

Henri Pulkkinen

# **DESIGN OF A SOFTWARE CONFIGURATION TOOL FOR A TEST SYSTEM IN MASS PRODUCTION**

Master of Science Thesis  
Faculty of Engineering and Natural Sciences  
Examiners: Dr. Niko Siltala  
Jyrki Latokartano  
February 2024

## ABSTRACT

Henri Pulkkinen: Design of a software configuration tool for a test system in mass production  
Master of Science Thesis  
Tampere University  
Master's Degree Programme in Mechanical Engineering  
February 2024

---

This thesis is about designing a software tool for a test system to create new configuration files. Target system is used to test touch screen devices after manufacturing. Configuration files are used to describe the device and related calibration hardware. Creating configuration files manually has been a challenge. Manual creation has been a slow and error prone process. Lack of testing capability has allowed some errors to reach production.

Configurator application was designed to help creation of new configuration files. First part of the study analysed existing configuration files to identify what parts of the system are required to be configured when creating a new configuration. Second part concentrated on designing the application through requirements and high-level software design was done to meet these requirements. Eight different configuration items were found. Configuration items covered only 10% of the parameters. Result was a design that combines configured parameters defined in code and rest are read in from a template. The application calculates parameter values from user input and makes sure that all required parameters are set. Validation for configuration is done to catch errors and increase quality. Configuration validation was built on validated data structure by using data classes from Pydantic library.

Keywords: software design, software architecture, configuration management

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

# TIIVISTELMÄ

Henri Pulkkinen: Konfiguraatiohallintatyökalun ohjelmistosuunnittelu testijärjestelmään massatuotannossa  
Diplomityö  
Tampereen yliopisto  
Konetekniikan diplomi-insinöörin koulutusohjelma  
Helmikuu 2024

---

Työssä suunniteltiin ohjelmistotyökalu luomaan uusia konfiguraatitiedostoja testijärjestelmälle, joka testaa kosketusnäyttölaitteita. Konfiguraatitiedostoja käytetään kertomaan testijärjestelmälle millainen testattava laite ja siihen liittyvät kalibrointityökalut ovat. Uusien konfiguraatitiedostojen luonti on ollut haastavaa. Niitä on luotu käsin, joka on ollut hidasta ja vikaherkkää. Puutteellisten testaustapojen myötä myös virheellisiä konfiguraatioita on päässyt tuotantoon. Se laskee parametreille arvot käyttäjän syötteestä ja varmistaa, että kaikki tarvittavat parametrit ovat annettu. Käyttäjän syötteet sekä lopullinen konfiguraatio validoidaan virheiden estämiseksi.

Konfigurointisovellus suunniteltiin avustamaan uusien konfiguraatioiden luomisessa. Tutkimuksessa perehdyttiin ensin olemassa oleviin konfiguraatioihin, joista etsittiin tarvittavat konfiguroitavat kokonaisuudet. Sen jälkeen suunniteltiin sovellus tekemällä vaatimusmäärittely sekä korkean tason ohjelmistosuunnittelu vastaamaan näitä vaatimuksia. Konfiguraatitiedostoista löydettiin kahdeksan konfiguroitavaa kokonaisuutta. Näiden kokonaisuuksien sisältämät parametrit ovat 10% koko konfiguraation sisällöstä. Ohjelmistosuunnittelun tuloksena on sovellus, jossa konfiguroidaan edellä mainittu 10% parametreista ja yhdistetään muuttumattomien parametrien kanssa. Se laskee parametreille arvot käyttäjän syötteestä ja varmistaa, että kaikki tarvittavat parametrit ovat annettu. Käyttäjän syötteet sekä lopullinen konfiguraatio validoidaan virheiden estämiseksi. Konfiguraation validointi ja yhdistäminen toteutettiin validoidulla tietorakenteella, jossa käytettiin Pydantic -kirjaston dataluokkaa.

Avainsanat: ohjelmistosuunnittelu, ohjelmistoarkkitehtuuri, konfiguraationhallinta

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck -ohjelmalla.

## **PREFACE**

Thanks to Lukasz Walach and Vili Saura for providing me peace from day-to-day work to work on this, Niko Siltala and Jyrki Latokartano for comments and guidance how to structure the report, my co-workers and Sanni for advices during the writing process.

Now, a year later it's nice to see the progress I have made during the whole thesis process.

Tampere, 29th January 2024

Henri Pulkkinen

# CONTENTS

1.	Introduction . . . . .	1
1.1	Research questions and restrictions . . . . .	1
1.2	Research methods . . . . .	2
2.	Software design and configuration management . . . . .	3
2.1	Software design . . . . .	3
2.1.1	Software architecture . . . . .	3
2.1.2	Software requirements . . . . .	4
2.1.3	Object oriented design . . . . .	5
2.1.4	Design patterns . . . . .	5
2.1.5	SOLID design principles . . . . .	8
2.1.6	Modeling . . . . .	10
2.2	Configuration management . . . . .	13
2.2.1	Software configuration management . . . . .	14
2.2.2	Configuration file . . . . .	14
2.2.3	Parametrization . . . . .	15
2.2.4	Data validation . . . . .	15
2.2.5	Pydantic . . . . .	16
3.	Target system . . . . .	18
3.1	OptoFidelity's TOUCH test system . . . . .	18
3.2	Software description . . . . .	19
3.3	Challenge . . . . .	22
3.4	Identifying configuration items . . . . .	23
3.4.1	Method to identify configuration items . . . . .	23
3.4.2	Analysing configuration files . . . . .	24
3.4.3	Analysis result . . . . .	26
3.4.4	Result summary . . . . .	29
4.	Designing configurator . . . . .	31
4.1	Overview . . . . .	31
4.2	Requirements . . . . .	31
4.3	High-level design . . . . .	32
4.4	Software design . . . . .	37
4.5	Example of validated data model usage . . . . .	45
5.	Discussion . . . . .	48
5.1	Findings . . . . .	48

