Tampere University

Lassi Rintala

ARCHITECTURE DESIGN OF A CONFIGURATION MANAGEMENT SYSTEM

Master of Science Thesis
Faculty of Engineering and Natural Sciences
Examiners: Assoc. Prof. Reza Ghabcheloo
Prof. Hannu-Matti Järvinen
December 2020

ABSTRACT

Lassi Rintala: Architecture design of a configuration management system
Master of Science Thesis
Tampere University
Degree Programme in Mechanical Engineering, MSc (Tech)
December 2020

In this thesis, the main goal is to answer the following question: How to design an architecture for a software configuration management system that aims to easily distribute various files to multiple hosts, control software execution remotely on multiple hosts, validate various configuration items, provide access for the documentation of each configuration item, and be maintainable by software developers?

Software architecture has become increasingly relevant in the modern society where software can be found almost everywhere, and architectural decisions in the software design can have huge impacts on the quality attributes of the software. Designing architecture can be facilitated by utilizing existing patterns, and architectures can be evaluated using different methods.

Configuration management can be defined in many ways, mainly depending on the scale of the context. This thesis discusses configuration management being similar to system administration: modifying software configurations of computer systems. There are also various approaches to how configuring remote systems can be implemented: agentless or with a designated remote agent, imperative or declarative.

Configuration errors still remain as a major source of outages in computer systems. Configuration validation is a way to proactively prevent such errors from happening. Validations are usually implemented using various schemata to describe valid structures and values for configurations.

Based on the collected requirements and user stories of Visy Oy employees, a software configuration management system was designed to facilitate commissioning and maintenance of Visy systems.

The designed architecture of the application was evaluated using a lightweight version of decision-centric architecture review method. As the result of the evaluation, it was concluded that the majority of the architectural decisions were suitable for the purpose of the designed application.

An implementation of the software configuration management system was developed based on the designed architecture using C++ programming language with various open-source libraries and software components that already existed in Visy codebase.

The goal of this thesis was met, but the designed configuration management system still left some space for improvement in the future.

Keywords: software architecture, configuration management, configuration validation

TIIVISTELMÄ

Lassi Rintala: Konfiguraationhallintajärjestelmän arkkitehtuurin suunnittelu
Diplomityö
Tampereen yliopisto
Konetekniikan DI-tutkinto-ohjelma
Joulukuu 2020

Tämän diplomityön päätavoitteena on vastata seuraavaan kysymykseen: Miten suunnitella konfiguraationhallintajärjestelmä, jonka tavoitteena on helposti jakaa erinäisiä tiedostoja monille tietokoneille, hallita ohjelmistojen suoritusta monella tietokoneella, validoida konfiguraatioita, tarjota pääsy konfiguraatioiden dokumentaatioon ja olla ohjelmistokehittäjille muokattavissa?

Ohjelmistoarkkitehtuurista on tullut kasvavassa määrin merkityksellisempi aihe modernissa yhteiskunnassa, jossa tietokoneohjelmia löytyy lähes kaikkialta, ja arkkitehtuuripäätöksillä voi olla huomattavia vaikutuksia ohjelmistojen laatuominaisuuksiin. Arkkitehtuurin suunnittelua voi helpottaa hyödyntämällä olemassa olevia malleja, ja arkkitehtuureja voidaan evaluoida käyttäen erilaisia menetelmiä.

Konfiguraationhallinta voidaan määritellä monella tapaa riippuen enimmäkseen siitä, miten suuresta kontekstista on kyse. Tässä työssä konfiguraationhallinnalla tarkoitetaan samaa kuin järjestelmänhallinnalla: tietokonejärjestelmien ohjelmistojen konfiguraatioiden muokkaamista. Etäjärjestelmien hallintaan on olemassa eri menettelytapoja: agentiton tai erillisen agentin kera, imperatiivinen tai deklaratiivinen.

Konfigurointivirheet ovat yhä syynä suureen osaan tietokonejärjestelmien palvelukatkoksista. Konfiguraation validointi on ennakoiva tapa estää tällaisia virhetilanteita tapahtumasta. Validaatiot toteutetaan yleensä käyttäen erinäisiä skeemoja kuvaamaan kelpaavia konfiguraation rakenteita ja arvoja.

Visy Oy:n työntekijöiltä kerättyjen vaatimusten ja käyttäjätarinoiden perusteella suunniteltiin konfiguraationhallintajärjestelmä helpottamaan Visyn järjestelmien käyttöönottoa ja ylläpitoa.

Työssä suunnitellun sovelluksen arkkitehtuurin sopivuus käyttötarkoitukseensa arvioitiin käyttäen kevennettyä versiota päätöskeskeisestä arkkitehtuurikatselmoinnista. Katselmoinnin tuloksena selvisi, että suurin osa tärkeimmistä arkkitehtuuripäätöksistä oli ohjelmiston tarkoitukseen hyviä.

Konfiguraationhallintajärjestelmästä tehtiin toteutus perustuen suunniteltuun arkkitehtuuriin käyttäen C++-ohjelmointikieltä ja hyödyntäen osittain sekä avoimen lähdekoodin kirjastoja että Visyllä jo olemassa ollutta koodikantaa.

Työn tavoite tuli täytettyä, mutta suunniteltu konfiguraationhallintajärjestelmä jätti vielä tilaa tulevaisuuden parannuksille.

Avainsanat: ohjelmistoarkkitehtuuri, konfiguraationhallinta, konfiguraation validointi

PREFACE

First, I'd like to thank all my colleagues who have given their input to the requirements and participated in the evaluation of the system designed in this thesis. I also want to thank the executive manager of Visy Oy, Petri Granroth, for facilitating the writing process while I was also working full-time.

I would also like to thank all my friends and family who have supported me in many ways during my studies and in the process of writing this thesis.

Last but not least, I'd like to thank my supervisor Heikki Huttunen and examiners Hannu-Matti Järvinen and Reza Ghabcheloo.

In Tampere, 10th December 2020

Lassi Rintala

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

API	Application Programming Interface
ATAM	Architecture Tradeoff Analysis Method
C	a general-purpose programming language
C++	an object-oriented successor of C programming language
C4	a model for visualizing software architecture, stands for "Context, Container, Component, Code"
CSP	Constraint Satisfaction Problem
DCAR	Decision-Centric Architecture Review
GUI	Graphical User Interface
IDE	Integrated Development Environment
INI	an informal standard format for configuration files, stands for "initialization"
IP	Internet Protocol
OSI	Open Systems Interconnection
RDP	Remote Desktop Protocol
SFTP	SSH File Transfer Protocol
SSH	Secure Shell
STL	Standard Template Library
TCP	Transmission Control Protocol
UI	User Interface
UML	Unified Modeling Language
VCL	Visual Components Library
VPN	Virtual Private Network
WinRM	Windows Remote Management
wizard	synonym for setup assistant, a user interface type that presents a user with a sequence of dialog boxes that lead the user through a series of well-defined steps
WLAN	Wireless Local Area Network
XML	Extensible Markup Language
XSD	XML Schema Definition

1 INTRODUCTION

Managing configuration files of various software systems can be a tedious process, especially if you need to edit the files manually with a simple text editor application and take backups by simply remembering to copy-paste files manually. The task becomes even more tedious if there are multiple computers to be configured.

One way to make configuration management faster, less prone to human errors and overall more comfortable is to use configuration management software. Such software can, for example, automatically find and fix errors, implement version control, deploy the configuration across multiple hosts and create template configuration items to reduce repetitive work.

Visy Oy is a Finnish globally operating company established in 1994 specializing in automatic access and traffic control systems for the industry. Such systems are used in, for example, container terminals, ports, factories and border checkpoints. Visy systems consist of multiple computers and other hardware distributed across customer premises and most of the systems utilize computer vision technology to recognize, for example, vehicle license plates, container identification codes and labels of hazardous materials.

Software applications and services together with their configuration files define the behaviour of Visy systems, thus playing an important role in whether the system works as intended. The current situation is that the configuration files are edited with simple text editors without any syntactic or semantic validation, and the files are manually copied up to dozens of different computers, which leaves room for a multitude of human errors slowing down the system commissioning and maintenance process.

Figure 1.1 shows how the configuration files are currently viewed and edited in most situations, using Microsoft Notepad application that does not provide any syntactic or semantic validation for any text file format. It is hard for the user to know what aspect of the software functionality each parameter concerns and whether the user has given any valid values to the parameters. Some comments are included occasionally in the context of the parameter to give a vague description of its functionality. Only by starting the software that utilizes the configuration file can one see any validation for the configuration in the software log files: error texts are logged about incorrect data types, missing configuration items and sometimes also about other erroneous configurations.

Figure 1.2 shows how the file systems of various computers in the systems are accessed, using Remote Desktop Protocol (RDP). This is generally viewed as a tedious process of

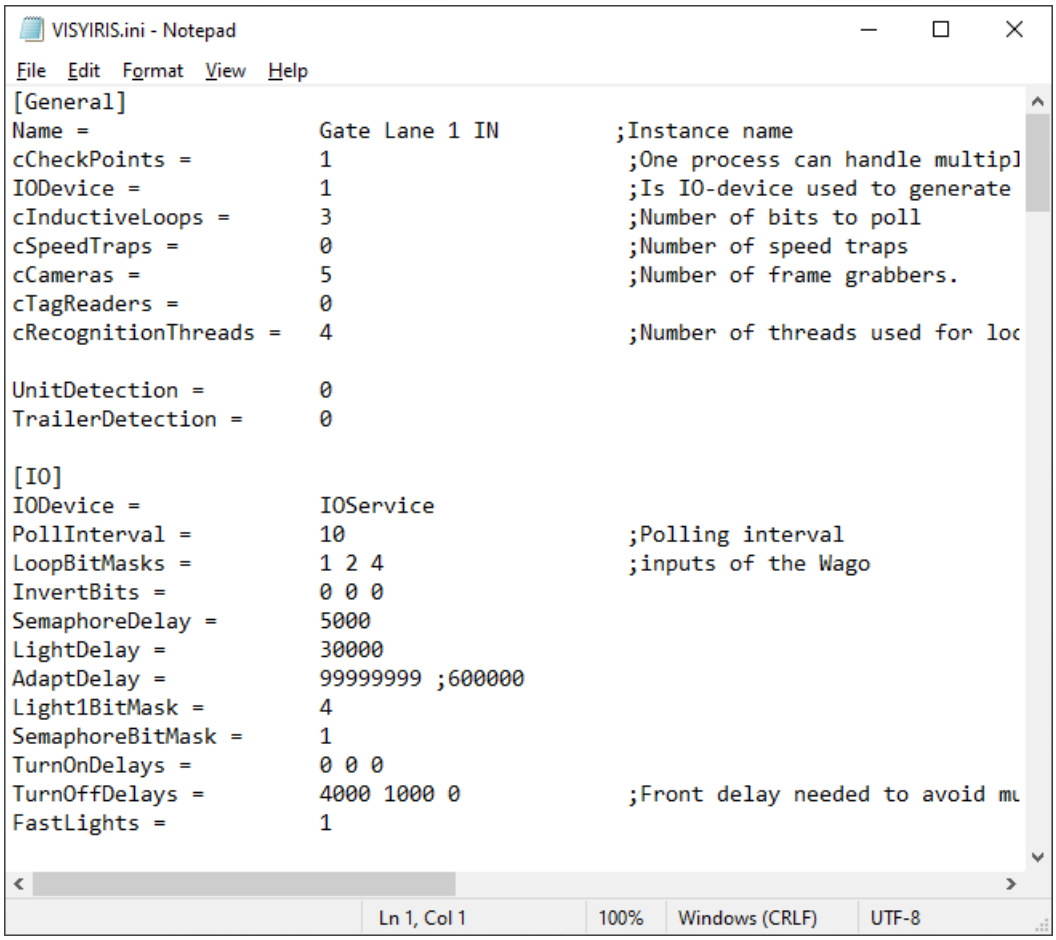

Figure 1.1. *Current way of viewing and editing the system configuration files.*

manual repetitive labour among employees. Currently, the following steps are required to access files in a system host:

1. Open a Virtual Private Network (VPN) connection to the remote network at customer premises.
2. Access a server in the network via Remote Desktop Protocol.
3. Access other computers in the network via RDP from the server.
4. Manually copy files between the personal computer and the remote system computers.
5. Manually restart all software that use the updated configuration files.

Earlier when the company systems consisted of a smaller amount of computers and had less parameters to configure, this was not considered an issue. Now that the systems ordered by a larger variety of customers have become more diverse and complex, demand for a more standard way of configuring systems has become more urgent to ensure better productivity in terms of system commissioning and maintenance.

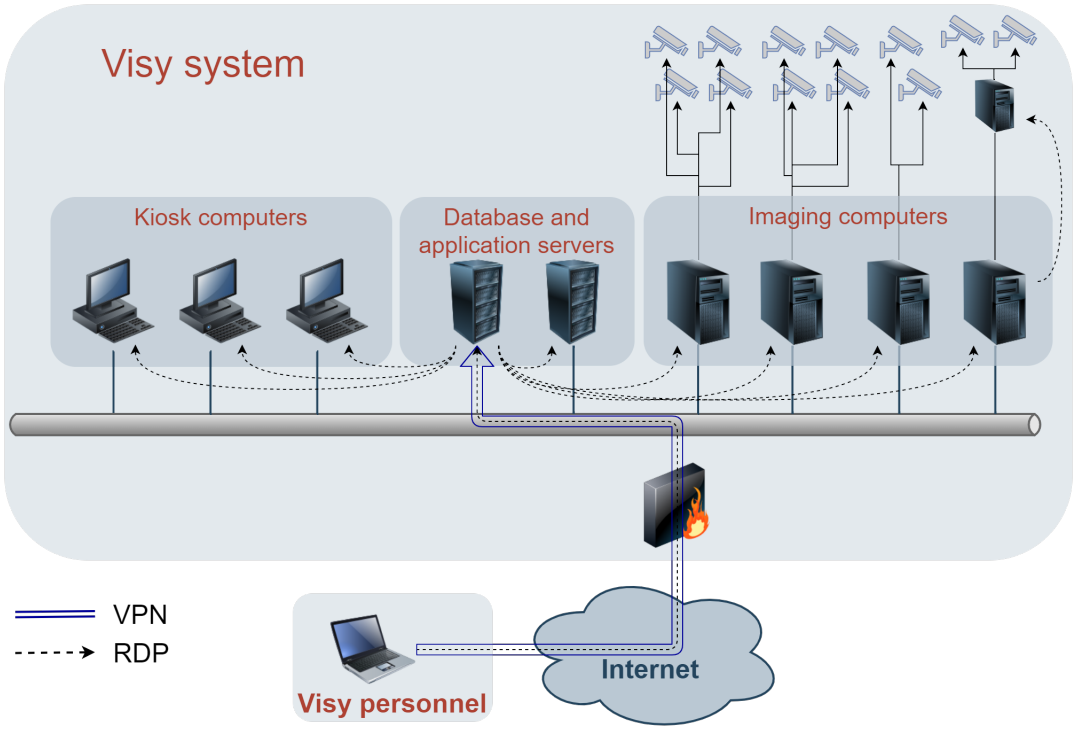

Figure 1.2. *Current system network topology and way of accessing hosts.*

The main goal of this thesis is to answer to the following question: How to design a software configuration management system that aims to easily

- distribute various files to multiple computers
- control software execution remotely on multiple computers
- validate various configuration items
- provide access for the documentation of each configuration item
- be maintainable by software developers?

Based on collected requirements and user stories of the company employees, a software configuration management system was designed and implemented to facilitate system commissioning and maintenance.

This thesis is structured as follows. Chapter 2 first gives an overview of what software architecture is, how it is used in software development and how architectures can be evaluated. Chapter 3 discusses the main concepts of software configuration management. Chapter 4 gives some motivation for configuration validation and perspective to how configuration items can be validated in practice.

Chapter 5 explains the details of how the software configuration management system was designed. The chapter also acts as the rationale for the designed architecture, as an integral part of the architecture documentation. Chapter 6 discusses evaluation of the designed software architecture using a lightweight version of decision-centric architecture review method.

Chapter 7 describes the implementation of Visy Configurator, how the designed architecture was applied as the basis of the implementation, how different aspects of the architecture aided or hindered the development and what kind of other challenges were encountered during the implementation.

Chapter 8 summarizes the thesis and introduces ideas for further development of the designed software system in the future.

2 SOFTWARE ARCHITECTURE

Software can be found almost everywhere in our current society and a need to create and maintain more complex systems faster and more cost-effectively than before seems to be one of the general goals in the industry. This is one of the reasons why software architecture is needed, to organize the development of these increasingly complex systems. [1]

This chapter discusses what software architecture is, how it can be designed and how it is applied in the actual implementation of the software. In this thesis generally, software architecture is mainly considered from an object-oriented perspective. This chapter also gives an overview of how a given software architecture can be reviewed using different methods.

2.1 Definition

Software architecture can be defined in multiple different ways, many of which have a lot of similarities between them. For example, ISO/IEC/IEEE 42010 standard defines architecture as

"fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution" [2].

Clements et al. define software architecture as follows:

"The software architecture of a computing system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both." [3]

Buschmann et al. have the following definition for software architecture:

"A software architecture is a description of the subsystems and components of a software system and the relationships between them. Subsystems and components are typically specified in different views to show the relevant functional and non-functional properties of a software system. The software architecture of a system is an artifact. It is the result of the software design activity." [4]

According to Perry and Wolf, one way to define software architecture is as a set of architectural elements: processing elements, data elements and connecting elements. Pro-

cessing elements transform the data elements, data elements contain the information used and transformed, and connecting elements serve to "glue" the architectural components together. [5]

As noted in the definitions above, software architecture mainly refers to the structural components of the concerned software system and the dependencies and interactions between those components. It describes the design of the system in a high level of abstraction, not necessarily addressing the lower level implementation details of the system. For example, coding style is not considered a part of software architecture. On the other hand, choosing a specific programming language or at least limiting the choice to only few, can be a part of a software architecture: having the architecture designed with classes and objects requires the programming language to be object-oriented, executing an interpreted language might make the software too slow for applications with real-time constraints, or choosing a rarely used language could be a disadvantage when looking for available third party solutions or when another developer with no knowledge of the language needs to maintain the code.

2.2 Modeling

There are different ways for modeling software architecture with various diagrams, each depicting different aspects of the software architecture. Modeling languages provide common means for documenting software architecture and communicating it to the stakeholders and other such parties interested in the architecture of the software. In this section, the following modeling languages will be discussed briefly: Unified Modeling Language (UML) and Context, Containers, Components, and Code (C4).

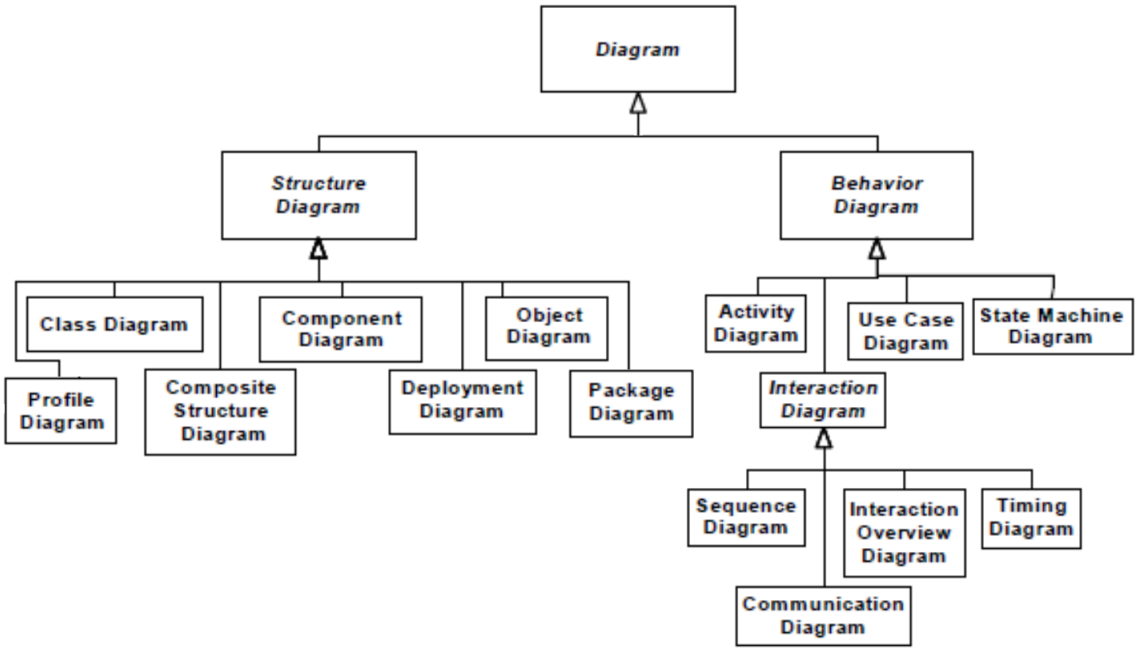

Figure 2.1. *UML diagram type categorization, adapted from [6].*

UML is a widely used standard modeling language for visualizing software architecture with diagrams. As shown in Figure 2.1, the standard contains two categories of diagrams: structural diagrams – describing the structure of the software – and behavior diagrams – describing the behavior and interactions between the structural components. The structural diagrams contain class diagrams, component diagrams, object diagrams, composite structure diagrams, deployment diagrams, package diagrams and profile diagrams. The behavior diagrams contain activity diagrams, state machine diagrams, use case diagrams, interaction overview diagrams, communication diagrams, sequence diagrams and timing diagrams. The aforementioned diagrams aim to provide software engineers means to design, analyse and implement a vast variety of different software architectures. [6]

Another alternative way of describing software architecture is C4. The C4 model describes software architecture in four different levels of detail and abstraction: context, container, component and code. As shown in Figures 2.2 and 2.3, the model is supposed to work like an image viewing or map application that you can zoom in and out to get diverse views with different levels of detail of the software architecture. [7]

System Context diagram provides the most abstract view of the architecture, only showing how the system interacts with its environment and what other actors the environment consists of. Container diagram shows what the system itself consists of, its high level building blocks, called containers, such as databases and user applications, and how these interact with each other and the system's environment. Component diagram then zooms into an individual container and shows what kind of components it consists of and how these components interact with each other and the surrounding containers. The last and lowest level in the C4 model is Code, which shows how an individual component is implemented. This implementation detail could be presented as, for example, a UML class diagram.

