went through in its development, as well as reviews with the employer.

The fourth guideline “Research Contributions” is met through the fact that an instantiation was produced. The fifth guideline, “Research Rigor”, was satisfied with the frequent consultation of best practices, documentation, and review with the supervisor. The sixth guideline, “Design as a Search Process”, came true, as the design of Tobamon started from the broadest descriptions and iteratively got more precise. The final guideline, “Communication of Research”, is met with the submission of this paper to Tampere University.

6.2 Design and Implementation Assessment

The final version of Tobamon described in this thesis, came to fulfill the Monitor element of the initial design described in Section 4.1 very closely. Much of the design of the complete Tobamon was left unimplemented, which is unfortunate. There is much potential in it.

It is the feeling of the author that the project not only took much more time to prosecute than was initially anticipated, but it also exceeded reasonable demands. It must however be acknowledged that in the software industry, projects notoriously have tendency to go past time estimations. Not only this, planned features often must be dropped to meet time demands.

As noted earlier, the selection and implementation of the chart drawing library took a particularly egregious and disproportionate amount of time. This was because the library and many of the concepts and terminology of chart visualization were at best only vaguely familiar to the author. It goes to show, that for a developer to adopt a new tool or framework, such as chart visualization, an extra amount of is going to be taken up by getting familiar.

Development on Windows and development on Linux come with differing obstacles and possibilities. It was found that the for the multiprocessing library of Python implementation of Windows, only one instruction for starting a new process is supported: spawn(). This is likely related to a peculiar behavior which was observed, whenever a process was started for the web framework Flask. The printing output of the main process became duplicated, if this was not deliberately prevented somehow.

6.3 Future Ideas

There are many possibilities for the further refinement of Tobamon. Many features envisioned in the initial design of the complete system had to be scrapped for time constraints. An obvious way to develop the product further would be to start implementing these missing features. The modular design of Tobamon was intended to facilitate maintenance and the addition of new features.

Further, implementations could be included for interfacing with devices using protocols other than the GraphQL. As noted earlier in this thesis, an implementation for interfacing with devices using the CIP protocol was originally considered. It could be added by creating a CIP-based implementation of the Listener component.

In this work, Tobamon was implemented using the Python language, which is a very convenient language that allows fast development. As a high-level scripting language, it might not be most ideal for software tool that is meant to be utilized and maintained for years or even decades. It is not necessarily the fastest performing language either. It might be a good idea to completely reimplement Tobamon in C/C++ or even Rust or Java at some future point in time.

6.4 Concluding Words

The Tobamon project has been a tremendous learning experience for the author. This has not only been an exercise in commissioned software development, but also a possibility to significantly develop one’s programming skills.

REFERENCES

[1] About Teleste, Teleste, (referenced 2.11.2021). Available: https://www.teleste.com/about-teleste

[2] G. Alexandru, Vue3-Charts, vue3charts.org, 2021, (referenced 28.10.2021). Available: https://vue3charts.org

[3] B. Angel, Programming Style Guides, the Blueprint of Clean Code, Pullrequest, 14.9.2018, (referenced 4.11.2021). Available: https://www.pullrequest.com/blog/programming-style-guides-blueprint-of-clean-code/

[4] APEXCHARTS.JS Modern & Interactive Open-source Charts, apexcharts.com, 2021, (referenced 28.10.2021). Available: https://apexcharts.com

[5] Chart.js, Chartjs.org, 2021, (referenced 28.10.2021). Available: https://www.chartjs.org

[6] Common Industrial Protocol (CIP™), ODVA, 2022, (Referenced 27.1.2022). Available: https://www.odva.org/technology-standards/key-technologies/common-industrial-protocol-cip/

[7] D. Crockford, JavaScript has Good Parts – presentation, deliciouspops, YouTube, 11.6.2017. (Referenced 13.1.2022). Available: https://www.youtube.com/watch?v=DogGMNBZZvg

[8] J. Daintith, E. Wright, A Dictionary of Computing. 6th edition. Oxford: Oxford Un

iversity Press, 2008. Print. Referenced (26.1.2022). Available: https://www-oxfordreference-com.libproxy.tuni.fi/view/10.1093/acref/9780199234004.001.0001/acref-9780199234004

[9] E. Engström, M. Storey, P. Runeson, M. Höst, M. T. Baldassarre, How software engineering research aligns with design science: a review, Empirical Software Engineering (2020) 25:2630–2660, 18.4.2020.

[10] M. Fernandez, Using Python to Connect to a GraphQL API, Towards Data Science, 14.11.2019, (referenced 30.10.2021). Available : https://towardsdatascience.com/connecting-to-a-graphql-api-using-python-246dda927840

[11] R.B. Fuller, A Comprehensive Anticipatory Design Science, Dalhouse University, Royal Architectural Institute of Canada Journal. v.34:no.9(1957), 1957. Available: https://dalspace.library.dal.ca//handle/10222/74680

[12] GraphQL A query language for your API, The GraphQL Foundation, 2021, (referenced 30.10.2021). Available: https://graphql.org

[13] A.R. Hevner et al., DESIGN SCIENCE IN INFORMATION SYSTEMS RESEARCH 1. MIS quarterly 28.1 (2004): 75–. Print. (Referenced 2.11.2021). Available: https://www.proquest.com/docview/218119584?accountid=14242&parentSessionId=kDIYhyjZ8kCTo72w2IqxwY%2BiR1UV0DdJ7VXZsidE5TU%3D&pq-origsite=primo

[14] ISO/IEC/IEEE International Standard - Systems and software engineering—Vocabulary, IEEE, ISO/IEC/IEEE 24765:2017(E), 2017, p.1-541, 2017.