5.2 Further development . . . . .	49
5.3 Validity and generalizability . . . . .	50
6. Conclusion . . . . .	51
References . . . . .	52

## LIST OF SYMBOLS AND ABBREVIATIONS

API	Application Programming Interface
Bring-up	Process of taking a system into use.
CI	Configuration Item
CLI	Command Line Interface
CM	Configuration Management
Diff	Utility to compare differences between files in UNIX-like operating systems.
DUT	Device Under Test
End effector	Device attached to a robot allowing it to physically interact.
Fixture	Test fixture where device under test is placed to.
GUI	Graphical User Interface
I/O	Input/Output
IDE	Integrated Development Environment
MP	Mass Production
SCM	Software Configuration Management
Smoke testing	Testing software to verify it starts and core functionality is working.
TnT	Touch and Test is OptoFidelity Ltd's test system software.
UML	Unified Modeling Language

# 1. INTRODUCTION

OptoFidelity's TOUCH test system's software is parameter driven. When a new product is introduced to be tested by the system, it needs to be described for the software via configuration files. Creating new configuration manually has been a challenge due to convoluted nature of the configuration files. Testing correctness of the configuration has been difficult due to lack of product specific hardware in development system, which is used to test the configuration before reaching production. A configurator tool was proposed to solve issues by automatically handling and validating parameters of the configuration.

Existing configuration management tools are concentrating to manage IT equipment and servers. Suitable software to manage the system was not available. In previous studies Rintala (2021) and Piipari (2013) has come to similar conclusion that a custom tool needs to be created. Tool designed in this study further separates from a conventional configuration management system, because parameters are calculated from user input and validated depending on business rules of the test system.

## 1.1 Research questions and restrictions

How to accomplish successful design of a software configuration tool is studied in two parts. First, it's needed to know what is in the configuration files to be able to model it in software. Second part is to find a suitable software design. Research questions are:

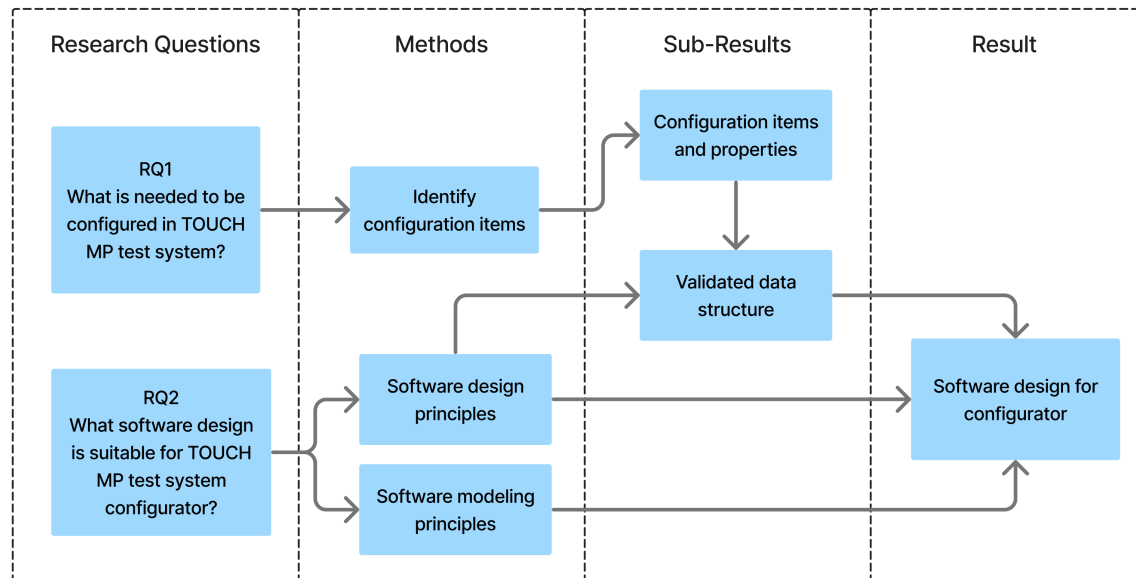
- RQ1 What is needed to be configured in TOUCH MP test system?
- RQ2 What software design is suitable for TOUCH MP test system configurator?

Scope of this work is to do a high level design for a configurator software. Goal was to do a design that can be handed out to software development team for implementation. Some restrictions came from mass production environment. Changes to the system, including the configuration files, parameters or configuration file structure wasn't allowed. Implementation constraints had to be considered during the design. Company's coding conventions needed to be followed and recommended programming language Python to be used.

## 1.2 Research methods

To answer RQ1, thematic analysis is used to find what is needed to configure and what kind of properties they have. Thematic analysis is a method of qualitative data analysis that involves identifying, analyzing and reporting patterns or themes within. It was chosen to be suitable method for understand the context and properties of the file by giving insight of underlying meanings and values of the content by highlighting the similarities and differences among different sections and elements. (Villegas n.d.)

To find an answer to RQ2, software design and modeling principles were used to derive to a suitable software design. These are presented in chapter 2.1. Together with, resulting configuration items from RQ1 and software design method, a validated data structure was designed. Diagram of research questions, methods and sub-results are in shown figure 1.1



**Figure 1.1.** Research questions and methods.

Theoretical background is presented in chapter 2. TOUCH test system and its configuration is introduced in chapter 3. Findings from configuration are used, and software designed in chapter 4. Result discussion and study conclusions are in chapters 5 and 6, respectively.

## **2. SOFTWARE DESIGN AND CONFIGURATION MANAGEMENT**

This chapter gives first an overview on software design and architecture topics, then configuration management and configuration files.