2.3 Patterns

Designing software architecture carefully can provide improved quality attributes for the software at hand. Such attributes include, for example, maintainability, performance and adaptability. One way to facilitate designing software architecture is to apply appropriate patterns in it. Patterns are reusable solution templates for commonly occurring specific types of issues that are faced when designing software. When reusable solutions are utilized to tackle recurring problems, the software developers need to spend less time on looking for their own solutions. [4]

This section describes three abstraction levels of patterns according to the definition provided by Meunier et al.: architectural patterns, design patterns and idioms. [4] These three levels could be thought of as analogous with more universally labeled levels in general planning and implementation: strategy, tactics and technique. The following subsections discuss each level of patterns in more detail.

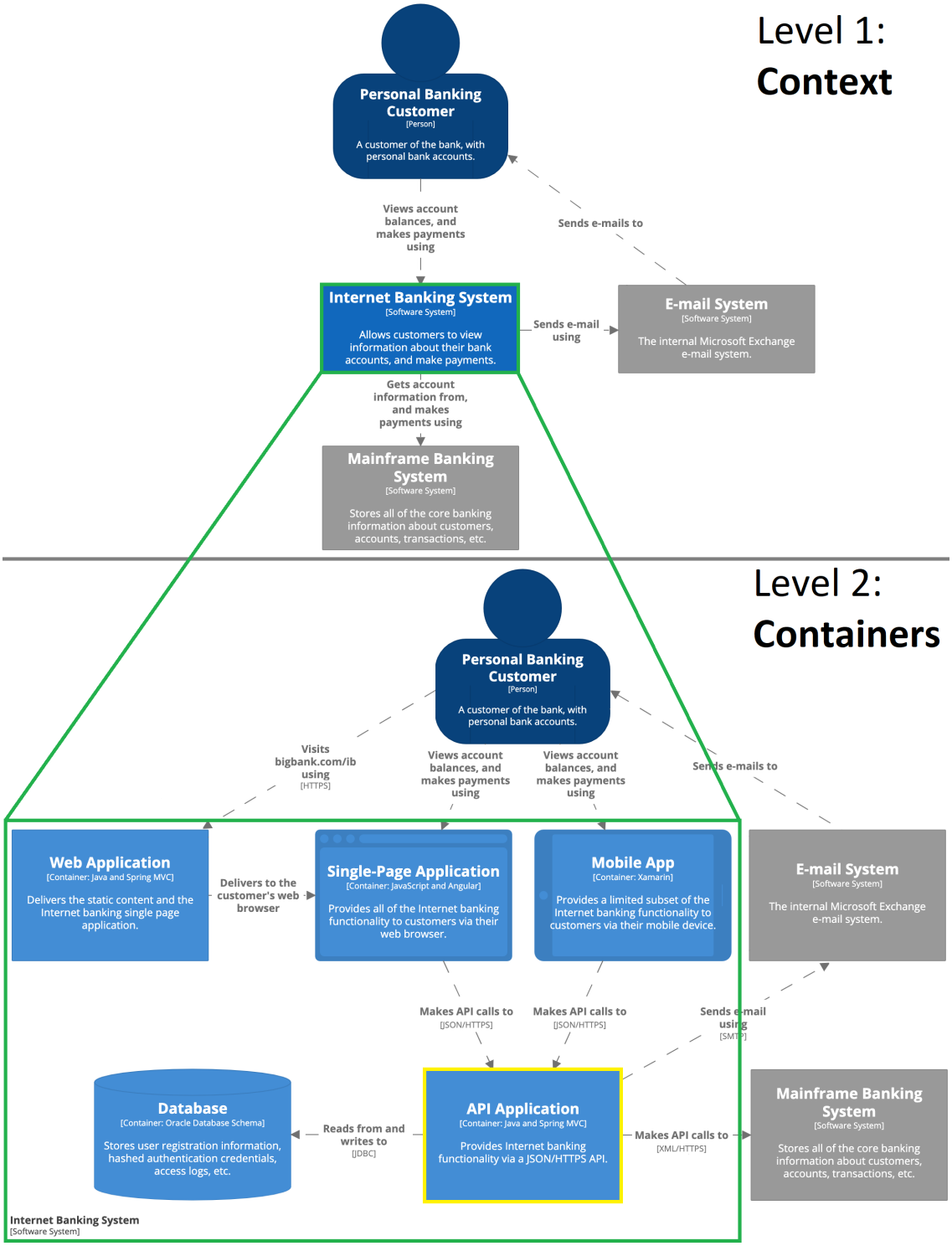

Figure 2.2. *An example of C4 diagram types Context and Containers, adapted from [7].*

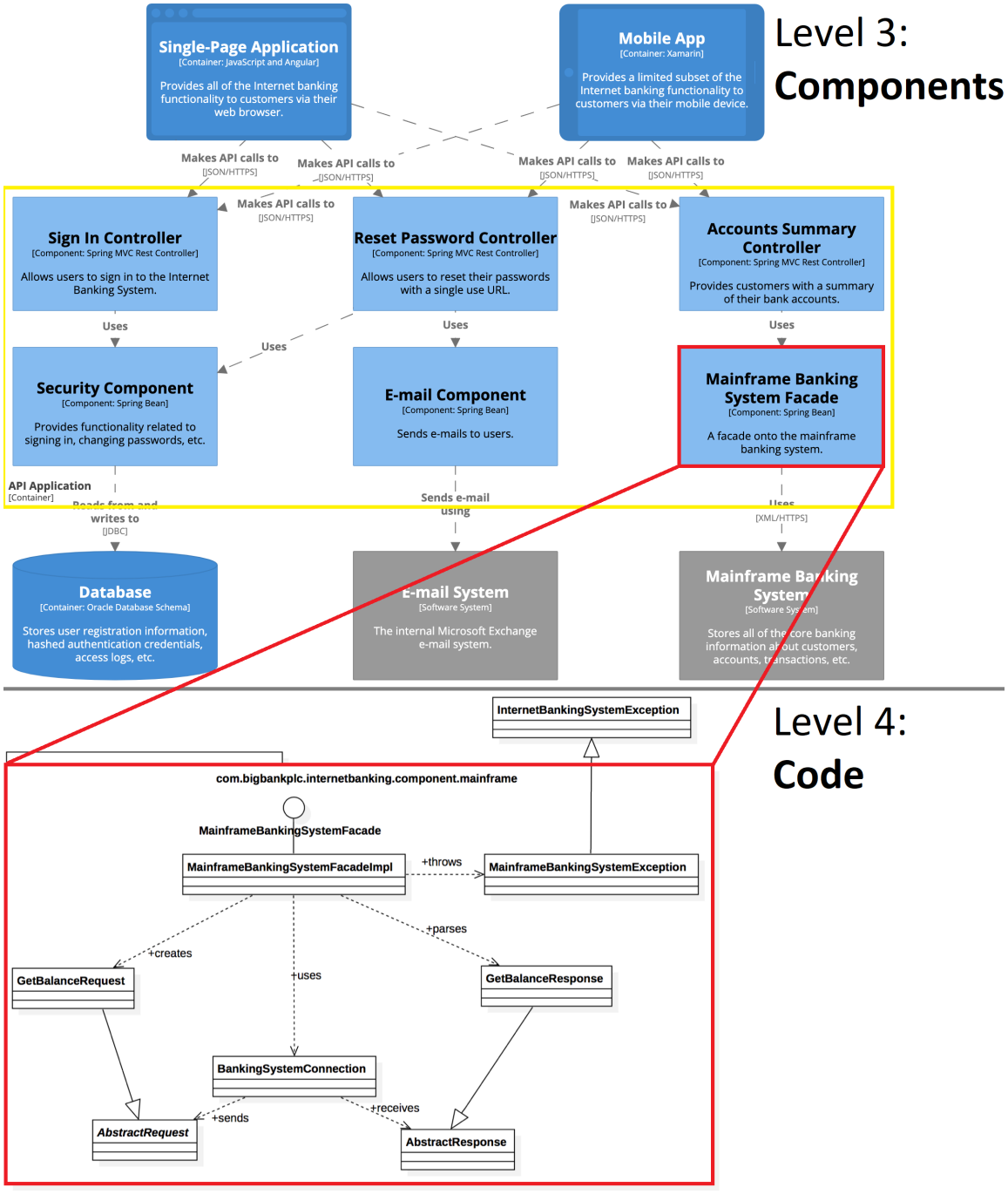

Figure 2.3. *An example of C4 diagram types Components and Code, adapted from [7].*

2.3.1 Architectural patterns

Architectural patterns are at the highest level of patterns. They represent template architectures for whole software applications by facilitating the specification of their fundamental structures. Such patterns include, for example, Layers pattern, Pipes and Filters pattern, Broker pattern, Model-View-Controller pattern and Microkernel pattern. [4] The following paragraphs discuss Layers architectural pattern in more detail as an example.

The Layers pattern helps decomposing complicated software components into smaller subtasks in different levels of abstraction [4], similar to what is shown in Figures 2.2 and 2.3 depicting the C4 model diagram types. Applying the pattern guides the architect to create several abstraction layers in the software components to keep them organized in a hierarchical manner. [4]

Figure 2.4 shows the main idea of Layers pattern. The layers in the pattern only interact with their adjacent layers. The topmost layer, Layer N, only uses the services of the layer right below it, Layer N-1. Layer N-1 in turn only uses the services of the layer right below it, Layer N-2, and so on. This is supposed to, for example, prevent late code changes rippling through the system, make parts of the system exchangeable and more testable, and make the layers reusable in other solutions as well. [4]

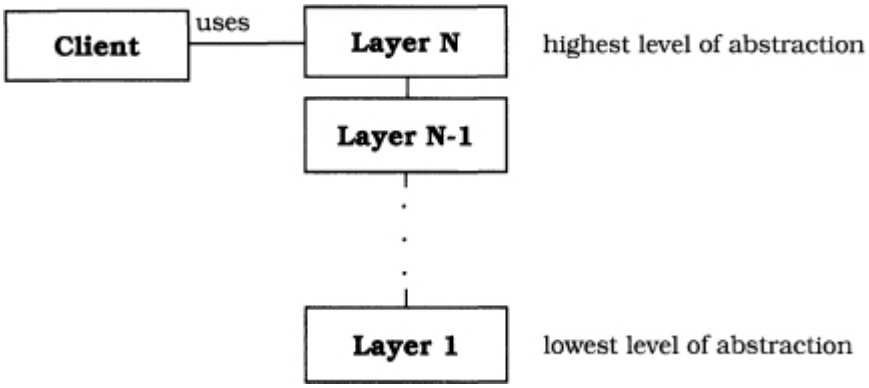

Figure 2.4. *Different levels of abstraction in Layers pattern. [4]*

Some frequently used examples of layered architectures are network protocol stacks, especially the Open Systems Interconnection (OSI) model and the prevalent Transmission Control Protocol (TCP) / Internet Protocol (IP) stack. The OSI model is a conceptual standard model whose purpose is to *"provide coordination of standards development for the purpose of systems interconnection"*. It strictly conforms to the Layers pattern and consists of seven different abstraction layers: Application, Presentation, Session, Transport, Network, Data Link and Physical. [8]

TCP/IP does not strictly conform to the OSI model nor the Layers pattern [4, 9]. This makes TCP/IP more of a *Relaxed Layered System* in which each layer can use the services of all layers below it, not just the one right below it [4]. Figure 2.5 shows an overview of the TCP/IP stack layers where both Transport layer and Internetwork layer are used by the Applications layer. This partial skipping of Transport layer can be seen usually

when configuring an application to establish a connection to another host: you do not only specify the TCP port, but you also specify the IP address at the application level.

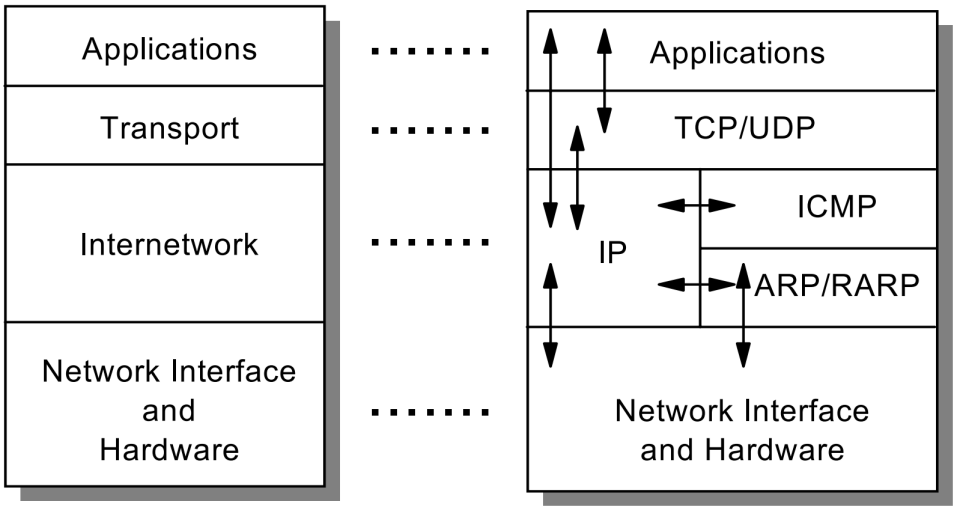

Figure 2.5. *An overview of the layers included in the TCP/IP protocol stack. [9]*

2.3.2 Design patterns

Design patterns are at the middle level of patterns, smaller in scale than architectural patterns, but larger than idioms. [4]

In object-oriented programming, design patterns can be divided into three categories: creational patterns, behavioral patterns and structural patterns. Creational patterns decouple the system from how its objects and classes are created, initialized and configured. Behavioral patterns facilitate executing various algorithms and assignment of responsibilities between objects. Structural patterns make it easier to compose larger structures from smaller classes and objects. [4, 10] Another way to divide the design patterns is to put them into five categories: Structural decomposition patterns, organization of work patterns, access control patterns, management patterns and communication patterns. [4]

One example of such design pattern is the Interpreter pattern, show in Figure 2.6, that is a behavioral pattern. The Interpreter pattern can be used to interpret sentences of a language as abstract syntax trees and to evaluate them. The pattern is mostly applicable to simple languages, because for complex languages the class hierarchy becomes too large to be manageable. [10]

For example, a boolean statement `(2 * x) < (y + 10)` could be interpreted as an abstract syntax tree, shown in Figure 2.7. In this case `Client` could parse the given statement and build the abstract syntax tree as instances of `TerminalExpression` and `NonterminalExpression`. `x` and `y` are parts of `Context` that `Client` passes to the `Interpret` method, starting a recursive traversal of the tree structure where each `NonterminalExpression`

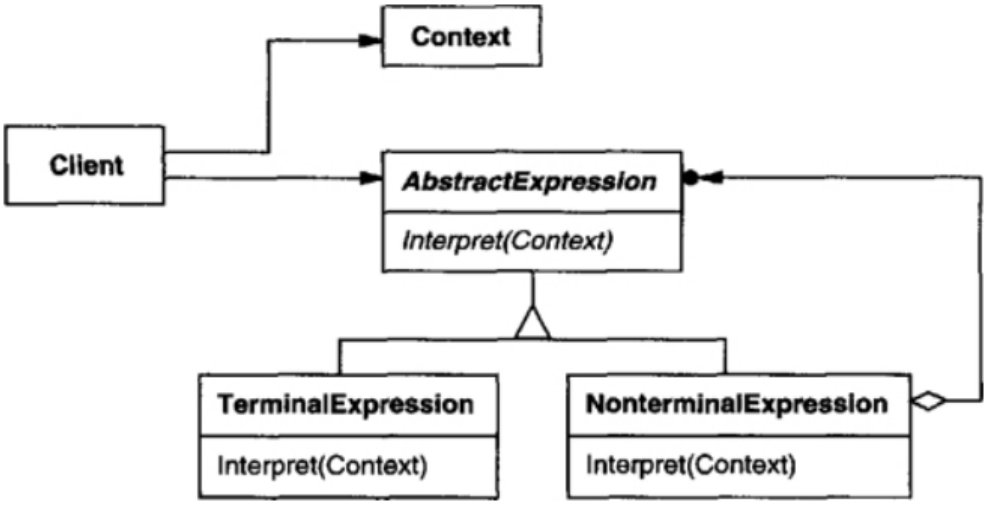

Figure 2.6. *A class diagram of the Interpreter pattern. [10]*

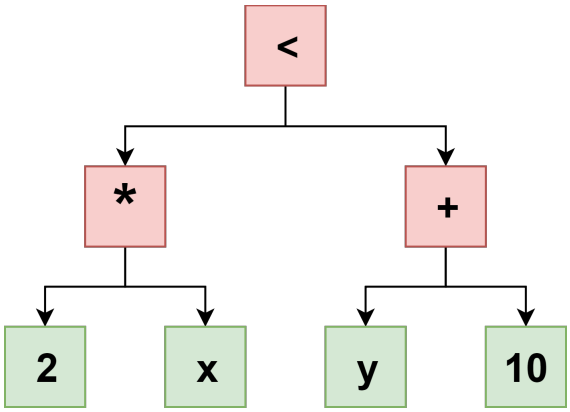

Figure 2.7. *Graphical representation of an abstract syntax tree. Red nodes represent instances of NonterminalExpression and green ones instances of TerminalExpression.*

calls `Interpret` for each of its child `AbstractExpression` to get their values in the given `Context` to be able to eventually evaluate its own value.

2.3.3 Idioms

Idioms are at the lowest level of patterns: They are patterns specific to programming languages, for facilitating memory management, object creation, naming and source code formatting for readability, efficient use of specific library components and so forth. [4] One example of such pattern in C++ is the erase-remove idiom, example usage of which is shown in Program 2.1, that just erases elements from an Standard Template Library (STL) container. [11]

```cpp
#include <vector>
#include <algorithm>

int main() {
  // Initialize the vector with integers from 0 to 9
  std::vector<int> numbers = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

  auto isOdd = [](int value) {
    return (value % 2 != 0);
  };

  // Use the erase-remove idiom to remove all odd numbers from the vector
  numbers.erase(
    std::remove_if(numbers.begin(), numbers.end(), isOdd),
    numbers.end()
  );

  // Now the vector only contains numbers 0, 2, 4, 6, 8
}
```

Program 2.1. *Example usage of the erase-remove idiom.*

2.4 Evaluation methodology

To be sure that the designed architecture fulfils the requirements set by the stakeholders and is overall considered acceptable, it needs to be evaluated: The evaluation confirms good solutions, draws attention to potential problems and helps to better understand the system. Fixing fundamental design errors proactively can save a lot of time, effort and money. [12]

The existing literature seems to use for example words *review*[13], *evaluation*[14], *analysis*[15] and *validation*[16] for naming similar processes of evaluating software architecture. In this thesis, mainly the word *evaluate* is used to describe such process.

To conduct an architecture evaluation, a clear description of the architecture's main features is required. Depending on the used evaluation process, also different types of stakeholders are required as reviewers, and they need to be thoroughly acquainted with the architecture at hand. Last but not least, an architecture evaluation requires time and effort[12, 17, 18], which can be considered an excuse for not conducting such tedious evaluations.