[15] JavaScript, 29.7.2021, Mozilla Corporation. (Referenced 12.1.2022). Available: https://developer.mozilla.org/en-US/docs/Web/javascript

[16] J. Juszczak, vue-chartjs, vue-chartjs.org, 2021, (referenced 28.10.2021). Available: https://vue-chartjs.org

[17] V. Khorikov, Unit Testing Principles, Practices, and Patterns, Manning Publications, 2020. (Referenced 14.6.2022). Available: https://learning.oreilly.com/library/view/unit-test-frameworks/0596006896/

[18] R. Koubbi, A 2020 Guide to Python 2 vs Python 3, careerkarma.com, 13.7.2020, (referenced 12.1.2020). Available: https://careerkarma.com/blog/python-2-vs-python-3/

[19] C. G. Lasater, Design patterns. 1st edition, Wordware Pub., 2007, (referenced 16.1. 2022).

[20] H. Liang, X. Pei, X. Jia, W. Shen, J. Zhang, Fuzzing: State of the Art, New York: IEEE, IEEE transactions on reliability, 2018, Vol.67 (3), p.1199-1218, 2018.

[21] T. Linz, A. Spillner, Software Testing Foundations, 5th Edition, 5th Edition, Rocky Nook, 2021.

[22] B. Nice, What is a Programming Style Guide and why should you care, Medium.com, 25.7.2019, (referenced 4.11.2021). Available: https://medium.com/level-up-web/what-is-a-programming-style-guide-and-why-should-you-care-9019e51bb7ad

[23] L. Niven, Niven's Laws, N-Space, Tor Books, 1990 (referenced 15.11.2021).

[24] NPM Docs, npmjs.com, npm, Inc.. (Referenced 12.1.2022). Available: https://docs.npmjs.com

[25] M. Mijač, Evaluation of Design Science Instantiation Artifacts in Software Engineering Research, Varazdin: Faculty of Organization and Informatics Varazdin, Conference proceedings, 313–321, 2019

[26] I. Ottoway, Pycomm3 A Python Ethernet/IP library for communicating with Allen-Bradley PLCs, docs.pycomm3.dev, 2021, (Referenced 3.11.2021). Available: https://docs.pycomm3.dev/en/latest/

[27] T. Peters, PEP 20 – The Zen of Python, Python.org, 22.8.2004, (referenced 4.11.2021). Available: https://www.python.org/dev/peps/pep-0020/

[28] Postman API Platform, (referenced 5.1.2022). Available: https://www.post-man.com

[29] S. Potter, Fewer Labels than Points #769, Github, 20.11.2014. (Referenced 2.11.2021). Available: https://github.com/chartjs/Chart.js/issues/769

[30] Python. (Referenced 10.1.2022). Available: https://www.python.org

[31] Requests: HTTP for Humans™. (Referenced 6.1.2022). Available: https://docs.python-requests.org/en/latest/

[32] H.W.J. Rittel, M.M. Webber, "Dilemmas in a General Theory of Planning." Policy sciences 4.2 (1973): 155–169. Web, (referenced 2.1.2022).

[33] Robot Framework, Robot Framework ry. Available: https://robotframework.org

[34] B. Ross, Potential Decoupling Series Data from Labels #389, Github, 30.6.2014, (referenced 2.11.2021). Available: https://github.com/chartjs/Chart.js/issues/389

[35] G. van Rossum, B. Warsaw, N. Coghlan, PEP 8 – Style Guide for Python Code, Python.org, 1.8.2013, (referenced 4.11.2021). Available: https://www.py-thon.org/dev/peps/pep-0008/

[36] Teleste Description of the Problem and the commissioned software solution, Ap-pendix.

[37] Vue.js – The Progressive JavaScript Framework. Available: https://vuejs.org

[38] WebAssembly, MDN Web Docs, (referenced 14.1.2022). Available: https://deve-loper.mozilla.org/en-US/docs/WebAssembly

[39] Welcome to GQL 3 documentation!, (referenced 17.1.2022). Available: https://gql.readthedocs.io/en/stable/

APPENDIX 1: TELESTE DESCRIPTION OF THE PROBLEM AND THE COMMISSIONED SOFTWARE SOLUTION

”Junajärjestelmiä kehitettäessä käytössä ei ole koko järjestelmää. Matkustajainformaatiojärjestelmä integroituu junan muihin järjestelmiin erilaisten rajapintojen kautta. Tämän osajärjestelmän kehittämistä ja testaamista varten täytyy pystyä monitoroimaan ja kontrolloimaan kyseisiä rajapintoja. Nyt käytössä on hajanaisesti erilaisia työkaluja. Tässä on tarkoitus toteuttaa yksi monitorointi- ja simulointityökalu joka täyttää useiden projektien vaatimukset. Lisäksi tuodaan uusia ominaisuuksia erityisesti rajapinnan monitorointiin.”