### **2.1 Software design**

Design is a process where information requirements are used to form a representation of data, program structure, interface definition and process details (Pressman 2001). It's about making major decisions to make software's structure coherent and well planned. Focus can be on very different levels. Main factors are interrelations of higher level components and logical operations on lower level. (Freeman and Wasserman 1983)

There isn't actual difference between software design and architecture. Word architecture is commonly used for the high level design, and word design is used for low level design. High level design choices supports low level design choices and vice versa, so there isn't clear separation between them on the continuum of design choices from higher to lower level. (R. Martin 2017)

Software design can be viewed as a process from defining the problem in the form of software requirements. Then some solution, from infinite set of solutions, is declared as a solution to solve those problems. The process is iterative because designers move back and forth between activities of requirements, finding possible solutions, testing solutions and documenting. (Pfleeger and Atlee 2006)

#### **2.1.1 Software architecture**

Software architecture is about organization and overall structure of the software. It ties together requirements engineering and design by defining main structural components and relationships between them. Software architectures can be divided in two abstraction levels that share the same idea of decomposing into components, but in different scales. In smaller scale it's about individual program and structure of data and components within. In larger scale it's about systems that is composed of programs or even other systems.

(Sommerville 2016)

Every application has an architecture regardless of whether it was designed or not. It's the principal design choices that the application builds on. (Taylor et al. 2010) Description of an architecture is separate from the architecture itself, it's something that the designers do to document the intended architecture. (Qin et al. 2008) Architecture and architecture description can exist independently, which means that documenting it is an important part to present it (Bass et al. 2021). Important role of an architecture description is to enable communication between stakeholders. (Koskimies and Mikkonen 2005)

Function of an architecture is to describe essential design decisions. Including description of the system, division to components, communication between components, functionality of components, processes, information flow, capacity, performance, re-usability, maintenance etc. (Koskimies and Mikkonen 2005) Architecture is an abstraction that shows certain details and suppresses others, but details about implementation level is not shown. (Bass et al. 2021)

Architecture design is followed by detailed design and implementation. One of its important roles is constraining quality attributes of the system, such as performance, reliability and maintainability. Quality of a software is a part of non-functional requirements, which makes architecture design to respond more on the non-functional requirements rather than functional requirements. Architecture design allows for stakeholders to communicate on common matters early in the development process. (Bosch 2000) Earliest design decisions reflect the whole system and have effect on system's implementation in a way that it's almost impossible to change afterwards (Qin et al. 2008). Changes to the system in early phase are cheaper (Koskimies and Mikkonen 2005).

### **2.1.2 Software requirements**

Software requirement is a software capability needed to solve a problem that is needed to accomplish some objective. The requirement must be fulfilled by the target system to perform up to standard or specification. (Dorfman and Thayer 1990) Requirements don't attain detail about design or implementation. Requirements are in a problem domain, and should be set so that they don't constraint the solution domain of developers. (Leffingwell and Widrig 2003). Discovering right requirements lets developers to concentrate on solving right problems and prioritize features. (Wiegers 2009)

Setting requirements often drives the outcome of a software project. Getting requirements wrong can lead to a failure of the project. Managing requirements is a process of eliciting, organizing and documenting the requirements to provide and maintain agreement on what the software should do. For a small project where amount of requirements is close to ten, the management process is trivial and not actually needed. For a bigger project it

becomes mandatory to keep up with requirements that will change during life cycle of the software. (Leffingwell and Widrig 2003)

There are three different types of requirements: functional, non-functional and constraints. A functional requirement describes a functionality or service that the system should provide. It can define how to react to a certain input in certain state. Basically it tells what the system should do. (Sommerville 2016) Non-functional requirements describe quality attributes that the system should have. It can include for example requirements for performance, security, maintainability, etc. (Pfleeger and Atlee 2006) They aren't directly tied to a certain service of the system (Sommerville 2016). Constraints can be set prior to development if there is a known choice or limitations in requirements elicitation phase. Constraints can define some design or implementation decision, e.g. platform or framework that is needed to be used. (Leffingwell and Widrig 2003)

### **2.1.3 Object oriented design**

Object oriented design is one of the programming paradigms and it's well suited for a wide range of applications (Koskimies 2000). Other common paradigms today are procedural programming and functional programming (University of Helsinki n.d.). Object oriented design resembles and models the view in way that is natural to humans. We can see objects and entities with attributes and behaviour all around us. Object's attributes can be manipulated with functions. Important characteristics of object oriented design is encapsulation of data and functions to manipulate it. It's well suited to organize code so that the code is reusable. Reusable code is critical part of software engineering. It leads to faster delivery times and higher software quality. Decoupled structure allows changes to be made with less side effects. (Pressman 2001) Challenge of a good object oriented design is decomposing the system into objects. Many factors like encapsulation, granularity, dependency, flexibility, evolution, re-usability, etc. need to be assessed. (Gamma et al. 1994)