There are many different ways to evaluate software architecture, most of which are quite time-consuming and thus not very convenient, especially for smaller organizations. [13, 16] The evaluation methods themselves can be evaluated as well: Some methods fit better for specific domains, whereas some only evaluate a narrower set of quality attributes, and some require different amount and type of reviewers. Thus, it is important to choose an appropriate evaluation method based on the given architecture, the purpose of its

evaluation, and available resources. [14, 17, 18]

Software architecture evaluation approaches can be separated into three categories: checklist-driven, scenario-based and decision-centric. Checklist-driven methods use a set of prepared questions that will guide the architects during the evaluation and lead to exploration into the architecture being evaluated. In scenario-based methods, scenarios describe specific interactions between the user and the system. Using this approach, the architecture is tested against scenarios associated with the quality attribute requirements for the system. Decision-centric methods review, analyze and record the rationale behind the architecture and design decisions made by the project members. The collected decisions are evaluated against the quality attribute requirements for the architecture. [16]

One example of a prevalent scenario-based evaluation method is Architecture Tradeoff Analysis Method (ATAM), shown in Figure 2.8, that concentrates on assessing any quality attributes of the given software architecture. The major goals of ATAM are to *"elicit and refine a precise statement of the architecture's driving quality attribute requirements, a precise statement of the architecture design decisions and evaluate the architectural design decisions to determine if they satisfactorily address the quality requirements"*. While being a very thorough and formal method of architectural analysis, ATAM is also heavyweight, taking usually at least two days with multiple stakeholders. [15]

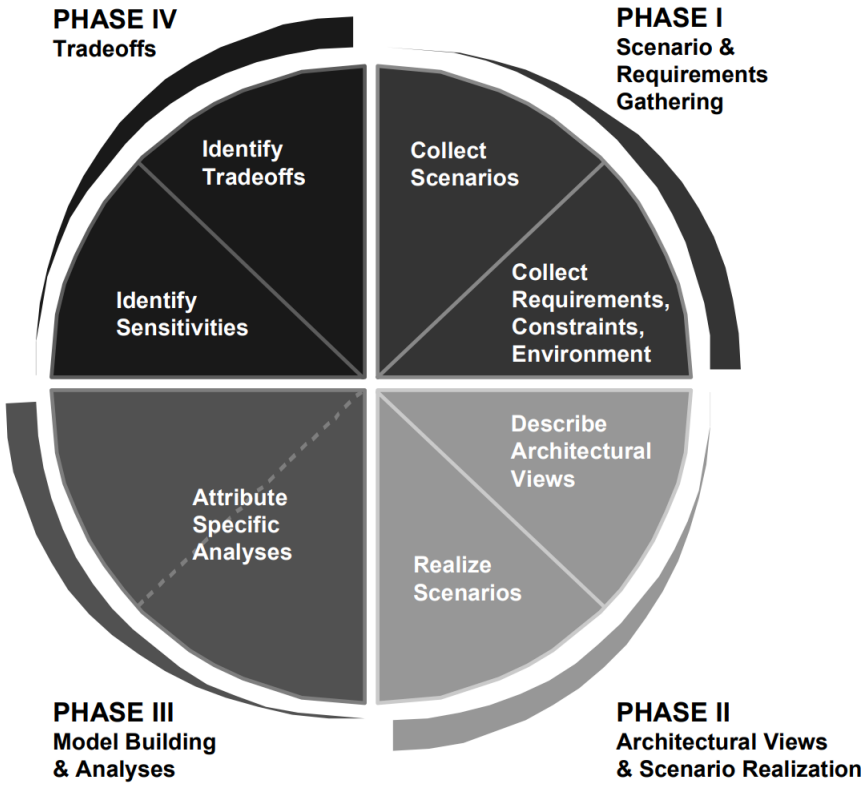

Figure 2.8. *Steps included in the ATAM process. [15]*

Decision-Centric Architecture Review (DCAR) is a relatively new decision-centric method for evaluating software architectures, published in 2014. It is supposed to be more

lightweight compared to earlier scenario-based methods, and can be conducted in less than five person-days. The goal of the method is to determine the soundness of the architectural decisions made by the architect. [13]

Inputs for the method include requirements, business drivers and architectural design. Outputs of the method are risks, issues and thorough documentation of the evaluated decisions and their decision forces. [13]

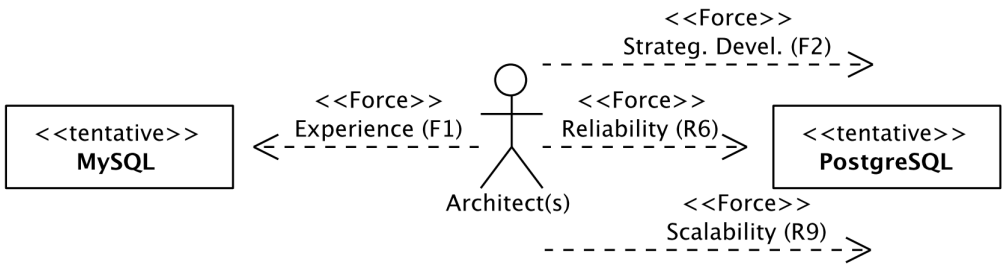

Figure 2.9. *An example of decision forces affecting the architect, adapted from [19].*

A force, in the context of DCAR, is basically anything that has a potential non-trivial impact of any kind on an architect when making decisions. A force might be, for example, a traditional requirement, the expertise of the development team, or any business or project constraint. Figure 2.9 shows an example of how different forces can affect the architectural decisions made by the architect. Forces may have different magnitudes and directions that may result in different decisions. [13, 19]

DCAR process consists of nine separate steps. The first step, *Preparation*, includes setting a date for the DCAR session, preparing required documents to be used during the session and letting the reviewers inspect the documents. The required documents include the management presentation the architecture presentation [13]

The second step is *DCAR introduction* where the DCAR method is introduced to all participants of the evaluation session. This introduction includes describing the DCAR steps, the scope of the evaluation, possible outcomes and participant roles and responsibilities. [13]

Step three, *Management presentation*, includes presenting the management presentation prepared in step one. The purpose of this step is to let the reviewers take note of business-related decision forces that should be taken into consideration during the evaluation. The reviewers may also ask questions during this presentation to elicit additional forces. [13]

In step four, *Architecture presentation*, the lead architect presents architecture presentation that was prepared in step one. The goal of this step is to give all participants a good understanding of the architecture. The presentation should be highly interactive, and the review team including other participants ask questions to complete and verify their understanding of the system. During this step, the reviewers identify more forces and architecture decisions, and revise the ones identified earlier. [13]

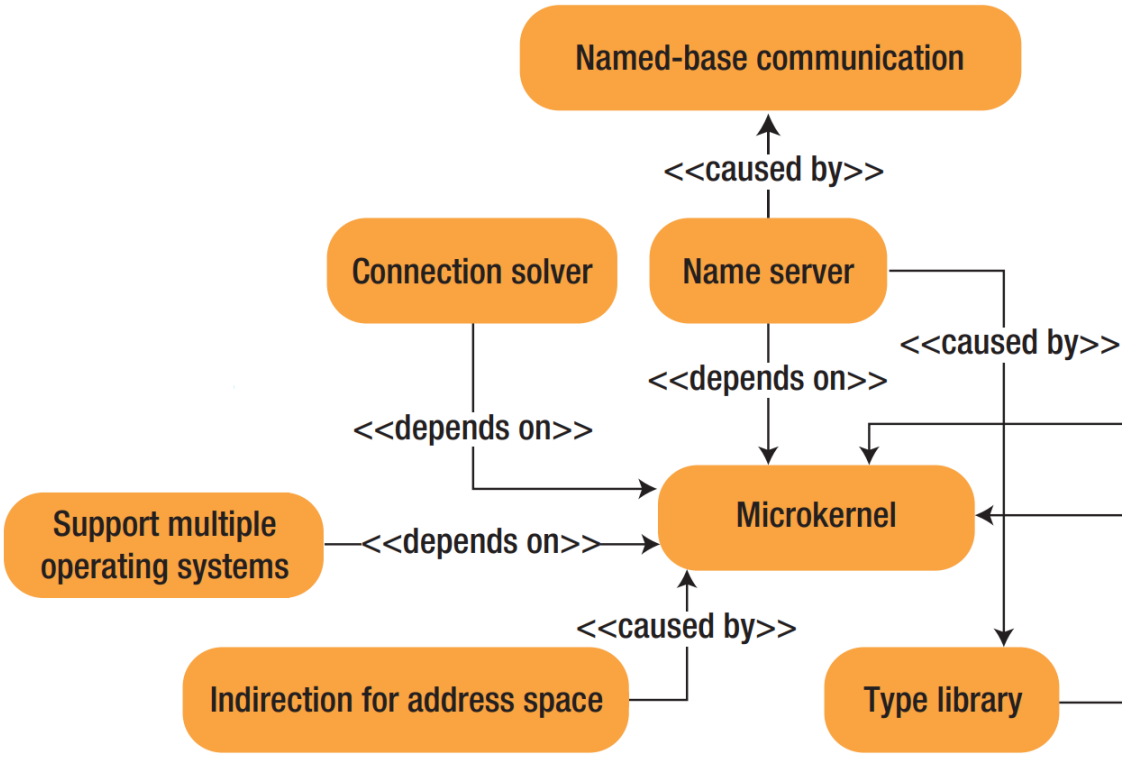

Figure 2.10. *An excerpt from an example relationship diagram created during a DCAR session, adapted from [13].*

The goal of step five, *Forces and decisions completion*, is to clarify the identified architecture decisions and their relationships, and complete and verify the identified forces relevant to these decisions. A relationship diagram – that is constantly revised during the next steps as well – is created to document the relationships and support their visualization. Figure 2.10 shows an excerpt from an example of such relationship diagram where architecture decisions, presented as rounded boxes, are connected to other related decisions with arrows. Depending on the type of relation, a text "caused by" or "depends on" is used in association with the arrow. These relationships help estimate the importance of each decision and are also helpful for understanding which decisions must be considered as decision forces for other decisions. [13]

In step six, *Decision prioritization*, all the identified architecture decisions are prioritized by the reviewers according to how important they think the specific decisions are. The prioritization is conducted as a vote: Each participant gets 100 points to freely distribute over the decisions. After voting, the points given for each decision are summed up and the rationale behind each person's rating is discussed. The decisions that get the most points are qualified to the next steps for documentation and evaluation.

Step seven, *Decision documentation*, is where the qualified decisions get documented in more detail. Figure 2.11 shows an example of how the documentation of one architecture decision could look like. The document describes the solution itself, considered alternative solutions, and forces in favor of and against the decision.

In step eight, *Decision evaluation*, the decisions are evaluated, starting from the highest-

Name	Redundancy of controllers			
Problem	The application should run even if the server fails			
Solution or description of decision	The system is deployed to two servers: one is active, the other one is inactive. The active server provides all system services, while the passive one is running in the background. When the active server fails, the inactive server becomes active. During the switch over, the active server tries to update the passive one to make sure that it has the same data and status. Both servers have an identical software configuration. This solution follows the *Redundant Functionality Pattern*.			
Considered alternative solutions	Apply the *Redundancy Switch Pattern*: Both servers are active; external logic is used to decide which output is actually used in the control. In this case, cyclic data copying could be avoided. However, applying this solution would require major modifications to the system. Even though availability would be increased, it would also cause additional costs. The customers are not prepared for paying more for higher availability. Additionally, the external logic component could become a potential single point of failure. Therefore, this alternative was discarded.			
Forces in favor of decision	• Easier to implement than the alternative solution • Scales easily to versions where redundancy is not used • No additional costs			
Forces against the decision	• Slower switch over time than the alternative would have • Hard to offer higher availability than the current 99.99%			
Outcome	Green	Yellow	Yellow	Red
Rationale for outcome	Current solution seems to be ok.	I am concerned about the slow switch over time.	Widely accepted solution. Availability might become a problem in the future.	We should really reconsider this decision, as the next release is likely to have higher availability requirements.

Figure 2.11. *An architecture decision documentation example. [13]*

priority decision. All participants decide by voting whether the forces in favor of the decision outweigh the ones against it, and consequently decide whether the respective decision is good, acceptable, or has to be reconsidered. The evaluation outcome and its rationale are filled in the documentation, as shown in Figure 2.11. [13]

Finally in step nine, *Retrospective and reporting*, an evaluation report is compiled from the artifacts created during the DCAR session. In a retrospective meeting after the DCAR session, the report is discussed with the architect for verification and eventually refined by the review team. [13]

3 SOFTWARE CONFIGURATION MANAGEMENT

This chapter gives an overview of what configuration management is, what different types of configuration management there is, what kind of activities it usually consists of and how it is utilized in the context of system administration.

3.1 Definition

Software configuration management can be interpreted as a process that mainly concerns either system administration or software source code management, or both of them. System administration involves tasks such as deploying, updating and configuring software, managing access privileges and managing network security on computer systems. Software source code management – also known as version or revision control – consists of tasks involved in software source code versioning and collaboration between developers. In the context of this thesis, software configuration management is considered to only include system administration tasks.

IEEE standard 828-2012 defines configuration management in systems and software engineering as

"a discipline applying technical and administrative direction and surveillance to: identify and document the functional and physical characteristics of a configuration item, control changes to those characteristics, record and report change processing and implementation status, and verify compliance with specified requirements" [20]

The U.S. Air Force's Software Technology Support Center provides a graphical representation of what elements configuration management consists of, shown in Figure 3.1, and uses the following definition:

"Configuration management (CM) is the process of controlling and documenting change to a developing system. It is part of the overall change management approach." [21]

Arundel, who teaches in his book how to use Puppet configuration management software, defines configuration management simply as the process of installing software on a computer and configuring the software with appropriate preference values. [22]

Red Hat, a company that provides Ansible configuration management software, defines configuration management as follows:

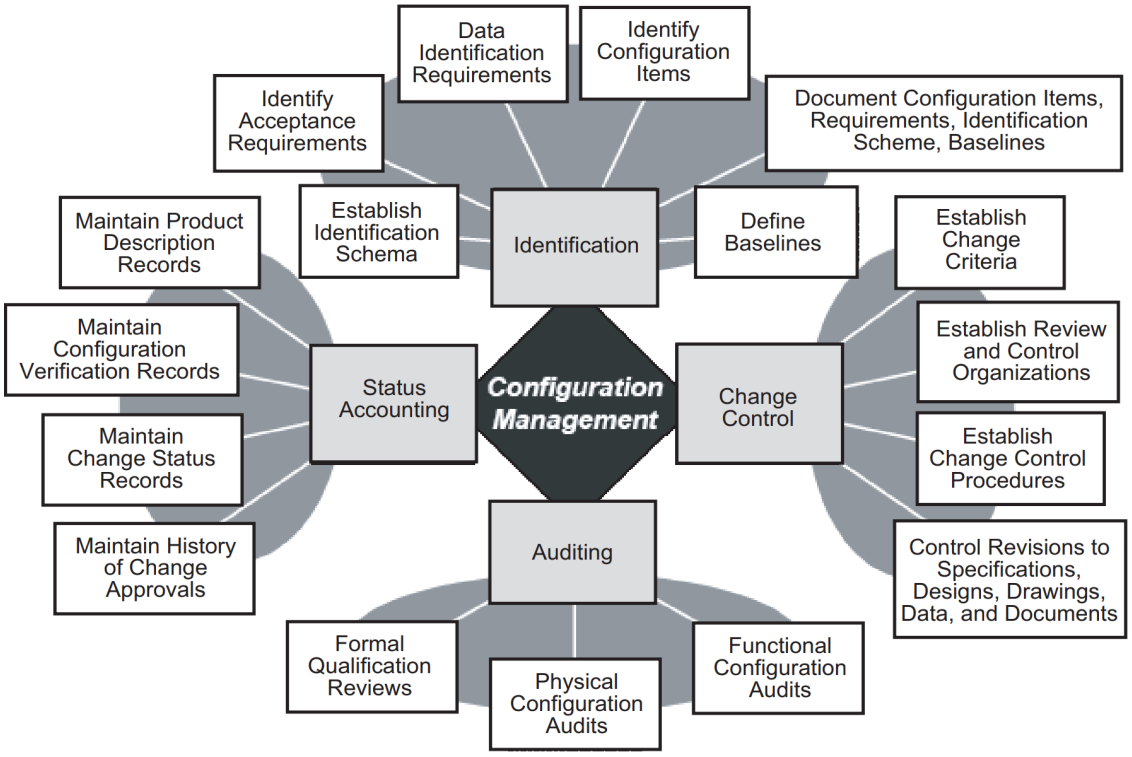

Figure 3.1. *The major elements of configuration management. [21]*

"Configuration management is a process for maintaining computer systems, servers, and software in a desired, consistent state. It's a way to make sure that a system performs as it's expected to as changes are made over time." [23]

The main differences between the aforementioned definitions seem to be the scope: They all talk about configurations and their management but some define configuration management as a wide field of different management related tasks whereas some narrow it down to a more simple process.

The configuration management system that is designed in this thesis most closely conforms to the definitions provided by Arundel and Red Hat. The system is supposed to facilitate deploying software packages and configuring them appropriately to meet the requirements set for a specific system that is installed for a customer.

3.2 Different approaches

The main motivation for using a dedicated configuration management tool for commissioning and maintaining software systems is in most cases the pursuit of efficiency and scalability, which can be achieved by automation. During the past years when DevOps as a methodology has become more relevant and mature, a variety of tools has been developed to automate different tasks including building, deploying and configuring software. This section discusses the different prevalent approaches to tackling the problem of tedious and error-prone manual software configuration management.

One way to categorize approaches to software configuration management is to divide them into ones that use a separate software agent on each host and ones that do not. Agentless systems use existing transport mechanisms such as Secure Shell (SSH) or Windows Remote Management (WinRM) that are already built in the operating systems whereas ones that require an agent use some custom protocol of their own. [24] Specifically, when talking about an agent, it usually means a software service daemon that is dedicated to only act as a part of the used configuration management system. For example, an SSH server daemon – that actually is an agent itself – is not usually considered a configuration management agent, because it is used universally for a variety of tasks. [25]

Both approaches have their advantages and disadvantages, though agentless is usually perceived as the better solution. Because agentless systems do not require the installation of a dedicated agent on each managed host, the configuration management system infrastructure itself is, in terms of data transport, very minimal and thus easier to manage. The prevalent protocols and their implementations for agentless systems are also very widely used and critically reviewed, which in turn guarantees, for example, better security and reliability. Also, when updating a configuration management system that uses agents, the agents themselves scattered across the infrastructure need to be updated as well, which can be considered an extra computational overhead and sometimes might even require manual work. [24]

Another way of categorizing approaches to software configuration management is to declare them as either imperative or declarative. In imperative configuration management systems, the user focuses on telling the system *how* to change the configuration to reach a specific state. While using imperative systems is more like writing procedural code, in declarative system the user just describes the desired state, focusing on *what* the configuration should eventually look like and let the configuration management system figure out how to reach that state. [26, 27]

Generally imperative configuration management systems are more expressive compared to declarative systems, which consequently allows the user to have more precise control over the management process. One downside of the imperative approach is that the user needs to be aware of the state of the managed hosts before making changes, whereas in the declarative approach the user does not need to know the prior state, because the system takes care of appropriate modifications to reach the declared state. When using an imperative system, neglecting the prior state of the host may cause divergence in the configuration from what the user originally intended, also known as configuration drift. [26, 28]

Programs 3.1 and 3.2 show a side-by-side comparison how a similar configuration can be achieved using different approaches. Program 3.1 shows how Apache can be installed and enabled using Chef, that is an imperative configuration management system, whereas Program 3.2 shows how the same thing can be done using Puppet, a declarative system. [29] As can be seen from these short examples, with Chef you issue commands

using `do` statement whereas with Puppet you only declare what you want the system to have in the end, using statements such as `ensure`.

```
case node[:platform]
 when 'ubuntu', 'debian'
  apachename = 'apache2'
 when 'redhat', 'centos'
  apachename = 'httpd'
end


package 'Install apache2' do
 package_name apachename
end


service apachename do
 supports :status => true
 action [:enable, :start]
end
```

Program 3.1. *Chef example of installing and enabling Apache, adapted from Example 3 in [29].*

```
class apache2 {
 if $::osfamily=='RedHat'
 {
  $apachename = 'httpd'
 }
 elseif $::osfamily=='Debian'
 {
  $apachename = 'apache2'
 }

 package {
  'apache'
  name => $apachename,
  ensure => 'present',
 }

 service {
  'apache-service':
  name => $apachename,
  enable => true,
  ensure => 'running',
 }
}
```

Program 3.2. *Puppet example of installing and enabling Apache, adapted from Example 3 in [29].*

3.3 Version control systems

Even though version control is mainly used in software source code management, it is also an important part of system administration. Version control systems enable their users to, for example, collaborate better with each other and roll back to an earlier version of the configuration if something goes wrong with the new one, as a way of implementing backups and disaster recovery. Such systems include, for example, Git[30], Subversion[31], Mercurial[32] and Concurrent Versions System[33].

The collaboration benefits of using a version control system include, for example, seeing who changed what, when and why, resolving and merging modification conflicts, and separating the configurations into stable and development branches. Being aware of

prior changes made in the system and their authors helps the person, whoever is going to make further changes or just trying to understand the system, contact the right people in case of any encountered issues, for example.

One way to categorize version control systems is to separate them as centralized systems, shown in Figure 3.2, and distributed systems, shown in Figure 3.3. In centralized systems, there is a single central copy or repository of your project where users commit their changes. In distributed systems, there is strictly speaking no main or master repository, but instead all cloned repositories act as equals: they all include the whole version history of the project and all its metadata. Despite the fact that the system is distributed, a repository in such system can be – and usually is – configured to act as a master repository through which all changes between different users are shared. Also other types of workflows are supported by distributed systems. [30]

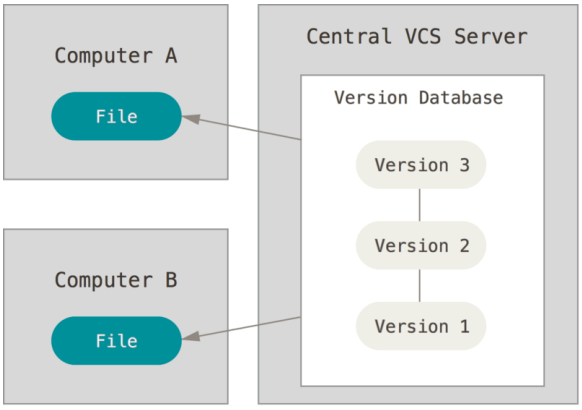

Figure 3.2. *A visualization of a simple centralized version control system. [30]*

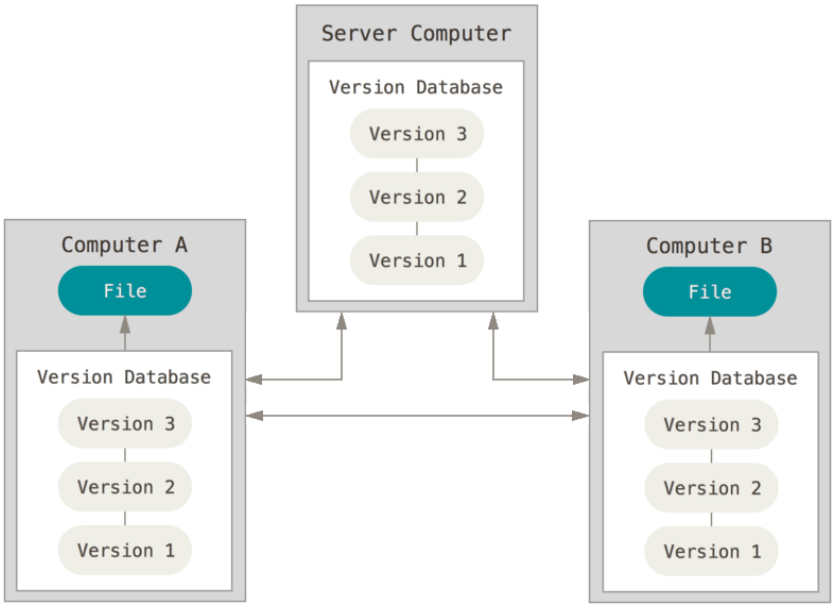

Figure 3.3. *A visualization of a simple distributed version control system, adapted from [30].*

4 CONFIGURATION VALIDATION

Testing the provided products and services is usually an important part in the quality assurance process of any organization, potentially saving significant amounts of both time and money [34] in the long run. Such tests may include, for example, usability testing, smoke tests, stress tests, integration tests, unit tests or any other kind of test. In contrast to reactive post-mortem issue resolving, proactive pre-deployment issue resolving by testing is usually the desired solution – and, in some cases, the only acceptable solution. For example, in many medical applications, malfunctioning is never allowed to occur.

Also data can be tested for its correctness, be it data produced and exchanged within the organization, or data provided from or to another organization. Executing such data validations manually by scanning through the data can be very tedious and error-prone, and automating the process usually makes it a lot more accurate, fast and efficient.

Configuration validation is a subset of data validation that focuses on validating data that is used as a configuration item in a system. More specifically, Huang et al. define configuration validation as *"the process of explicitly defining specifications and proactively checking configurations against those specifications to prevent misconfigurations from entering production"* [35].

Misconfiguration has historically been the prevalent cause of errors in computer systems [36] and still remains a major source of service outages [37, 38, 39, 40, 41, 42]. Many of such outages and other defects could be avoided by correctly validating configurations before deploying them into production [35]. Reactive configuration issue solving can also be made more efficient by using appropriate tools [43], but this chapter concentrates on the proactive side.

In most parts, this chapter and the whole thesis narrows down the definition of data used in validations to just plain text files, meaning all of the file contents are supposed to be interpreted as text. The following sections elaborate further on what configuration validation is, how it can be implemented and how using it can prevent more significant errors from happening.

4.1 Validation types

Data validations can be categorized in several different ways. Such categorizations may be done by separating validations into syntactic and semantic validations, validation levels, validation constraint types, check sums, and various other validation types. This section discusses what kinds of commonly used methods there are for validating configuration data.

4.1.1 Syntax and semantics

When validating text for its correctness according to the given language, two main subsystems exist: syntax and semantics. Chomsky defines syntax as *"the study of the principles and processes by which sentences are constructed in particular languages"* [44]. In other words, syntax is all about the correct order of words, symbols and signs within sentences.

In contrast to syntax, semantics can be defined as *"the study of the relationships between symbols or signs such words, phrases, sentences, and discourses, and what these elements mean or stand in for:— their denotations and senses"* [45]. Semantics is about the literal meaning of the given text, checking whether it can actually be interpreted rationally to have meaningful content.

Syntax and semantics do not only apply to human languages, but also, for example, to programming languages and communication protocols. For example, assigning an integer value to a parameter that represents the first name of a person

```
FirstName = 123
```

is syntactically correct, but semantically incorrect, because an integer is not a valid name for a human. In contrast to that,

```
FirstName: John
```

might be semantically correct, but syntactically incorrect, because it does not follow the syntax rule of using an equality sign to state an assignment.

4.1.2 Data types and constraints

At the lowest levels of semantic validation, when we consider the value of a single parameter – that is already syntactically correct – we need to check whether the value has correct data type. Data type of a parameter value can be for example string, single character, integer, decimal number, boolean or an array of any aforementioned types.

In a plain text, file everything can be interpreted as text, so the value of a parameter can

always be interpreted as string type. Thus, string-typed parameters may not need any further type validation.

In contrast to strings, integers can only have numbers in them. Decimal numbers can additionally also have a decimal separator somewhere in the middle of the value. Whereas many other data types can have virtually infinite different values and still be valid, boolean-typed parameters can only have two different values: true or false.

Once the data type has been validated and identified correct, we need to check that the value is included in the set of acceptable values – be it finite or infinite. There can be various different constraints imposed on the value of any parameter restricting the set of acceptable values, some that are applicable to all data types and some only for e.g. numerical data types. For example, it does not make sense to restrict a string-typed parameter `x` with condition $x \leq 100$, but for an integer that would be sensible.

Not only the values of specific parameters are of interest when it comes to validating configuration files. Also, the number of occurrences – or *cardinality* – of specific parameters may be restricted by other values or cardinalities in the configuration. For example, it does not make sense for `DateOfBirth` parameter to occur more than once in a configuration item specifying the data of one person. It also does not make sense for `DateOfBirth` to be missing altogether.

4.1.3 Validation levels

One way to define data validation levels is to separate the levels hierarchically according to the scale of the used data entities. For example, the lowest level validations could consider only the most simple syntax checks within one line of a data file, and the highest level validations could compare aggregate validation results within huge organizations around the world for consistency checks.

For defining validation levels, Simón suggests a six-level model used in European Statistical System for validating statistical data, illustrated in Figure 4.1. [46] In the model, Level 0 represents the format and file structure checks of a given file type. In other words, the first level is responsible of checking the syntax correctness. Level 1 consists of value checks within one file, most of which are implemented as value constraints. Such checks may compare different data items to each other within one file, for example, checking that `DateOfDeath` is later in time than `DateOfBirth`. Level 2 is designated for checks between different files provided by the same source. These files could be different revisions of the same file – also known as time series checks – or totally different files. For example, such validations could include checking a person has a later `DateOfBirth` than both their parents' `DateOfBirth` in their own files. Level 3 consists of checks between data from different sources. This validation level includes, for example, mirror checks that are for checking consistency between different data sources. Level 4 could be defined as consistency checks between different domains within the same institution. Validations in this

level check the plausibility of the same phenomenon or different correlated phenomena from different domains: for example, unemployment from registers and from labour force surveys. Level 5 at the top includes consistency checks with data within and outside an institution. For example, checking consistency between data provided by the European Union member states according to the European Union commission legal acts and data collected by the member countries for national purposes. [46]

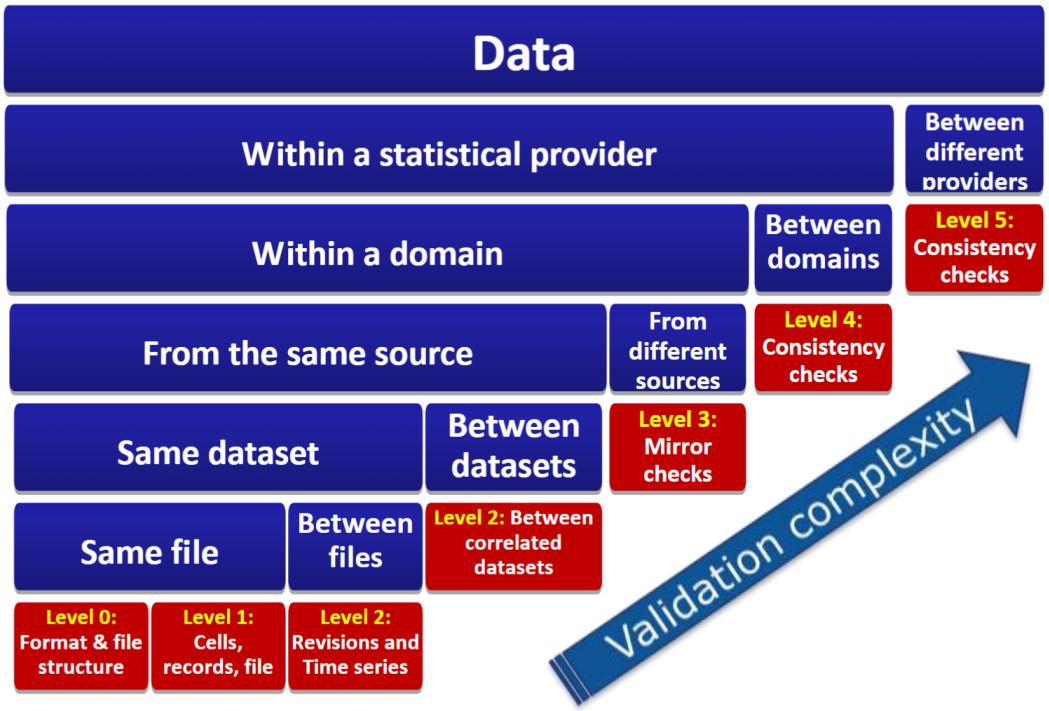

Figure 4.1. *An overview of six validation levels suggested by Simón. [46]*

Huang et al. similarly propose a model with three validation levels specifically for configuration files, shown in Figure 4.2. [35] The first level checks the parameter syntax and value existence. The second level checks value ranges, consistency, uniqueness and relation to other values. The third level checks that the value points to the desired component.

Both of the aforementioned validation level models seem to indicate that the higher level and more complex validations we use to validate our data, the more confident we can be about the correctness of the data. Simón also points out that at the higher levels it is more difficult to identify "fatal errors" that imply rejection of the data.

The three levels suggested by Huang et al. mostly seem to match with the first three levels suggested by Simón. It makes sense for the model suggested by Huang et al. to only include the first three levels of validation because system configuration files are very different compared to statistical data handled by European Statistical System: Configuration data is only supposed to be used narrowly as configuration for a specific system whereas statistical data could be anything gathered from anywhere. Thus, levels 3, 4 and 5 in Simón's model are not necessarily relevant in the context of configuration file validations.

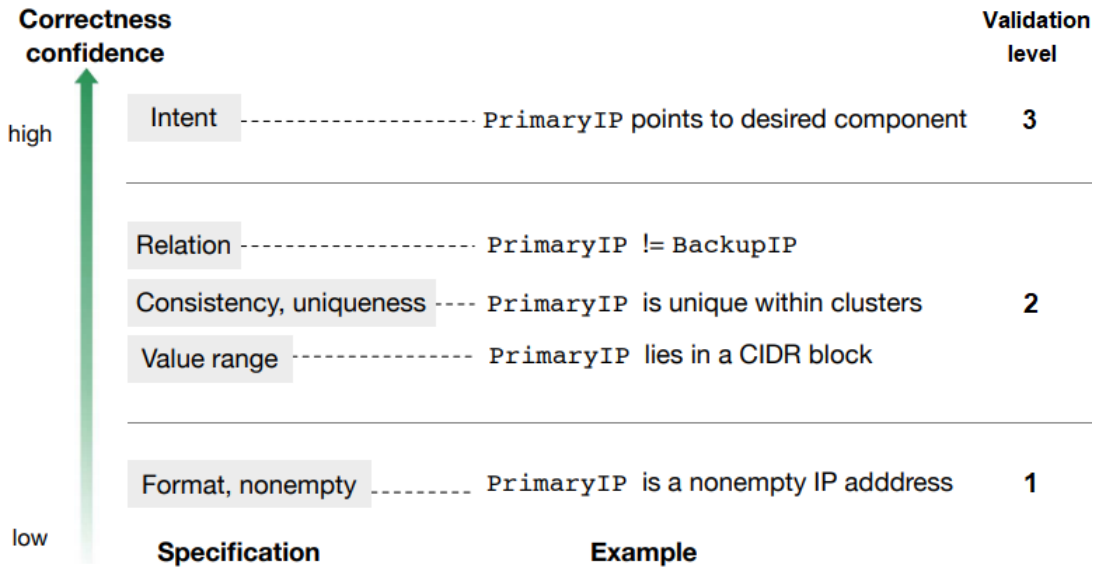

Figure 4.2. *Three different validation levels suggested by Huang et al., adapted from [35]*

4.2 Implementation models

Configuration validation in a computer system is usually implemented according to a specified data model or schema that defines all the data types and constraints of the system parameters and their relations to other parameters. Some of such schema languages include, for example, XML Schema Definition (XSD)[47], Document Type Definition[48], RELAX NG[49], JSON Schema[50] and CPL[35].

Program code can be generated from such schemata, resulting in classes representing the types of elements in the respective schema. Generating such code from schemata and using it is referred to as data binding. Data binding is frequently used, for example, with Extensible Markup Language (XML) documents and many different programming languages. [51]

The generated code also allows the user of that code to trivially marshall documents into objects in the software memory and, vice versa, unmarshall objects to documents. [51]

An alternative to generating code based on a schema is run-time schema parsing. When the schema is not translated into programming language code and compiled as part of a software binary, it is possible to change the effects of the schema without compiling the software again. This is useful especially in cases where the software schema needs to be changed by people who do not have the correct tools for compiling the software again using the new schema. One downside of the run-time parsing approach is the computational overhead: The software needs to parse the schema every time the application is initialized, and constantly accessing and comparing the various values of schema elements at run-time might be slower compared to executing the generated schema-based code.

Program 4.1 shows an example of XML schema definition. The schema defines a person with attributes first name, last name, age in years and gender. Constraints for the age included in lines 12 and 13 show that only ages between 0 and 150 are considered valid according to the schema. Also the gender is restricted to be only either male, female or other.

```
1  <?xml version="1.0"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3    <xs:element name="Person">
4      <xs:complexType>
5        <xs:sequence>
6          <xs:element name="FirstName" type="xs:string"/>
7          <xs:element name="LastName" type="xs:string"/>
8          <xs:element name="AgeYears">
9            <xs:simpleType>
10             <xs:restriction base="xs:integer">
11               <xs:minInclusive value="0"/>
12               <xs:maxInclusive value="150"/>
13             </xs:restriction>
14           </xs:simpleType>
15         </xs:element>
16         <xs:element name="Gender">
17           <xs:simpleType>
18             <xs:restriction base="xs:string">
19               <xs:enumeration value="Male"/>
20               <xs:enumeration value="Female"/>
21               <xs:enumeration value="Other"/>
22             </xs:restriction>
23           </xs:simpleType>
24         </xs:element>
25       </xs:sequence>
26     </xs:complexType>
27   </xs:element>
28 </xs:schema>
```

Program 4.1. *Example of an XML schema definition.*

Program 4.2 shows an example XML document that is valid according to the schema in Program 4.1. The document represents a male person named John Doe whose age is 29 years. If, for example, the age was changed to 200 or the gender was changed to "Intersex", the document would not be valid anymore.

Program 4.3 shows an example of how the schema in Program 4.1 could be translated into a class in C++ programming language: The child elements of the person type are listed as private member variables and their respective public accessor methods. The

```
<Person>
  <FirstName>John</FirstName>
  <LastName>Doe</LastName>
  <AgeYears>29</AgeYears>
  <Gender>Male</Gender>
</Person>
```

Program 4.2. *Example of an XML document that is valid according to the schema defined in Program 4.1.*

possible choices of gender are listed as a separate enumeration. For the sake of keeping the example simple, marshalling and unmarshalling the XML document is not addressed in this class.

```
class Person {

public:
    enum class Gender {
        Male,
        Female,
        Other
    };

    void setFirstName(const std::string & value) {
        myFirstName = value;
    }

    void setLastName(const std::string & value) {
        myLastName = value;
    }

    void setAgeYears(const int & value) {
        myAgeYears = value;
    }

    void setGender(const Gender & value) {
        myGender = value;
    }

    std::string getFirstName() const {
        return myFirstName;
    }

    std::string getLastName() const {
        return myLastName;
    }

    int getAgeYears() const {
        return myAgeYears;
    }

    Gender getGender() const {
        return myGender;
    }

private:
    std::string myFirstName;
    std::string myLastName;
    int myAgeYears;
    Gender myGender;
}
```

Program 4.3. *Example of how the XML schema defined in Program 4.1 could be translated into a C++ class.*

5 DESIGN CONSTRAINTS AND DECISIONS

All software have their own environment, functional and non-functional requirements, and a variety of other constraints according to which they are required to operate. This chapter lists the design constraints for Visy Configurator and how those constraints are addressed with design decisions.

5.1 Intended users and use cases

Visy Configurator is mainly intended to be used by employees who commission and maintain systems, and by software developers who test and document the functionality of each configuration item in the software. Currently in some cases and probably even more in the future, the end customers may also use the application to configure their systems themselves.

The users themselves were also part of the reason to implement a new configuration management system instead of using an already existing one: The system had to be simple enough and easy to use instead of requiring the user to write any sort of program code, similar to what was shown in Program 3.1 and Program 3.2. The system was to be specifically designed for the prevalent infrastructure used in Visy systems, so it also did not need to be as generic and versatile as the available third-party configuration management systems.

The company employees were asked what kind of features they would like to have in the configuration management system. For the time being, only the most essential features were chosen to be designed and implemented. Table 5.1 shows a list of user stories collected based on their answers.

Being able to access the documentation of each parameter in the configuration files has only been possible so far by actually browsing through the source code of the executable using that specific file and the parameters included in it. Thus, especially for new employees, it has been quite hard to learn and remember what kind of effect each and every parameter has in the system. Having an easy access to the documentation helps understanding and maintaining the configurations in every system.