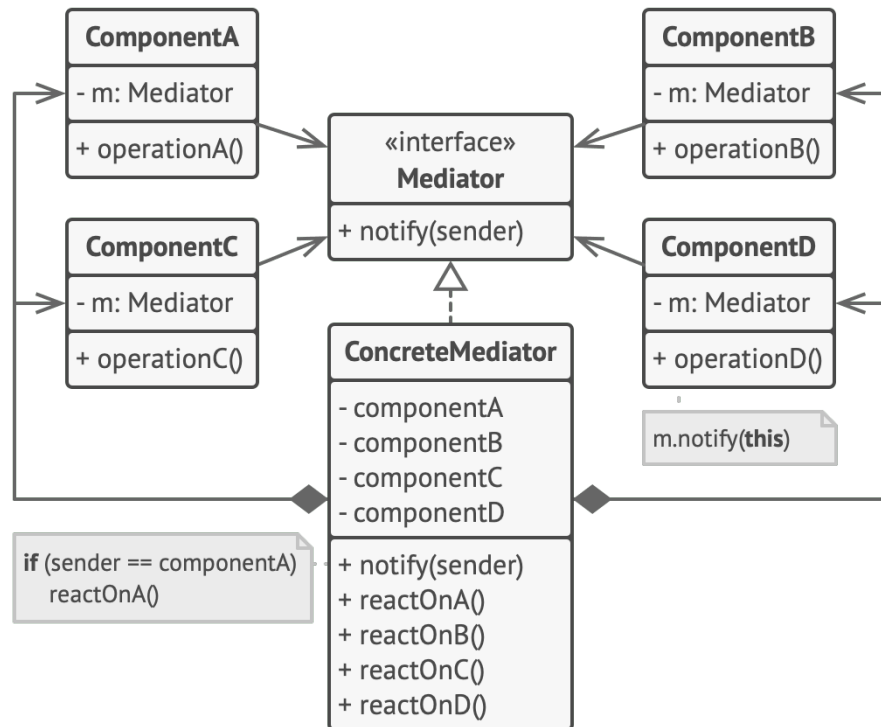
### **2.1.4 Design patterns**

Design patterns collect existing and proven experience in software engineering. Pattern is a proven solution framework for a specific recurring problem in software design or implementation. They can be used when selected properties of software architecture is needed. (Buschmann 1996) A pattern describes a recurring problem and the core of a solution to that problem. The recurring problem and solution of a design pattern stays higher level, so the problem can be solved many times without ever doing it exactly the same way (Alexander et al. 1977). This is because the problem and solution are described in a higher level and they can be applied to variety of levels and implemented in different

ways. Design patterns enables to reuse solutions and established common terminology to communicate about design decisions by giving a higher-level perspective of the problem and solutions to it. (Shalloway and Trott 2004) Building analogy for a design pattern could be, that proven designs like single-room and two-room apartments exists to answer to specific needs, and there are different ways these can be built.

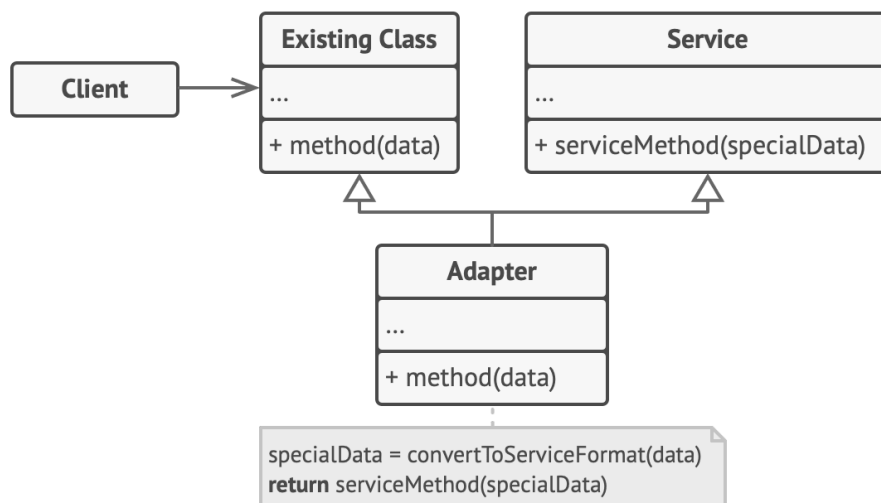
There are three categories of design patterns, creational, structural and behavioral. Creational patterns, like Factory Method, Prototype and Singleton, abstracts instantiation process. Aim is to make the system independent of how objects are created, composed and represented. Structural patterns like Adapter, Composite and Proxy, are about how to compose classes and objects to form larger structures. Behavioral patterns like Mediator, Command and Iterator, are concerned with algorithms and responsibility assignment between objects. (Gamma et al. 1994) There are many more patterns in each category suited for different problems and purposes.

One of the many patterns, for example, is Mediator. It's a behavioural pattern and its intention is to reduce dependencies between objects. If components must communicate with each others directly, they become dependent on the other components. Mediator pattern solves this with a mediator object that the components use to communicate with each other. Figure 2.1 shows an example class structure. All components use mediator object to send a notification about state changes. Then mediator object can pass this information to other components. Benefits are that components doesn't need to know about other components and business rules of what happens on each component's notification is encapsulated in the mediator. (Gamma et al. 1994; Refactoring.Guru n.d.)



**Figure 2.1.** Mediator pattern class diagram. (Refactoring.Guru n.d.)

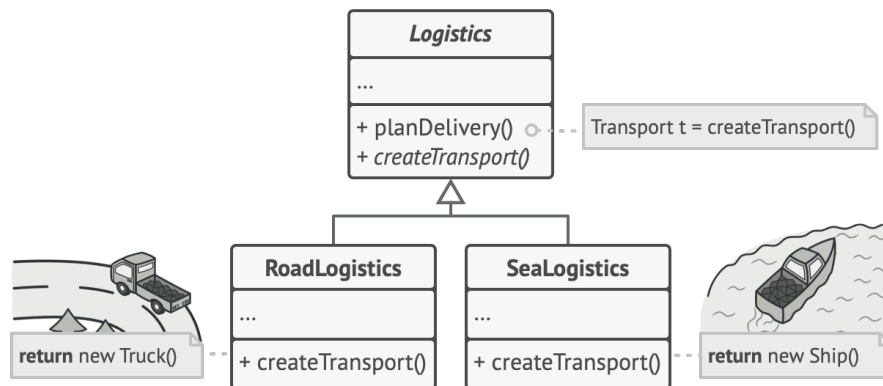
Adapter pattern is a structural pattern that can adapt different interfaces to work together. Figure 2.2 has *Client* that is using *Existing Class* to do operations on data. *Service* class has a different interface than *Existing Class*. In order for the *Client* to use *Service* class, it needs to be adapted by *Adapter* class. (Refactoring.Guru n.d.)



**Figure 2.2.** Adapter pattern class diagram. Adapted from (Refactoring.Guru n.d.).

Factory Method, also known as Virtual Constructor, is a creational pattern. It lets subclass to control creation of an object. It has the advantage that the exact type of created object

doesn't need to be known. Figure 2.3 has *Logistics* class that has an interface for creating transportation with *createTransport* function. The client code can use this function to create a transportation, without having to know the type of transportation.