Validating configuration files has also been considered a must-have feature. Previously the only way to validate any configurations have been by manually reading through the

Table 5.1. *Visy Configurator user stories.*

As a ...	I want to ...	so that ...
user	access the documentation of each parameter in the configuration files	I know what will be changed in the system when I change the value of any parameter
user	validate configuration files	I know whether there are any syntactic or semantic errors in the configuration I've built
user	distribute my configuration files to all computers in the system from a single point of access	I do not have to manually copy-paste files over multiple remote desktop connections
user	create backups of the whole system software from a single point of access	I do not have to manually copy-paste files over multiple remote desktop connections
user	roll back to an earlier version of the system software from a single point of access	I can quickly change a new, potentially incorrect, configuration back to an old one that works correctly
user	control the execution of applications and services in the system from one point of access	I do not have to manually restart software using multiple remote desktop connections

files, trying to look for any errors, and by trying to start the executable using the files. For some erroneous configurations, the software logs error messages into log files during software startup. Though, most of the time there are no errors indicated at all in the log files or the error messages are very ambiguous. Having built-in validation rules and mechanisms for executing such validations helps speeding up the work of actually building the software configuration: The user would no longer need to spend huge amounts of time manually looking for errors, but let the application handle that automatically instead.

Being able to distribute software updates from a single point of access to the remote computers is one of the core features making Visy Configurator a powerful configuration management tool: It enables system commissioning to become more scalable by automating tasks that were previously done manually. The full commit process consists of the following steps:

1. Initialize a connection to the remote host.
2. Kill all Visy processes.
3. Uninstall all Visy services.
4. Copy all files to the remote hosts and remove files that do not belong in them.
5. Install all Visy services.
6. Start all Visy services.
7. Start all Visy applications.

8. Set all Visy applications to start on system startup.

The full commit process described above ensures that the system is in a desirable state after the commit, no matter what kind of changes are made to the system configuration. Depending on what exactly is being updated in the commit process, some of the steps can be skipped. For example, if only the files of a specific pre-existing service is being updated, it is enough to just kill that specific service, upload the files, and start it again – without touching any of the other processes executing on that host. As another use case example, if the user only wants to restart specific services or applications in the remote hosts, it is enough to only kill the specific processes and start them again.

Implementing the commit process in the aforementioned way results in the system being a declarative configuration management system, as discussed in Chapter 3: The user does not have to write any commands to be executed on the remote hosts, but rather just define the services and applications to be installed and executed on them, along with their individual configurations.

Creating remote double backups of the system software has also been a tedious process, manually copying files from dozens of computers to another remote backup location. Using Visy Configurator would enable the users to create such backups easily from a single point of access, including the software configuration from all of the computers on site. This would also apply on pre-existing projects where the application has not been used before.

Figure 5.1 shows the Visy Configurator use case diagram created based on the user stories. In the figure, *Local workspace* represents the local copies of the system files. The user can commit the local files to the system or vice versa fetch the system files to the local workspace to create a backup. Practically *Visy system* in the figure represents a set of hosts in the same network with the host running Visy Configurator, each running some Visy software.

5.2 Design principles

This section list the most important design principles that were chosen because of the application user requirements, future outlooks on further development and prioritized non-functional requirements that improve, for example, software usability and maintainability.

Making deployment of systems more effortless is the main idea of the designed configuration management software. So the foremost priority of the software is that it should be easy to use for deploying many instances of software across multiple hosts from one point of access.

Maintainability was chosen as one of the most important aspects of the architecture design, as it is highly probable that the software will be expanded in the future with new features, such as new types of more complex configuration validations and new ways of

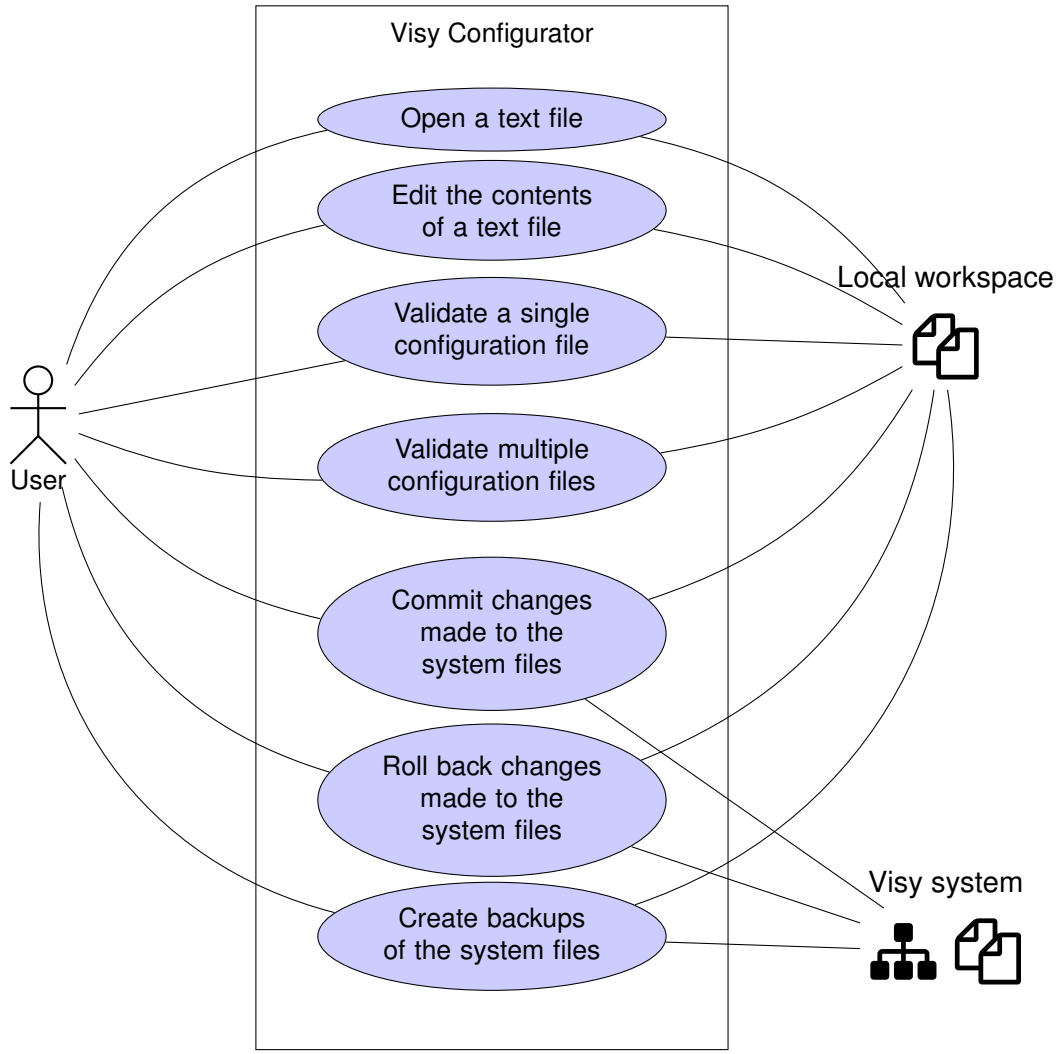

Figure 5.1. *Visy Configurator use cases.*

generating configurations. Thus one of the design decisions was to keep the software components' implementations decoupled from each other and dependent only on each others' interfaces.

Portability was also kept as a high priority goal in the architecture design. Even though all Visy software is currently running on Windows, in the future it might be running at least partly on Linux. Thus any compiler and operating system specific software components were kept in the bare minimum, mainly utilizing standard libraries and other cross-platform components.

Software robustness was also chosen as a more-or-less obvious high importance goal. Because the users of the application could be allowed to input a variety of incorrect data to the application from many different sources – including, for example, configuration schema files and user interface (UI) text fields – the data from each source should be validated sufficiently and errors should be indicated to the user.

In case of fatal errors where the program execution can not be continued in a sensible manner – for example, when there is a syntactic or semantic error in a parsed schema file

– exceptions can be thrown at the location of error and caught only in the topmost level of execution where a corresponding error dialog is displayed. In case of less significant errors where the application still functions as intended, no exceptions are thrown but corresponding error texts are still indicated to the user.

High performance was not considered as a high priority goal to achieve. Commissioning and maintaining systems is not usually anything time-critical, but rather more about making thoughtful decisions taking into consideration all the aspects in the environment where the system exists. Optimizing software performance can also lead to more complex design solutions, when for example parallelizing tasks. Design aspects considered more important than performance were for example code readability and simplicity.

The software should be compatible at least file-synchronization-wise also with older systems, and should preserve its total backwards compatibility when updates to the configuration schema are made. Breaking backwards compatibility is allowed on rare occasions, e.g. when the old code becomes obsolete and better ways to do things are developed.

5.3 System components

The system architecture diagram was created using the C4 model discussed in Chapter 2 that presents the software architecture in four different levels of abstraction: context, container, component and code. Figure 5.2 shows the configuration management system context diagram where the roles of each container in their environment and the main ideas of the designed system are described in a high level of abstraction: Visy Configurator is the graphical user interface (GUI) application used by personnel to configure systems.

Figure 5.3 shows how the application architecture is designed. To implement high level separation of concerns within the application, it is divided into five separate components: Application core / GUI, configuration schema interface, configuration validation, configuration factory and project management. The architecture of the application conforms to the Relaxed Layer System architectural pattern discussed briefly in Chapter 2 because the application core layer does not only use configuration validation component right below it but also configuration schema layer that is below configuration validation.

The following sections go more into details on how the individual components of Visy Configurator application are designed.

5.3.1 Application core / GUI

The application core is the component that includes the executable entry point, instantiates the peripheral components and contains the graphical user interface for user input and output. Actions in the other components are triggered via the user interface. The user interface can be implemented utilizing, for example, a GUI framework that inverts

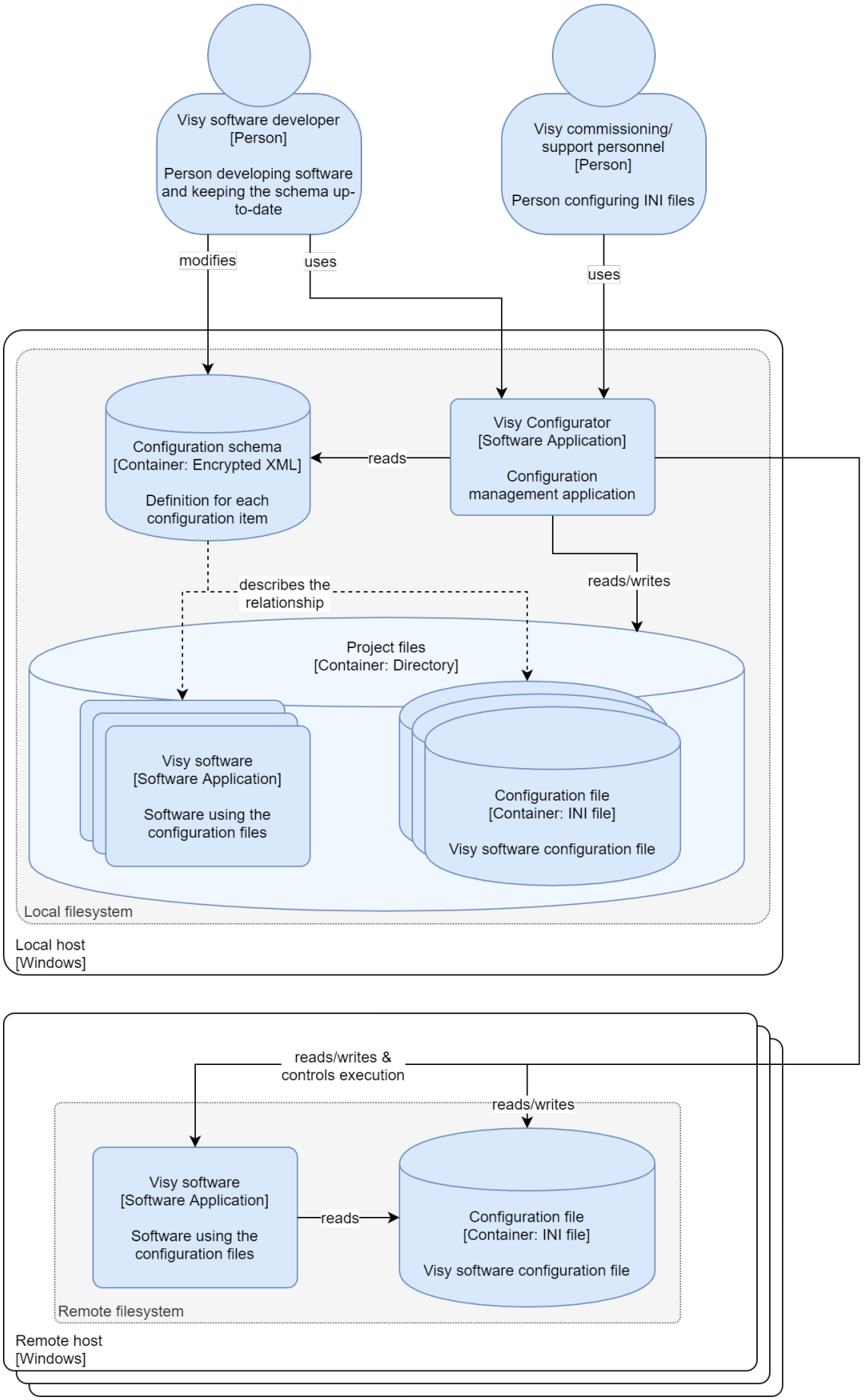

Figure 5.2. *Visy Configurator system context diagram.*

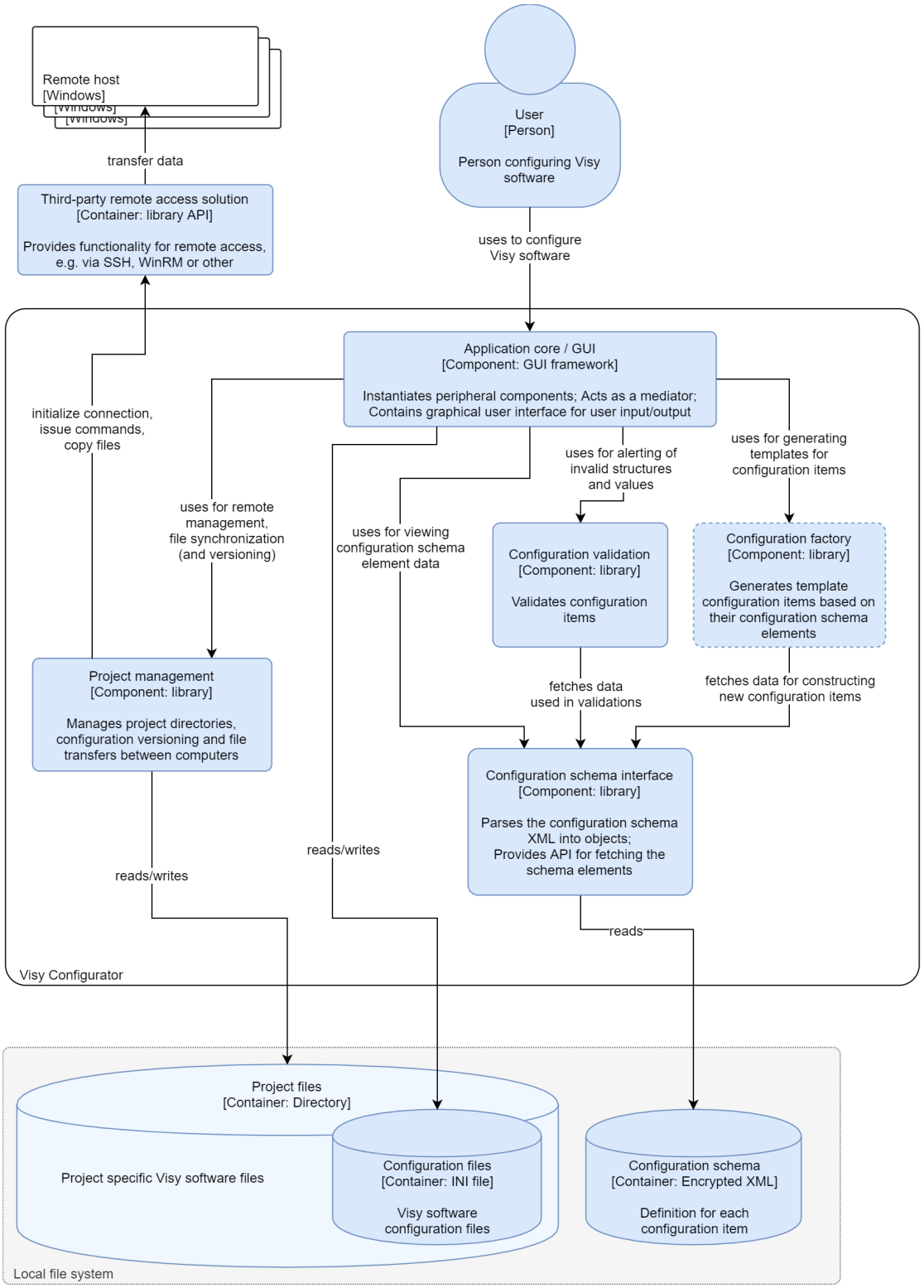

Figure 5.3. *Visy Configurator system container diagram.*

the flow of control.

5.3.2 Configuration validation

Figure 5.4 shows the component diagram of the configuration validation component. The component is used for validating the system configuration files that have `.ini` file extension. This provides the user interface both syntactic and semantic validation results of each validated configuration file, including for example what is the error reason and on which lines of the file the error exists. The user interface may then, for example, highlight the errors in the files using various graphical indicators.

The configuration validation component can execute a variety of different validations: configuration item cardinality, value data type, value limits, enumerations for possible values, and so forth. There is also support for more complex validations that include comparisons between two or more variables within a configuration file.

The subcomponents of configuration validation are layered to reflect the corresponding configuration items they are supposed to validate. The user can, via the user interface, validate either whole directories of configuration files recursively or just single files. The validator of a single file in turn validates parameter groups whose validators validate individual parameters at the lowest level.

5.3.3 Configuration factory

The configuration factory component was not yet fully designed due to its low priority and not being actually required as a feature. Thus, it is marked with a dashed line in Figure 5.3. The component would be used for constructing new configuration items, for example, in case the configuration validation has detected that there are some configuration items missing. As input data, this component would receive the configuration schema information and the parent configuration item to whom the created item should be added.

5.3.4 Configuration schema interface

Figure 5.5 shows the contents of the configuration schema interface component. The purpose of this component is to provide access to the schema definitions of each configuration file used by any Visy application, service or library. This information is used for example in validating the existing configuration items, constructing template configuration items and providing the user documentation of each available configuration item.

The file encryption interface subcomponent is marked with a dashed line in Figure 5.5 because it was not actually required and thus not yet decided whether it will be a part of

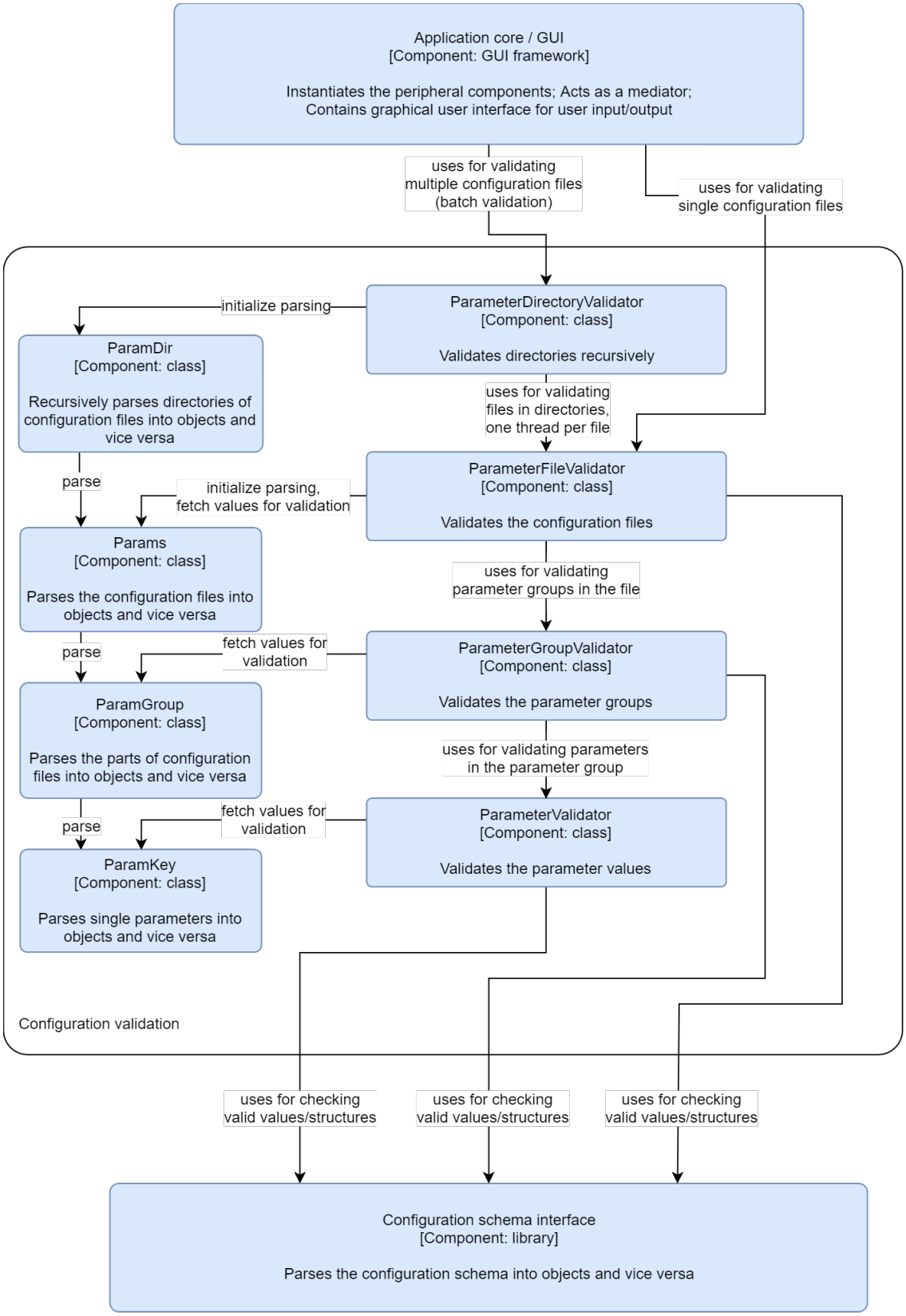

Figure 5.4. *Configuration validation component diagram.*

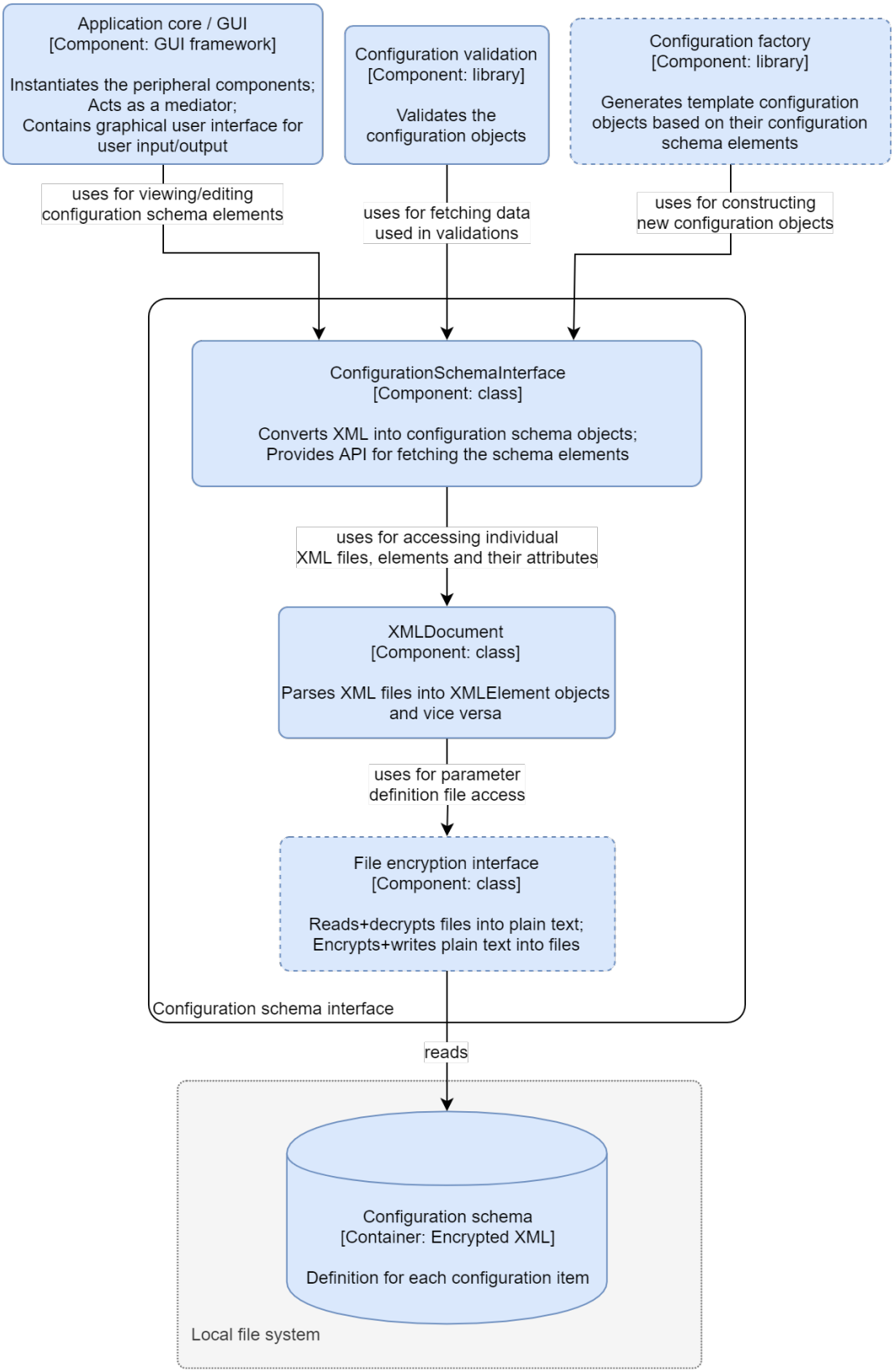

Figure 5.5. *Configuration schema interface component diagram.*

the final design. The configuration schema files could eventually be either unencrypted or encrypted XML files, depending on whether any actual need for encrypting them arises.

5.3.5 Project management

Figure 5.6 shows the contents of the project management component. This component is responsible of creating new project directories and remotely copying files, uninstalling, installing and restarting services and applications on various hosts connected to the system network.

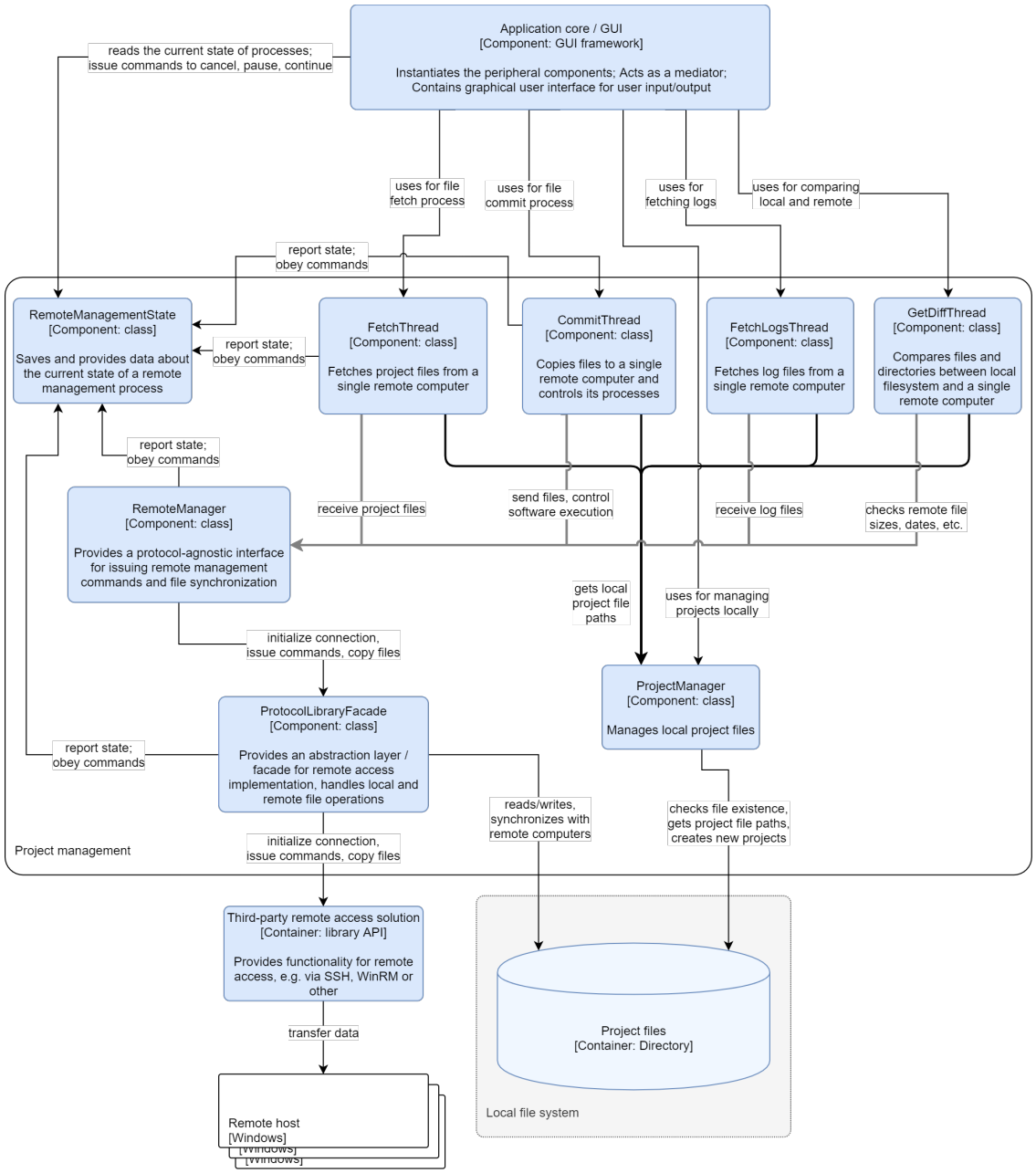

Figure 5.6. *Project management component diagram.*

Because the ease of deployment of software was one of the most prioritized aspects of the software, it was also decided that Visy Configurator should be kept agentless, meaning there is no need to install a separate agent service on each computer in the system for the application to be able to execute all its required tasks. Earlier in the design process, Visy Remote Agent was also a part of the architecture design, executing on all system hosts, but due to the aforementioned reason the idea was abandoned.

The user interface instantiates the type of remote management thread – one per concurrently managed host – it needs for the current action requested by the user. All these threads report their status to the user interface through `RemoteManagementState` objects and obey the commands issued from the user interface.

`ProjectManager` subcomponent manages local project files, including the directory structure and its versioning. Versioning shall be implemented using a third-party version control software – such as Git – that has a decent application programming interface (API) available. Version control was discussed in more detail in Chapter 3.

Hierarchical separation of concerns, or layering, manifests itself in the component when looking at how any of the remote management threads use their peripheral subcomponents: Using `RemoteManager` does not require the thread objects to know anything about the fact that the underlying remote access is implemented using e.g. SSH protocol, and in turn, `RemoteManager` using `ProtocolLibraryFacade` does not need to know how the underlying lower level protocol library interface – with a multitude of different functions and data types – is used in practice.

It is also worth mentioning that `GetDiffThread` compares the files found in the local filesystem and the remote filesystem. This functionality in turn also helps detecting any configuration drift that might happen when, for example, someone changes the remote system files the traditional way: The user initiates a remote desktop connection to the system and edits the files in that filesystem directly, causing differences between the local files and the remote files.

6 ARCHITECTURE EVALUATION

The architecture of the designed configuration management system was chosen to be evaluated using DCAR method that was described earlier in more detail in Chapter 2. Mostly because of its lightweight nature, DCAR was considered suitable for the purpose in a small company, where not too many software architects, developers and management personnel are available for a time-consuming architecture evaluation process. Experience from other authors [52] also implied the light weight of DCAR and thus suitability for this context.

Having never conducted a proper architecture evaluation before, some of the steps in the process took a little longer than planned beforehand due to the confusion in some instructions. Eventually the evaluation process took as long as a full-scale DCAR would have taken, about six hours in total, but with fewer people involved, still resulting in less person-hours.

This following sections describe how the architecture evaluation session was conducted and what kind of results the process yielded.

6.1 Preparation, introduction to the process and presentations

Before the actual evaluation session, in DCAR step one, management and architecture presentations were prepared, and the reviewers did some preliminary inspection of the designed architecture based on the freshly polished architecture diagrams. Also templates for required documents to be filled during the process were gathered or created.

Because executing a full-fledged DCAR process was not feasible, a lightweight version of an already lightweight architecture evaluation process was prepared. Only four reviewers were available for the evaluation process: the main architect of the designed system and three other experienced software developers and architects. Also the management and architecture presentations were cut shorter due to the reviewers being already acquainted with the software architecture and the requirements for the software.

The actual architecture evaluation session started with DCAR step two, when the reviewers gathered in the same room and the evaluation process was introduced to the them in more detail. After that, in DCAR steps three and four, the management and architecture

presentations were conducted, briefly stating the requirements for the designed system and presenting the architecture diagrams, including some of the associated architecture decisions.

6.2 Forces and decisions identification and prioritization

After the presentations, architecture decisions and forces affecting them were identified, and a relationship diagram shown in Figure 6.1 was composed accordingly in DCAR step five. The diagram contains architecture decisions and connections of type *caused by* and *depends on* between them, describing how the connected decisions are related. Unlike in the original DCAR process, some forces and requirements, distinguished from the decisions with blue color, were also put in the diagram to include additional clarification for the rationale behind the decisions.

After creating the diagram, all identified architecture decisions were listed, as shown in Table 6.1. Then, in DCAR step six, a shorter list of the most important decisions were prioritized based on a vote where each reviewer was given one hundred points to spread out for each decision they thought to be important and worth further inspection. Five of the top prioritized decisions, highlighted with gray background color in the table, were chosen to be documented and evaluated further according to their total points $\sum P_i$.

Some rationale, mentioned during the vote, for giving the decisions their respective points by the reviewers included quotes such as

"Undoing changes and overall traceability is very important."

"SSH would seem like the best solution for remote access."

"It is important to be able to work on the configuration of a host even when it is offline, so that once it gets online again, the updates can then be taken into use instantly."

"Separation of concerns with modular design is an obvious decision to do, and thus important. Modifying one aspect of the software afterwards should not require doing changes in many different places in code."

"Schema files are good for extensibility and developers are familiar with XML already."

"Centralized editing of files and the possibility for offline work seems like a good practice."

The decision to use third-party version control software to implement version control for the project files gathered the most points in the vote, and it also caused the most debate between the architect and other reviewers when documenting and evaluating the decision in the next steps of DCAR.

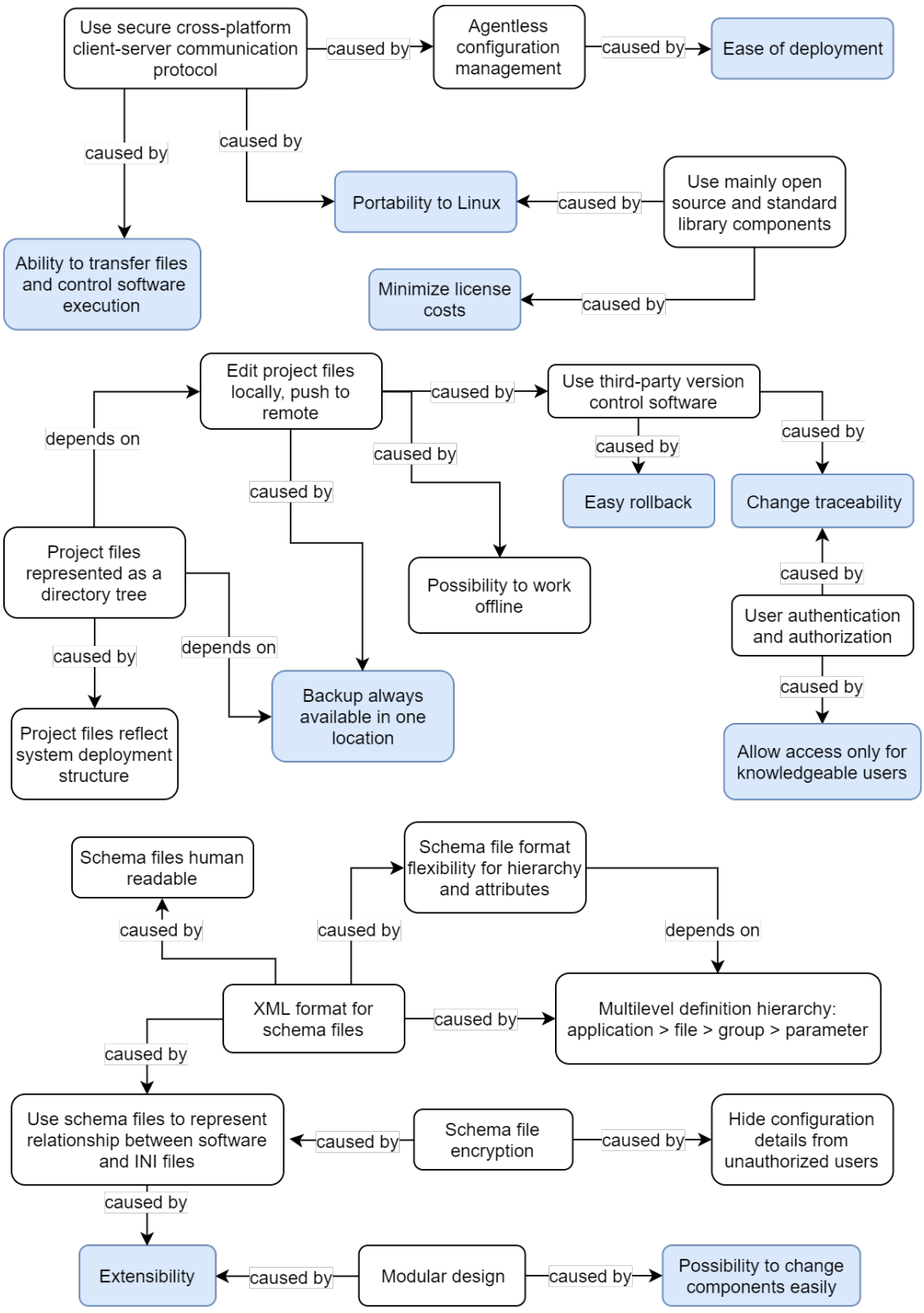

Figure 6.1. *Relationship diagram of Visy Configurator architecture decisions.*

Table 6.1. *Visy Configurator architecture decision prioritization. P_i denotes the points given by each respective reviewer.*

Architecture decision name	P_1	P_2	P_3	P_4	$\sum P_i$
Use secure cross-platform client–server communication protocol	15	0	0	10	25
Agentless configuration management	35	0	15	30	80
Use mainly open source and standard library components	0	0	0	0	0
Modular design	15	10	20	20	65
Edit project files locally, push to remote	0	20	10	0	30
Possibility to work offline	0	0	5	0	5
Use third-party version control software	25	50	30	20	125
User authentication and authorization	10	0	0	0	10
Project files represented as a directory tree	0	0	0	0	0
Project files reflect system deployment structure	0	0	0	0	0
Use schema files to represent relationship between software and configuration files	0	20	20	20	60
Schema files human readable	0	0	0	0	0
Schema file format flexibility for hierarchy and attributes	0	0	0	0	0
XML format for schema files	0	0	0	0	0
Multilevel definition hierarchy: application > file > group > parameter	0	0	0	0	0
Schema file encryption	0	0	0	0	0
Hide configuration details from unauthorized users	0	0	0	0	0

6.3 Decisions documentation and evaluation

After the decisions were prioritized, they were documented more precisely according to the provided templates in DCAR step seven. Tables 6.2, 6.3, 6.4, 6.5 and 6.6 show how each of those decisions were documented, describing the decision name, the problem it solves, the solution, alternative solutions and so forth.

After documenting the decisions in more detail, they were evaluated by the reviewers in DCAR step eight. The evaluation results were then added to the documentation in the *Outcome* and *Rationale* fields. Outcome could be either *Good* highlighted with green, *Acceptable* highlighted with yellow or *Needs to be reconsidered* highlighted with red. Rationale contains the reason for why the reviewers chose the specific outcome.

The only prioritized decision that was not deemed unanimously good was the decision to utilize third-party version control software to implement versioning, that also earned the highest points during the prioritization vote. Hiding the intricacies of version control software from the user was considered to be challenging and therefore users who have never

used version control software might be tempted not to use the versioning functionality at all. Also using a full-fledged version control system under the hood was considered a bit of an overkill for the purpose of just keeping a local version history with no multiple user accounts involved. On the other hand, implementing a custom version control scheme could have also been too complicated, and thus giving the responsibility of versioning to a third-party solution was considered also a favorable decision, eventually setting the decision evaluation outcome as acceptable.