**Figure 2.3.** Factory pattern class diagram. (Refactoring.Guru n.d.).

### 2.1.5 SOLID design principles

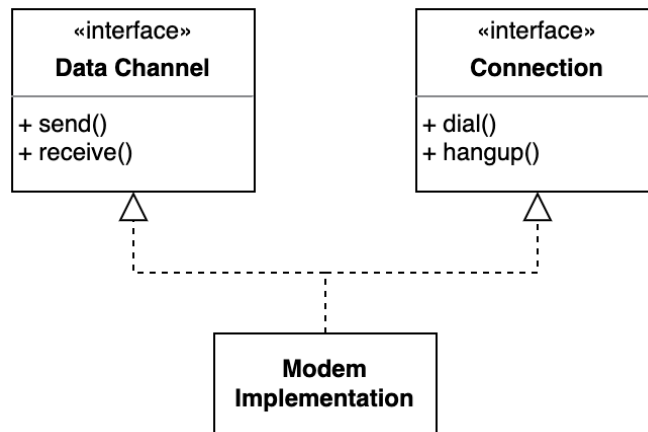
Depending on the given business problem, identifying classes and objects then deciding interaction between them can become a complex task. Filling classes with methods they shouldn't belong to can make the software inflexible for changes. SOLID principles are fundamental guidelines to build object-oriented software. Intention is to make a class structure that leads to a robust, extensible and maintainable code base. It also provides common terms to communicate with the team about different design choices. SOLID is an acronym from the five different rules: (Joshi 2016)

- **Single responsibility principle,**
- **Open/closed principle,**
- **Liskov substitution principle,**
- **Interface segregation principle,**
- **Dependency inversion principle.**

It describes good practices in a smaller scale than design patterns. Building analogy could be that a proven pattern is to build bathroom and shower in the same room.

**Single responsibility principle's** goal is a separation of concerns. It states that every class should have only one responsibility. Class is like a container that can be filled with any amount of data and methods. Trying to achieve too much with one class can lead to a big class that need to be changed and tested for any change in business rules. Single responsibility principle suggest to split different responsibilities to different classes. This way, changes in one responsibility stays in one place without affecting other functionality. (Joshi 2016)

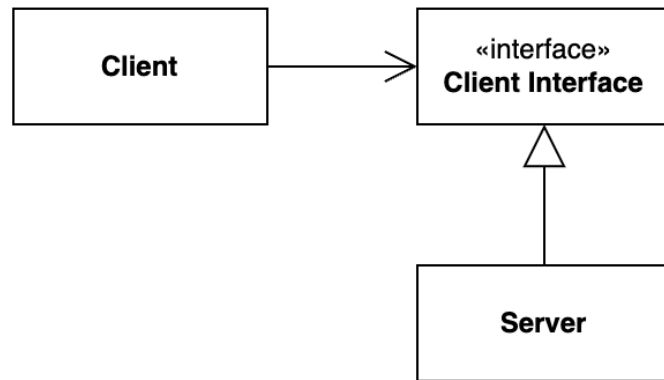
Example modem class could include methods for connecting and sending data in a single class. By following single responsibility principle in figure 2.4, the responsibilities of managing connection and managing data are split into own classes. (R. Martin and M. Martin 2007)



**Figure 2.4.** Example of applying single responsibility principle. Adapted from (R. Martin and M. Martin 2007, Ch. 8 Fig. 8-3).

In **open/closed principle**, open means that a class should be open for extension, and closed means that it should be closed for modifications. It means that after creating and using a class in different parts of an application, it should not be changed. Changes to that class might very well break the system. In order to implement new functionality, the class should be extended instead of modified. Result of extension is that existing parts of the application can't see the change and new functionality is easy to test because it's isolated. (Joshi 2016)

Open/closed principle is used in figure 2.5 by making an interface class for a client. If the client would use server directly, changes for server would need changes also for client. By making them dependent on interface module, allows changes that don't break the interface, to be made. (R. Martin and M. Martin 2007)



**Figure 2.5.** Example of applying open/closed principle. Adapted from (R. Martin and M. Martin 2007, Ch. 9 Fig. 9-2).

**Liskov substitution principle** says that a derived class should be able to replace its base class. When a class inherits some other class, it can be extended but it should not break the functionality of the base class. (Joshi 2016) Violating this principle often results to implementation that does runtime type checking to adapt to child class's behaviour. This leads to violation of the open/closed principle. (R. Martin and M. Martin 2007)

**Interface segregation principle's** idea is that a client of an interface class should not depend on other clients needs through the interface. This can be the case if one big interface is used by different clients that need different functionality. Change need to the interface by one client's need forces updates to all clients. Interface segregation principle states that the interface should be split to serve different clients. (Joshi 2016)

**Dependency inversion principle** says that a high-level class should not depend on low-level class. High-level class should depend on abstract classes and interfaces. This is to avoid high-level classes to be tightly coupled to a low-level class to omit need of change if the low-level class need to be changed. (Joshi 2016)

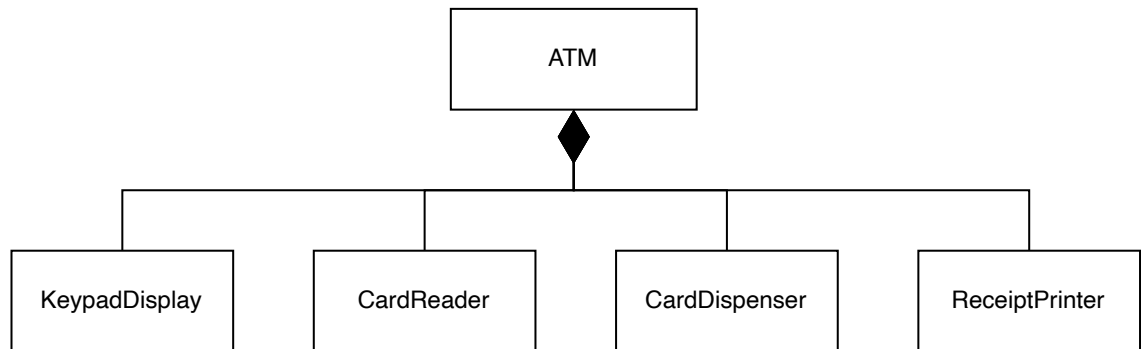
### 2.1.6 Modeling

Modeling a system is important part to manage complexity. A model is an abstraction of the system by abstracting irrelevant details away for given context. It's a simplification of the system to allow it to be understood. (Miles and Hamilton 2006)