Table 6.2. *Architecture decision documentation for "Agentless configuration management".*

Name	**Agentless configuration management**
Problem	Ease of deployment without additional software agent to maintain
Solution or description of decision	Use existing operating system facilities for remote access and control. Does not require installing additional custom-made software daemon.
Considered alternative solutions	Implement and install a separate Visy Remote Agent on each controlled remote host: Flexibility to implement whatever kind of protocol and transfer arbitrary data between systems. However, this approach requires also maintaining the agent software itself.
Forces in favor of decision	• Does not require installing an additional software daemon • Less work in design and implementation • No maintenance of another piece of software or version conflicts between server and client etc.
Forces against the decision	Less flexible than custom agent
Outcome	Good
Rationale	Causes some limitations, but seems like the only sensible choice

After the session, a report of the results was composed, essentially containing the same information already shown in this chapter.

Table 6.3. *Architecture decision documentation for "Modular design".*

Name	**Modular design**
Problem	Software should be maintainable and extensible
Solution or description of decision	Software split into components with their own responsibilities and clearly defined interfaces between each other. Implements separation of concerns and decoupling between components. Exception safety.
Considered alternative solutions	Monolithic design: • Access data more directly and faster • Implementing new features faster in some cases, because less module interfaces to maintain
Forces in favor of decision	• Extensibility for upcoming new features • Makes the software more maintainable and understandable
Forces against the decision	• Slightly less performance compared to a monolith • Slower prototyping
Outcome	Good
Rationale	Advantages of monolithic design are hard to imagine in practice, at least with a long development and maintenance time horizon

Table 6.4. *Architecture decision documentation for "Edit project files locally, push to remote".*

Name	**Edit project files locally, push to remote**
Problem	How to synchronize files and control processes on remote hosts automatically
Solution or description of decision	Edit all project files centrally and distribute them by pushing to remote hosts
Considered alternative solutions	• Edit files directly on the remote host using e.g. disk share. • Use Microsoft Management Console for restarting, installing and uninstalling services.
Forces in favor of decision	• Backup always available • Facilitates version control • Allows to commit all changes together at once
Forces against the decision	• Takes more space on the computer running the application • Single point of failure when editing files and the computer breaks up • Does not prevent users from modifying files directly on remote hosts
Outcome	Good
Rationale	Less flexible but easier to follow and version

Table 6.5. *Architecture decision documentation for "Use version control software".*

Name	**Use version control software**
Problem	How to keep track of changes in files over time and do easy rollback to earlier versions
Solution or description of decision	Version control software to keep track of versions and users who commit changes. All files of the whole project are contained within one repository that is located on the same host as the application.
Considered alternative solutions	Develop own version control scheme, e.g. directories with timestamps as names
Forces in favor of decision	• Traceability • Backups of earlier versions • No need to implement and maintain custom solution
Forces against the decision	• Flexibility with custom version control • Overkill using heavy version control software with many unneeded features • Integrating an existing version control software might be harder than custom solution • Using any kind of version control is a new thing for many employees
Outcome	Acceptable
Rationale	Complicates software project considered how small part of the whole software it is. Considering user experience, might be hard for people who have never used version control software before.

Table 6.6. *Architecture decision documentation for "Use schema files to represent relationship between software and configuration files".*

Name	**Use schema files to represent relationship between software and configuration files**
Problem	How to describe definitions, validations and documentation for various software configuration files and their parameters
Solution or description of decision	Use schema files to represent relationship between software and configuration files. Modified by software developers to reflect changes in code. Schemas parsed run-time instead of translating into program code.
Considered alternative solutions	• Hard-coded validations and documentation for all parameters in the application binary itself • Metadata about documentation and validations included in configuration files themselves • Huge repository of configuration files and inference of correct values from those files using machine learning
Forces in favor of decision	Supports all kind of Visy software configuration files without need to compile again
Forces against the decision	Laborous to maintain compared to machine learning inference
Outcome	Good
Rationale	Ok, but might require some schema maintenance tools

7 IMPLEMENTATION

This chapter explains how the proposed software architecture of Visy Configurator was implemented in practice and what kind of challenges were encountered during the implementation process.

The great majority of the program source code at Visy is written in C++ programming language, including, for example, the class that is responsible of parsing the configuration files used in the systems. There are also a variety of C++ compilers for different platforms [53], so it also promotes making software cross-platform. The language has been for a long time an industry standard [54, 55] and is used widely in many different applications [56]. There also exists a huge spectrum of third-party software components [57, 58], both closed and open source. Due to these reasons, C++ was chosen as the primary programming language for implementing the application.

One of the constraining implementation decisions was to mainly utilize libraries that are commercially free to use and incur no liability to make the application open source as well. This added some challenges in implementing the software according to the architecture when finding the suitable frameworks and libraries with adequate licensing for the task.

By the time of writing this thesis, version control was not yet implemented as a part of the project management component that was described in Chapter 5, but its future implementation is discussed briefly in Chapter 8.

The following sections dive more into details on how the different components of Visy Configurator were implemented.

7.1 User interface

One of the most significant implementation aspects was the user interface design because most of the software architecture is designed around it in an attempt to make the user experience enjoyable. Figure 7.1 shows how the main user interface of the application was designed to be laid out. The largest area in the middle of the user interface is occupied by the configuration editor and the peripheral parts include components such as file explorer and validation results view.

The UI layout was purposely chosen to be similar with modern integrated development environments (IDE). In a way configuring the systems is similar to programming: The

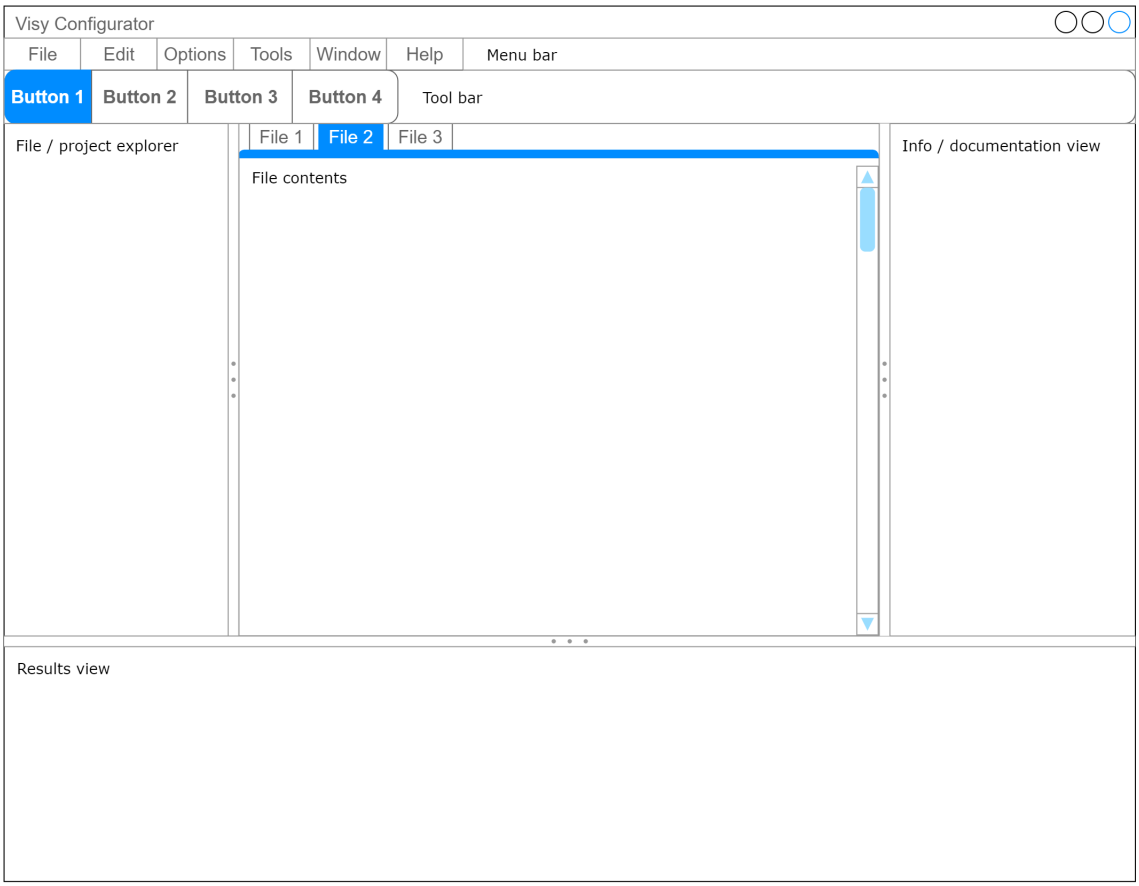

Figure 7.1. *The original layout design of Visy Configurator main user interface.*

user writes configurations similar to program source code and instead of compiling and seeing compilation results, the user just executes syntactic and semantic validations on the files and sees the validation results. Thus, the IDE-like UI feels quite intuitive to use when configuring systems.

Most of the time the implementation of each component's API was user interface driven, using a top-down approach. First, a UI prototype and a program skeleton calling appropriate functions of the peripheral components were built. For example, a button *"New project"* that would open the project wizard was created, and based on the inputs from that wizard, the new project would be created by calling *createProject* function with the given inputs.

Because there was already a license for C++Builder IDE and Visual Components Library (VCL) that is tightly integrated in the IDE and they were easy to use for rapid UI development, the user interface prototyping originally started with those tools. As shown in Figure 7.2, the configuration editor component was implemented as a group of expandable category UI components each containing one parameter group and inside those separately each parameter within that parameter group: This would completely prevent the user from making at least any syntactic errors when editing the configuration because the resulting configuration file would always be generated by the software based on the limited user inputs. The user access to documentation of each configuration item was

also implemented as a separate UI component that was always visible on the right-hand side of the screen, as seen in the figure.

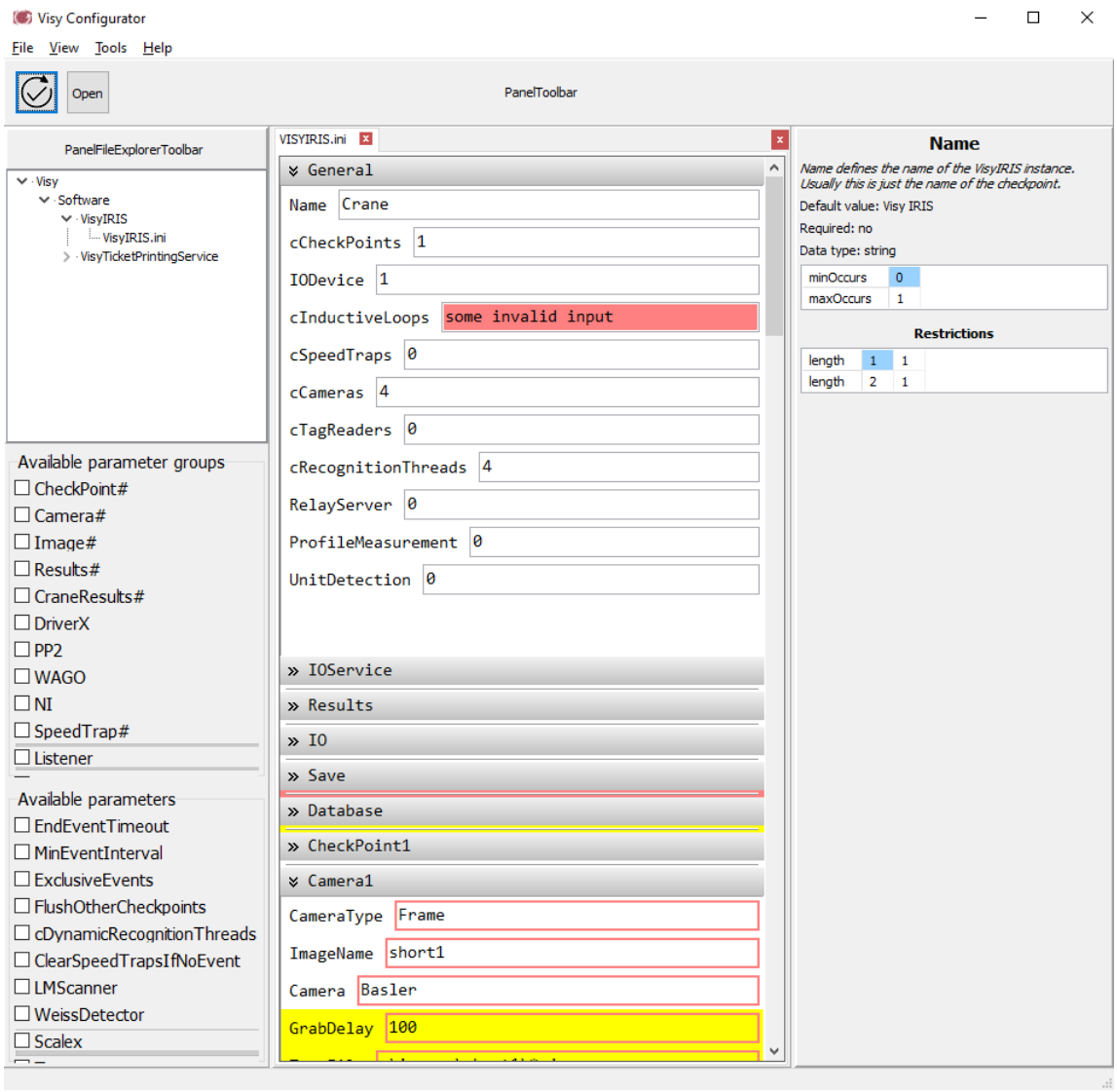

Figure 7.2. *The first revision of Visy Configurator user interface.*

Later it was decided that the original user interface requires too much custom programming for implementing necessary functionalities such as undo and redo, and it would be too awkward to use for rapidly making changes in the configuration files because of the restricted input mechanism. This led to the decision that the configuration editor should be just a single highlighting text editor for a single file – the way it was already before, but with extra functionality included.

Because VCL lacked native support for a versatile highlighting text editor, there were not suitable ones to be found from third parties and implementing a custom one was considered too much work, the user interface was reimplemented using wxWidgets cross-platform GUI library that had not only Scintilla text editor component already integrated to it natively but included also many other favorable features. Figure 7.3 shows the new user interface with a text editor highlighting INI syntax and semantic errors. For user

documentation access, it was also decided that showing a tooltip when hovering the mouse cursor over the configuration item would be better, as shown in the figure.

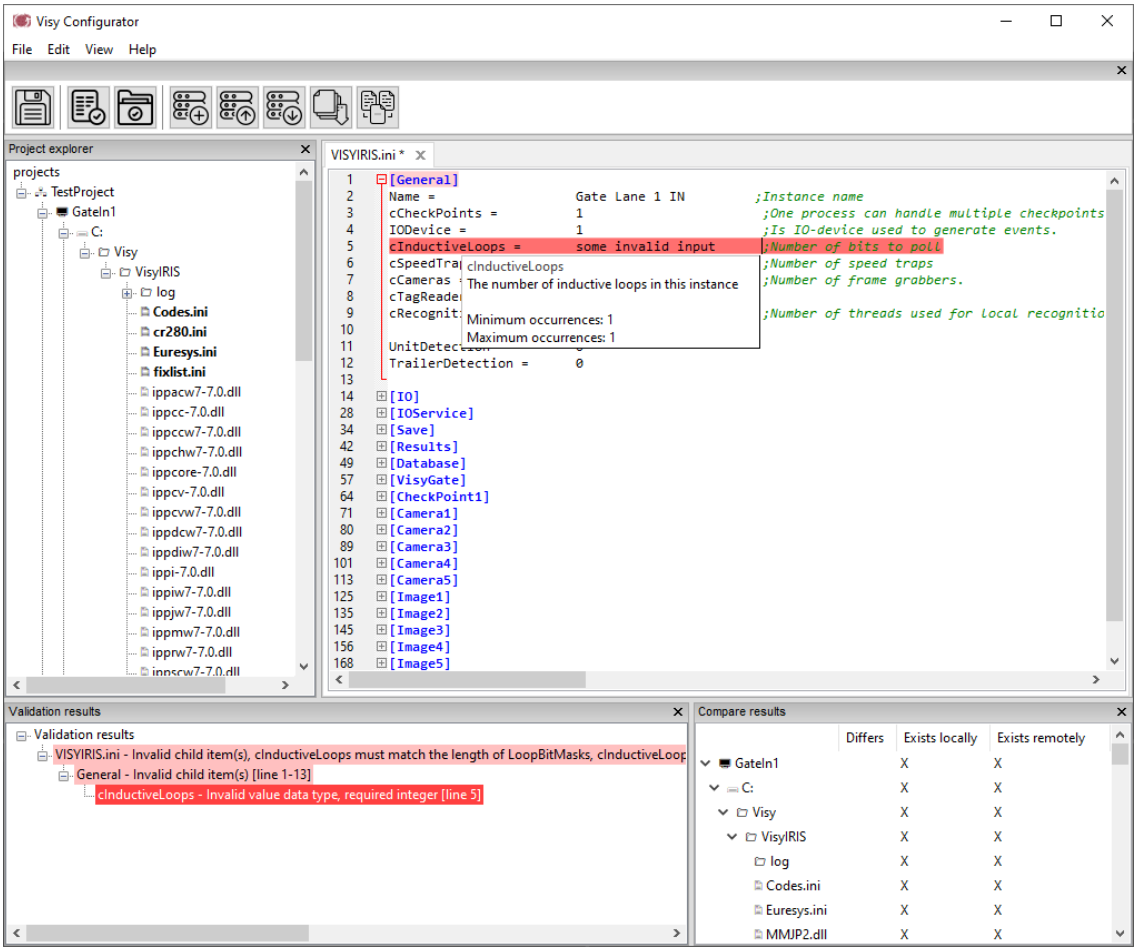

Figure 7.3. *New Visy Configurator user interface.*

Switching to a completely different user interface library went smoothly without any major changes to the existing interfaces in the peripheral components such as configuration validation and configuration schema interface. Consequently, it proved that one of the design goals of the architecture was at least partially met: Decoupling software component implementations increased their maintainability.

7.2 Configuration validation

After a sufficient study about available third-party INI file validation libraries, it was concluded that none of them provided all the functionality required for validating INI files used by Visy software. Thus, a custom library for INI file validation was created using the pre-existing Visy code for parsing INI files as part of the implementation to ensure compatibility with all software that utilize the same component for reading configuration files.

The configuration schema was decided to be implemented as multiple XML files that

```
<?xml version="1.0"?>
<applicationDefinition name="TestApplication.exe">
  <container name="files">
    <fileDefinition name="Main.ini">
      <container name="parameterGroups">
        <parameterGroupDefinition name="Group" minOccurs="0" maxOccurs="1">
          <container name="parameters">
            <parameterDefinition name="Param" minOccurs="1" maxOccurs="1">
              <description>Human-readable text</description>
              <restriction base="integer">
                <minInclusive value="10"/>
                <maxInclusive value="99"/>
              </restriction>
            </parameterDefinition>
            <parameterDefinition name="AnotherParam">
              <description>Human-readable text</description>
              <restriction base="float"/>
            </parameterDefinition>
          </container>
          <validations>
            <validation name="Param must be greater than AnotherParam">
              <condition>value(parameters/Param)>value(parameters/
                        AnotherParam)</condition>
            </validation>
          </validations>
        </parameterGroupDefinition>
      </container>
    </fileDefinition>
  </container>
</applicationDefinition>
```

Program 7.1. *Example of an application schema definition.*

would inherit some features from other schema systems such as XSD [47]. There would be one file for each executable and library, so depending on the project, only the necessary schema files need to be packaged along Visy Configurator. Program 7.1 shows an example of how the configuration schema of an application called TestApplication.exe could be defined.

The parameter data types and value restrictions are defined in the restriction element, similar to what XSD has [47], as shown in Program 7.1. Additionally, there's also another element addressing the restrictions on dimensional properties of the value. This element defines the available array dimensions of the parameter: scalar, vector or matrix and how small or big they can be.

Program 7.1 also shows that the configuration item occurrence limits are defined with the minOccurs and maxOccurs attributes similar to how they are defined in XSD [47]. These values are then used to validate the occurrences of the items.

In addition to the aforementioned simple restrictions that only consider one configuration item at a time, there is also support for more complex restrictions that are referred to as *custom validations* in this thesis. Custom validations allow defining validation conditions for configuration items that must be met for the compared configuration items included in the validation conditions. Program 7.1 shows an example of such validation inside the validations element. The custom validations utilize the Interpreter design pattern discussed in Chapter 2 to implement the abstract syntax tree evaluation. Here, the context of the interpreter is the validated INI file.

At the start of execution of the application, the configuration schema is parsed into instances of different kinds of schema object classes that all implement the same interface. This allows generating a tree-like structure of polymorphic schema objects, implementing the Composite design pattern [10].

7.3 Remote host management

A couple of different alternative solutions were considered at first for implementing the remote host management functionality. Table 7.1 shows a comparison of remote host management solutions according to the acceptance criteria for Visy Configurator, in the order of most important to least important. In the table, *3rd-party app.* denotes third-party configuration management software applications such as Puppet or Chef that were discussed briefly in Chapter 3. The row "No additional software installed" is marked with parentheses for libssh because installing OpenSSH server is natively supported by Windows, enabled from the system settings user interface, but it still requires an internet connection and needs to download the software separately. On Linux systems, SSH client and server are both installed by default in many distributions.

Table 7.1. *Remote host management solutions comparison.*

	3rd-party app.	WinRM	libssh
Execute commands remotely	✓	✓	✓
Transfer files	✓	✓	✓
Encrypted	✓	✓	✓
Agentless	✓	✓	✓
C/C++ API available		✓	✓
Authentication without password input	✓	✓	✓
No additional software installed		✓	(✓)
Cross-platform	✓		✓

Using already existing configuration management software could have been an alternative to Visy Configurator for remote management, but that would always require installing such software separately and writing appropriate scripts for each system. This was con-

sidered to be in conflict with the design constraint about ease of deployment mentioned in Chapter 5 and would also provide inferior usability compared to a dedicated user interface specifically designed for Visy systems. Also, the lack of C++ API in those solutions degraded their fitness to be used by the application.

Another way to do remote management would have been using WinRM protocol through the WinRM Client Shell API, which allows initiating a remote shell connection with another Windows computer [59]. This could have been considered an appropriate partial solution for the short term, but it would only support systems running on Windows. As was mentioned earlier in Chapter 5, source code portability was also kept as a desirable attribute for the software.

Finally SSH was chosen as the protocol to utilize for remote management. The software component used for the actual implementation was libssh, an open source library providing both C and C++ APIs. The remote command execution was conducted using SSH and remote file transfer was implemented using multiple concurrent SSH File Transfer Protocol (SFTP) sessions, speeding up the transfers drastically compared to a single SFTP session transferring files in a serial manner.

The full commit process consists of the following steps:

1. Initialize a connection to the remote machine.
    - Open a new SSH session by using public key authentication.
2. Kill all Visy processes
    (a) Call `net stop` for each service whose name starts with "Visy".
    (b) Call `taskkill` for the rest of currently running processes whose name start with "Visy".
3. Uninstall all Visy services.
    - Call `sc delete` for each service whose name starts with "Visy".
4. Copy all files to the remote machines and remove files that do not belong there.
    (a) Open an SFTP session for traversing, creating and deleting the remote directories recursively.
    (b) Open a new SSH and SFTP session for each file, running the file transfers in parallel.
5. Install all Visy services.
    - Call `sc create` for each Visy service executable.
6. Start all Visy services.
    - Call `net start` for each installed Visy service.
7. Start all Visy applications.

   (a) Call `schtasks` to create new scheduled tasks for starting desktop applications on Windows startup.
   (b) Run the scheduled tasks once to start the applications.

The full fetch process consists of the following steps:

1. Initialize a connection to the remote machine.
   - Open a new SSH session by using public key authentication.
2. Copy all files from the remote machines to the local computer.
   (a) Open an SFTP session for traversing the remote directories and creating local directories recursively.
   (b) Open a new SSH and SFTP session for each file, running the file transfers in parallel.

8 CONCLUSIONS AND FUTURE WORK

The goal of this thesis was to design an architecture for a software configuration management system and evaluate the designed architecture while also establishing background information with literature review around the discussed topics. The motivation to design such system was to facilitate commissioning and maintaining systems provided by Visy Oy to be more automatic and precise.

All software has some sort of architecture, be it intentionally designed or not. Software architecture can be inspected from various viewpoints and defined in various different ways. Designing software architecture carefully can – and in most cases does – produce software with better quality attributes than software that is just quickly assembled without much effort in the design. For example, keeping different concerns in separate components makes software more maintainable.

The set of all possible different architectures for the configuration management system with the specified requirements is practically infinitely vast – the vastness also depending on the definition of what is still considered as part of architecture rather than its implementation. The architecture of Visy Configurator was designed with the best understanding and experience the architect had at that time, and fulfilling the requirements could have been solved in a multitude of different ways.

Configuration management as a term can have different meanings, depending on the context where the term is used. In this thesis, when talking about implementing configuration management in the designed system, the definition mostly resembles system administration: installing and removing software and updating their preference values on various computers.

On the other hand, configuration management can also be perceived in the bigger picture, as shown in Figure 3.1, that was introduced in Chapter 3. Implementing and using Visy Configurator will achieve and further support some of these configuration management elements. Configuration validation aspect of the software supports both auditing the system and status accounting. Change control is supported by the automated remote management procedures. Revision control shall be implemented later using a proper version control system, which will also support status accounting. The identification element gets support from the configuration schema and how the user interface is able to show the documentation of each configuration item in it.

Evaluation of the designed architecture was conducted with only four persons of whom

no-one had any earlier experience of architecture evaluations. All of the reviewers were also employees of Visy Oy, which in turn probably caused some bias towards the habits of the company. Having some external reviewers might have yielded very different results compared to the ones achieved during the DCAR session: External reviewers do not have the same bias that internal reviewers do, though they might have many other biases that would still balance the opinions about the architecture decisions to a more objective direction.

Nevertheless, the evaluation was still a lot better than not evaluating at all, and at least it gave some perspective to whether the designed architecture was any good or bad, while introducing the reviewers to the DCAR process that may be used later to evaluate another piece of software as well. All in all, reviewing the architecture using DCAR turned out to be a good choice, especially because of its light weight required by the small amount of both reviewers and time.

Visy Configurator is already being carefully tested in one production system. So far, it has been working as expected. However, there are still many new ideas to be implemented for the software. The future plans for Visy Configurator include, for example,

- designing and implementing the version control part of the software
  - This feature is probably implemented as a Git repository in the local filesystem.
  - Version control would allow the user to see the differences between versions in the local filesystem and execute an easy rollback to an earlier version.
- re-implementing the remote management commit process to utilize self-destructing packages instead of issuing each command and transferring each file separately
  - Put all changed files in a compressed package and include in it a script that takes care of issuing the required commands and eventually destroying the package itself.
  - This approach is safer in case of connection failure during the commit process: Because the actual modifications to the managed host are done locally and only applied once the whole package is copied, the danger of leaving the host in an undefined state is lower.
  - Provides higher system availability, because the actual downtime of the software is lower: Time to decompress a package probably takes less time than to transfer each file individually over SFTP.
- generating configuration item templates based on the configuration schema using the configuration factory component
  - The act of configuring a system with specific constraints on its parameters becomes a constraint satisfaction problem (CSP)[60]. Automatic configuration data imputation could benefit from CSP algorithms, not just imputing default values but values that actually satisfy the constraints as well. [61]

- designing and implementing automatic schema inference from existing INI files and program code
  - Implementing an inference engine similar to what Huang et al. introduce in [35] would be beneficial for avoiding time-consuming manual schema writing as much as possible. Without an inference engine, it is more likely that the software developers consider editing the schema too tedious, which in turn might cause them to skip updating it to reflect the changes they make in program code.
- designing and implementing other new features
  - For example, it might be useful to have a graphical tool that generates configuration items based on the user's drawings about regions of interest in images that define the limits for the computer vision algorithms.

Eventually, Visy Configurator will become obsolete – like all software do. Once Visy as a company grows even bigger and the amount of installations with it, perhaps utilizing alternative third-party solutions for configuration management will become more relevant. If the Visy software infrastructure is refactored to better fit such third-party solutions, deciding to delegate software configuration management to a third party within Visy systems might be a valid choice in the future.

In the future, it is also possible that Visy software will no longer utilize INI file format for configuration files, but rather use other more standardized ones, such as XML or JSON. Strictly standardized file formats currently have, for example, better support for nested structures and more third-party solutions available for serialization and validation, providing the opportunity to delegate those features of Visy Configurator to a third party as well.

If I started to design a similar system again, I would search more thoroughly how the INI files and their contents are used in the whole Visy codebase. Designing the configuration schema structure and the configuration schema interface component was not as trivial as I thought it would be. The configuration files themselves were simple, but how they were used by different pieces of software turned out to be quite complex. Designing the system to properly support configuration documentation and validations was iterated over a couple of times, because every once in a while more perplexing ways to use the configuration items were found.

Some of the results of this thesis can be generalized into other software projects as well. Conducting architecture evaluations on software architectures is a good idea and worth the time used for it, especially if the designed software system is huge in size, is in an important role for the using organization or is supposed to have a long lifetime. Software architecture evaluations can give early warnings about poor design decisions, and proactive issue resolving is easier than reactive.

Also, using separation of concerns in software architecture has been discovered to be a

generally good practice already long ago, and using the same practice in the designed configuration management system also turned out to yield preferable results in terms of how maintainable and modifiable the software was.

All things considered, the aforementioned goals of this thesis were met. Future work includes perfecting the configuration management system to its full potential.

REFERENCES

[1] Vogel, O., Arnold, I., Chughtai, A. and Kehrer, T. *Software Architecture: A Comprehensive Framework and Guide for Practitioners*. eng. Berlin, Heidelberg: Springer Berlin / Heidelberg, 2011. ISBN: 9783642197352.

[2] *ISO/IEC/IEEE 42010 Systems and software engineering — Architecture description. Defining architecture*. URL: http://www.iso-architecture.org/42010/defining-architecture.html (visited on 03/13/2020).

[3] Bass, L., Merson, P., Bachmann, F., Stafford, J., Clements, P., Nord, R., Garlan, D., Ivers, J. and Little, R. *Documenting software architectures: views and beyond*. eng. SEI series in software engineering. Addison-Wesley Professional, 2010. ISBN: 9780321552686.

[4] Meunier, R., Stal, M., Rohnert, H., Buschmann, F. and Sommerlad, P. *Pattern-oriented software architecture: a system of patterns*. eng. Wiley Software Patterns Series. Wiley-Blackwell, 2013. ISBN: 0471958697.

[5] Perry, D. E. and Wolf, A. L. Foundations for the study of software architecture. eng. *Software engineering notes* 17.4 (1992), 40–52. ISSN: 0163-5948.

[6] *OMG Unified Modeling Language (OMG UML)*. Version 2.5.1. Dec. 2017. URL: https://www.omg.org/spec/UML/2.5.1/PDF (visited on 03/17/2020).

[7] Brown, S. *The C4 model for visualising software architecture*. URL: https://c4model.com/ (visited on 10/02/2020).

[8] *ISO/IEC 7498-1:1994. Information technology — Open Systems Interconnection — Basic Reference Model: The Basic Model*. June 15, 1996. URL: https://standards.iso.org/ittf/PubliclyAvailableStandards/index.html (visited on 04/15/2020).

[9] Parziale, L., Britt, D. T., Davis, C., Forrester, J., Liu, W., Matthews, C. and Rosselot, N. *TCP/IP Tutorial and Technical Overview*. 8th ed. International Business Machines Corporation, Dec. 2006. URL: https://www.redbooks.ibm.com/redbooks/pdfs/gg243376.pdf (visited on 04/27/2020).

[10] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. eng. 1st ed. Addison-Wesley Professional, 1994. ISBN: 0201633612.

[11] Meyers, S. *Effective STL*. eng. 1st ed. Addison-Wesley Professional, 2001. ISBN: 9780201749625.

[12] Reijonen, V., Koskinen, J. and Haikala, I. Experiences from Scenario-Based Architecture Evaluations with ATAM. eng. *Software Architecture*. Vol. 6285. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, 214–229. ISBN: 3642151132.

[13] Decision-Centric Architecture Reviews. eng. *IEEE Software* 31.1 (2014), 69–76. ISSN: 0740-7459.

[14] Bass, L. and Nord, R. L. Understanding the Context of Architecture Evaluation Methods. eng. IEEE, 2012, 277–281. ISBN: 146732809X.

[15] Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H. and Carriere, J. The architecture tradeoff analysis method. eng. IEEE, 1998, 68–78. ISBN: 0818685972.

[16] Erder, M. *Continuous architecture : sustainable architecture in an agile and cloud-centric world*. eng. Amsterdam, [Netherlands: Morgan Kaufmann. ISBN: 0-12-803285-5.

[17] Dobrica, L. and Niemela, E. A survey on software architecture analysis methods. eng. *IEEE transactions on software engineering* 28.7 (2002), 638–653. ISSN: 0098-5589.

[18] Patidar, A. and Suman, U. A survey on software architecture evaluation methods. eng. Bharati Vidyapeeth, New Delhi, 2015, 967–972. ISBN: 9380544154.

[19] Heesch, U. van, Avgeriou, P. and Hilliard, R. Forces on Architecture Decisions - A Viewpoint. eng. IEEE, 2012, 101–110. ISBN: 146732809X.

[20] *IEEE Standard for Configuration Management in Systems and Software Engineering*. eng. 2012.

[21] Configuration Management Fundamentals. eng. *CrossTalk, The Journal of Defense Software Engineering* (July 2005).

[22] Arundel, J. *Puppet 3 Beginner's Guide*. eng. 1st ed. Packt Publishing, 2013. ISBN: 9781782161240.

[23] *What is configuration management?* URL: https://www.redhat.com/en/topics/automation/what-is-configuration-management (visited on 05/22/2020).

[24] *The Benefits of Agentless Architecture*. 2018. URL: https://www.ansible.com/hubfs/pdfs/Benefits-of-Agentless-WhitePaper.pdf (visited on 07/01/2020).

[25] Senner, J. *Agent vs Agentless with SSH. What about agentless configuration management (CM)?* Jan. 27, 2015. URL: https://www.orcaconfig.com/blog/agent-vs-agentless-ssh/ (visited on 07/01/2020).

[26] Hunter, T. and Porter, S. *Google Cloud Platform for Developers: Build Highly Scalable Cloud Solutions with the Power of Google Cloud Platform*. eng. Birmingham: Packt Publishing, Limited, 2018. ISBN: 1788837673.

[27] *Declarative vs. Imperative Models for Configuration Management: Which Is Really Better?* June 29, 2020. URL: https://www.upguard.com/blog/declarative-vs.-imperative-models-for-configuration-management (visited on 07/03/2020).

[28] Brikman, Y. *Why we use Terraform and not Chef, Puppet, Ansible, SaltStack, or CloudFormation*. Sept. 26, 2016. URL: https://blog.gruntwork.io/why-we-use-terraform-and-not-chef-puppet-ansible-saltstack-or-cloudformation-7989dad2865c (visited on 10/16/2020).

[29] Yigal, A. *Chef vs. Puppet: Methodologies, Concepts, and Support*. Jan. 16, 2017. URL: https://logz.io/blog/chef-vs-puppet/ (visited on 10/11/2020).

[30] *Git*. URL: https://git-scm.com (visited on 10/29/2020).

[31] *Apache Subversion*. URL: https://subversion.apache.org/ (visited on 10/16/2020).

[32] *Mercurial SCM*. URL: https://www.mercurial-scm.org/ (visited on 10/16/2020).

[33] *Concurrent Versions System - Summary*. URL: http://savannah.nongnu.org/projects/cvs (visited on 10/16/2020).

[34] Dubie, D. *How much will you spend on application downtime this year?* Aug. 2, 2009. URL: https://www.networkworld.com/article/2261272/how-much-will-you-spend-on-application-downtime-this-year-.html (visited on 08/25/2020).

[35] Huang, P., Bolosky, W., Singh, A. and Zhou, Y. ConfValley: a systematic configuration validation framework for cloud services. eng. EuroSys '15. ACM, 2015, 1–16. ISBN: 9781450332385.

[36] Yin, Z., Ma, X., Zheng, J., Zhou, Y., Bairavasundaram, L. and Pasupathy, S. An empirical study on configuration errors in commercial and open source systems. eng. SOSP '11. ACM, 2011, 159–172. ISBN: 9781450309776.

[37] Newman, L. H. *How a Tiny Error Shut Off the Internet for Parts of the US*. June 11, 2017. URL: https://www.wired.com/story/how-a-tiny-error-shut-off-the-internet-for-parts-of-the-us/ (visited on 08/24/2020).

[38] Graham-Cumming, J. *Details of the Cloudflare outage on July 2, 2019*. July 12, 2019. URL: https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/ (visited on 08/24/2020).

[39] Graham-Cumming, J. *Cloudflare outage on July 17, 2020*. July 18, 2020. URL: https://blog.cloudflare.com/cloudflare-outage-on-july-17-2020/ (visited on 08/24/2020).

[40] Facebook. *Yesterday, as a result of a server configuration change, many people had trouble accessing our apps and services. We've now resolved the issues and our systems are recovering. We're very sorry for the inconvenience and appreciate everyone's patience.* Mar. 14, 2019. URL: https://twitter.com/facebook/status/1106229690069442560 (visited on 08/24/2020).

[41] *Summary of the December 24, 2012 Amazon ELB Service Event in the US-East Region*. URL: https://aws.amazon.com/message/680587/ (visited on 08/25/2020).

[42] Treynor, B. *Today's outage for several Google services*. Jan. 24, 2014. URL: https://googleblog.blogspot.com/2014/01/todays-outage-for-several-google.html (visited on 08/25/2020).

[43] Attariyan, M. and Flinn, J. *Automating configuration troubleshooting with dynamic information flow analysis*. 2010. URL: https://www.semanticscholar.org/paper/Automating-Configuration-Troubleshooting-with-Flow-Attariyan-Flinn/e3ad98bea50b82859655fdfaac9755c6f951899f (visited on 08/24/2020).

[44] Chomsky, N. *Syntactic structures*. eng. Janua linguarum, Series minor ; 4. The Hague : Mouton, 1957. ISBN: 90-279-3385-5.

[45] *Theoretical Syntax and Semantics*. URL: https://linguistics.utah.edu/syntaxsemantics.php (visited on 08/25/2020).

[46] Simón, Á. *Definition of validation levels and other related concepts*. July 2013. URL: https://studylib.net/doc/7757314/definition-of-validation-levels-and-other-related-concepts (visited on 08/17/2020).

[47] Gao, S., Sperberg-McQueen, C. M., Thompson, H. S., Mendelsohn, N., Beech, D. and Maloney, M. *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures*. Apr. 5, 2012. URL: https://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/ (visited on 03/30/2020).

[48] Maler, E. *Guide to the W3C XML Specification ("XMLspec") DTD*. Version 2.1. URL: https://www.w3.org/XML/1998/06/xmlspec-report-v21.htm (visited on 07/08/2020).

[49] *ISO/IEC 19757-2:2008. Information technology — Document Schema Definition Language (DSDL) — Part 2: Regular-grammar-based validation — RELAX NG*. Dec. 15, 2008. URL: https://standards.iso.org/ittf/PubliclyAvailableStandards/index.html (visited on 10/27/2020).

[50] Wright, A., Andrews, H., Hutton, B. and Dennis, G. *JSON Schema: A Media Type for Describing JSON Documents draft-handrews-json-schema-02*. Sept. 16, 2019. URL: https://tools.ietf.org/pdf/draft-handrews-json-schema-02.pdf (visited on 08/27/2020).

[51] McLaughlin, B. *Java and XML Data Binding*. eng. Place of publication not identified.

[52] Cruz, P., Salinas, L. and Astudillo, H. Quick Evaluation of a Software Architecture Using the Decision-Centric Architecture Review Method: An Experience Report. *Software Architecture*. Ed. by A. Jansen, I. Malavolta, H. Muccini, I. Ozkaya and O. Zimmermann. Cham: Springer International Publishing, 2020, 281–295. ISBN: 978-3-030-58923-3.

[53] *C++ compiler support*. Mar. 12, 2020. URL: https://en.cppreference.com/w/cpp/compiler_support (visited on 03/13/2020).

[54] *Current Status. Recent milestones: C++17 published, C++20 underway*. URL: https://isocpp.org/std/status (visited on 03/13/2020).

[55] *History of C++*. Feb. 8, 2020. URL: https://en.cppreference.com/w/cpp/language/history (visited on 03/13/2020).

[56] Stroustrup, B. *C++ Applications*. Apr. 9, 2019. URL: http://www.stroustrup.com/applications.html (visited on 03/13/2020).

[57] *C++*. URL: https://github.com/topics/cpp (visited on 03/13/2020).

[58] *Browse Open Source Software. C++*. URL: https://sourceforge.net/directory/language:cpp/ (visited on 03/13/2020).

[59] *WinRM Client Shell API*. URL: https://docs.microsoft.com/en-us/windows/win32/winrm/client-shell-api (visited on 10/02/2020).

[60] Ghedira, K. *Constraint Satisfaction Problems: CSP Formalisms and Techniques*. eng. 1. Aufl. Computer Engineering and IT. Somerset: Wiley-ISTE, 2013. ISBN: 9781848214606.

[61] Yang, D. and Dong, M. Applying constraint satisfaction approach to solve product configuration problems with cardinality-based configuration rules. eng. *Journal of intelligent manufacturing* 24.1 (2013), 99–111. ISSN: 0956-5515.