Unified Modeling Language (UML) is a widely used standard for modeling software. UML is a modeling language, a graphical notation to present concepts. It can present many different views like (Gomaa 2011)

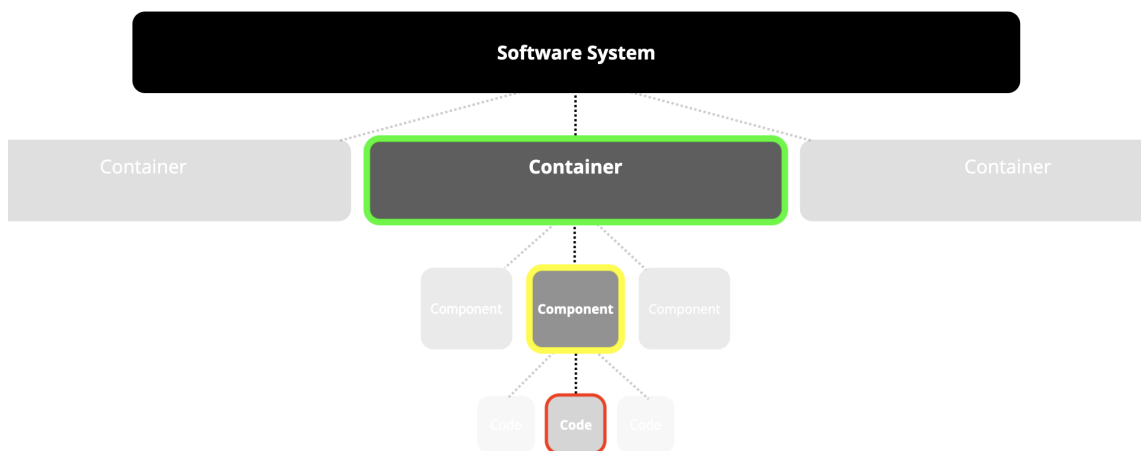
- use case view to show requirements with the system and actors using it,
- static view to show class diagrams,
- dynamic interaction view to show objects and communication,

- dynamic state machine view to show how the system behaves in different states
- and more.



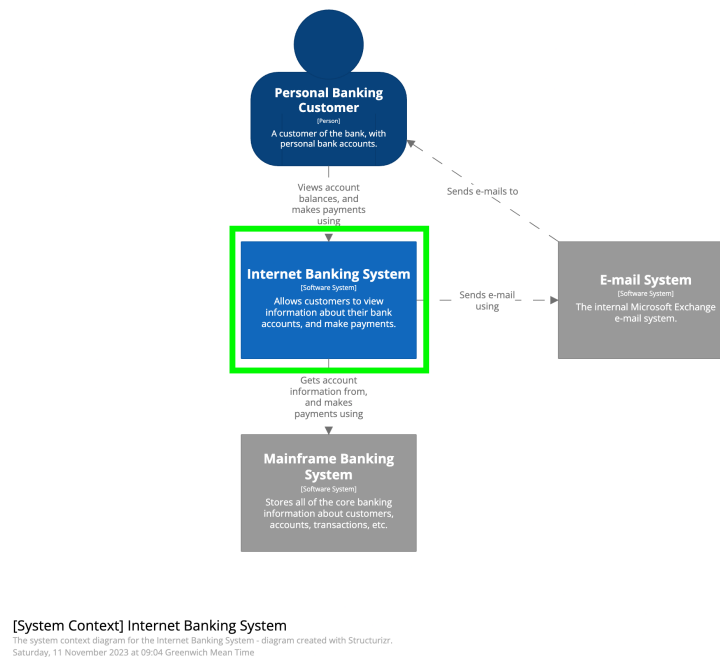
**Figure 2.6.** Example static view to show class diagram. Adapted from (Gomaa 2011).

C4 is a model for visualising software architecture. It's made of four different hierarchical levels of abstraction: context, containers, components and code. In figure 2.7 shows this in abstract level. There software system is the context, which is followed by containers, components and code to describe this system. These levels are like an internet map service that reveal more detail while zooming in. (Brown n.d.)



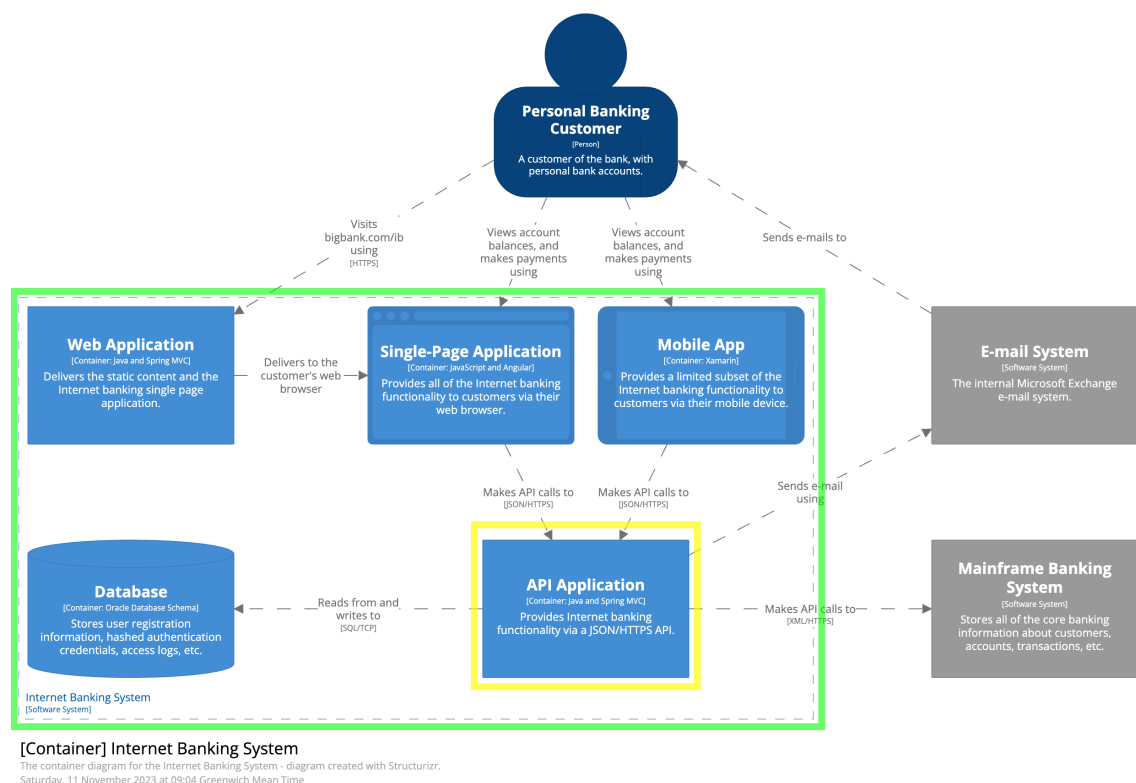
**Figure 2.7.** Abstract view of C4 model. Adapted from (Brown n.d.).

Abstraction first approach presents context diagram 2.8 first. It shows how actors like user, and other software systems, act with the software system in question. (Brown n.d.)



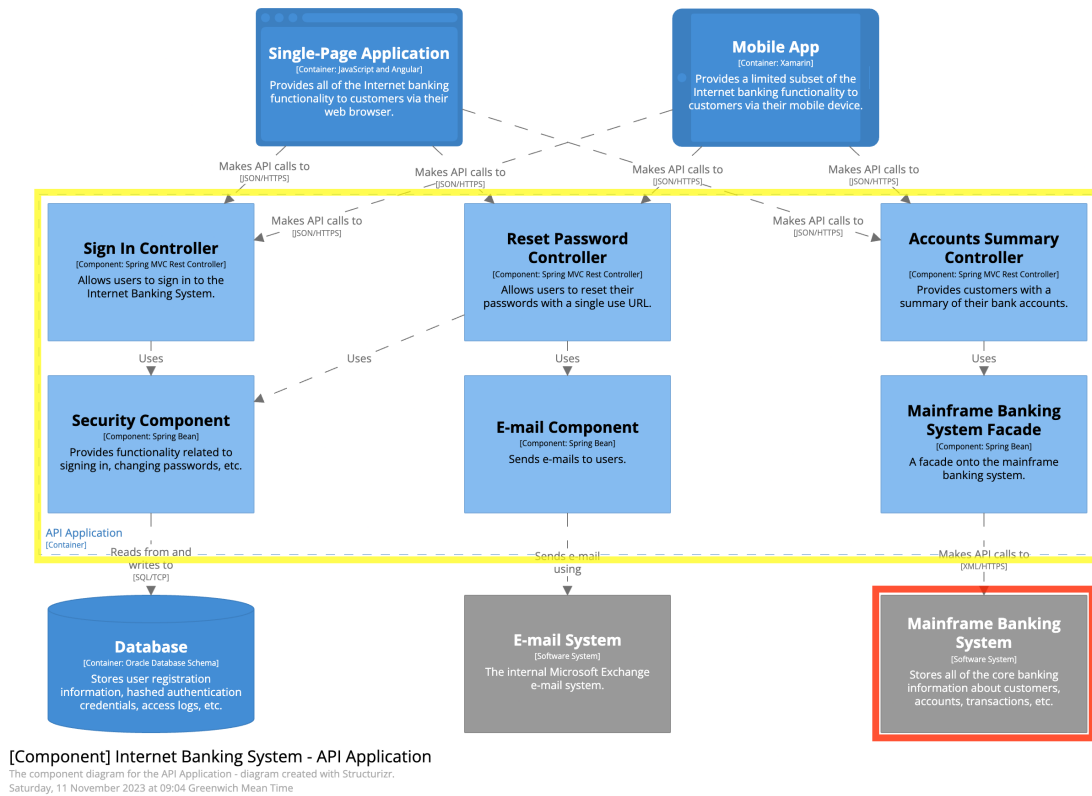
**Figure 2.8.** System context view of C4 model. Adapted from (Brown n.d.).

Container view 2.9 zooms in to reveal high-level architecture of the software system by showing distinct parts, like applications and databases, and how they communicate with each other. (Brown n.d.)



**Figure 2.9.** Container view of C4 model. Adapted from (Brown n.d.).

Components view 2.10 shows what modules each application contains. Finally code context shows how modules are implemented with class diagrams. Code level is typically shown using UML. C4 model endorses simple and clear communication inside and outside of development team. Code level detail is optional and recommended to only show the important parts. (Brown n.d.)



**Figure 2.10.** Component view of C4 model. Adapted from (Brown n.d.).

## 2.2 Configuration management

Configuration management (CM) is an engineering discipline focusing on change management. Basis for a CM process is to improve quality and integrity of code, tools, documents and virtually anything that is needed to manage. Benefit of CM is to help manage changes by understanding the changes and providing consistent way to reproduce different versions. Four fundamental values for configuration management are (Moreira 2010)

- **Identification** of configuration items (CIs) establishes a baseline of software-related items to control them. CIs can be plan, requirements, specifications, designs, source code, executables, tools, system information, test cases, etc.
- **Control** changes to all configuration items. Sub-processes for control are version control, change control, build management and release engineering.
- **Audit** is ability to provide correctness, completeness and consistency of baseline. It's function is to determine is the configuration and changes aligned with physical

and functional specification.

- **Reporting** status of components. It's a form of communication helping to understand what has changed and how.

### 2.2.1 Software configuration management

Software configuration management (SCM) is a discipline to manage projects and synchronize work for different developers in the project. It provides support for entire development life cycle of a product. SCM is established by defining processes and methods, preparing plans and usage of tools, to help development and management. (Crnkovic et al. 2003) Configuration management is identifying, organizing and controlling modifications to the software. SCM can be viewed in many ways and roles from developer to management, but basic functionality covers (Babich 1986)

- version management,
- configuration selection,
- concurrent development,
- distributed development,
- build management,
- release management,
- change management,
- integration with other tools.

Previous list shows that it's a term that covers many different areas of software life cycle. SCM context in this study is in configuration selection.

### 2.2.2 Configuration file

Configuration file defines options, settings and preferences applied to applications, operating systems and IT infrastructure devices (Bigelow 2023). It allows customized interaction with an application, or how the application interacts with other systems (Kenlon 2021). Configuration of software or hardware devices can be complex by having lots of parameters and options. These could be either in a single or multiple configuration files. For example, different aspects like storage and networking can be split into separate files. On the other hand, managing multiple files can be difficult. (Bigelow 2023) Configuration files are often human readable and can be modified with a text editor. Common formats are INI, XML, YAML and JSON. (Bigelow 2023)

Following example of an INI formatted configuration has two sections, *General* with a parameter *Version* and *Features* with parameters to enable features *A* and *B*.

```

; Comments starts with a semicolon
[General]
Version = 1.0.0
[Features]
; The features to enable or disable: True or False
FeatureA = True
FeatureB = True

```

**Listing 2.1.** Example in INI format.

Same configuration in XML format has same information and structure with different looks.

```

<!-- Comments start with a hyphen and a bracket -->
<General>
  <Version>1.0.0</Version>
</General>
<Features>
  <FeatureA>True</FeatureA>
  <FeatureB>True</FeatureB>
</Features>

```

**Listing 2.2.** Example in XML format.

### 2.2.3 Parametrization

Parametrized software refers to the adaptation of software to the desired range of functions by setting parameters. Parametrized software allows for customization and flexibility in how the software operates based on the parameters set by the user or another program. This can be particularly useful in scenarios where different users or use-cases require different functionalities from the same software. (Johner 2019) Parametrized programming enables code reuse by having programs in general form that can be applied to different cases (Goguen 1984). Downside is that it requires more effort in testing, because parameters can have different values (Fisher 2017).

Parametrization can be achieved e.g. with plugins, building different version of software, or like in this work it's about setting parameters in configuration files. Example of configuration file parametrization can be seen in above configuration listings 2.2.2, where *FeatureA* and *FeatureB* are toggled by setting the parameter *True* or *False*.

### 2.2.4 Data validation

Data validation is the act of checking integrity, accuracy and structure of data before it's used. Different types of validation can be made to ensure that the data meets its

requirements. (Kerner 2022) A few different types of validation are (Astera Software 2023; SoftwareTestingHelp 2023)

- Presence check to verify that all required parameters exist.
- Business rules to check that data follows required conditions like "if A exists, then B can not exist".
- Format check to make sure data such as date is in expected format.
- Consistency check to catch inconsistent data, e.g. order date must be before shipping date.
- Length check to ensure correct number of characters.

### 2.2.5 Pydantic

Pydantic is a parsing and data validation library for Python programming language. In simplest form, pydantic uses Python's type hints for validation. Additional validator methods can be attached or custom validator can be defined. Pydantic model is a data class that inherits functionality from *BaseModel* class. The following code examples are derived from (Pydantic Services Inc. 2023a). Simple Pydantic model can be defined as

```
from pydantic import BaseModel, validator

class User(BaseModel):
    id: int
    name: str

    @validator("name")
    def name_must_contain_space(cls, v):
        if "_" not in v:
            raise ValueError("must_contain_a_space")
        return v.title()
```

**Listing 2.3.** Definition of a Pydantic model.

where model *User* attributes *id* and *name* are attached with type validators of integer and string respectively, by using Python's type hinting. Custom validator is attached to attribute *name* to check that it contains a whitespace. Model can be used for example to read input from input form or a file

```
input_data = {
    "id": 123,
    "name": "John_Doe",
}
```

```
# Create user from input using keyword argument unpacking
user = User(**input_data)
```

```
print(user.model_dump())
```

**Listing 2.4.** *Instantiation of Pydantic model with input data.*

which outputs following data parsed into the model.

```
"""
{
    'id ': 123,
    'name ': 'John Doe ',
}
"""
```

**Listing 2.5.** *Model attributes printed.*

Creating a model against the validation rules

```
flawed_data = {
    "id": "not_a_number",
    "name": "JohnDoe",
}
```

```
user = User(**flawed_data)
```

**Listing 2.6.** *Instantiation of Pydantic model with flawed data.*

will give an error message.

```
"""
ValidationError: 2 validation errors for User
id
    value is not a valid integer (type=type_error.integer)
name
    must contain a space (type=value_error)
"""
```

**Listing 2.7.** *Validation errors printed.*

As it can be seen from the error message, Pydantic validates the whole model and doesn't stop at first error. (Pydantic Services Inc. 2023a)

### 3. TARGET SYSTEM

This chapter introduces the target system, challenge to solve, and what the tool should do to manage the configuration.

#### 3.1 OptoFidelity's TOUCH test system

OptoFidelity's TOUCH is an automated system for testing touch panels and products with touch-enabled interfaces. Testing and measuring touch interface's characteristics is essential to guarantee quality of a smart device. The standard system consists from a Cartesian motion platform with a brass finger end effector and a positioning camera 3.1. Other end effectors like multiple fingers and force actuator are available. Touch and Test (TnT) software suite is delivered to control the system with a computer. (OptoFidelity Ltd n.d.)

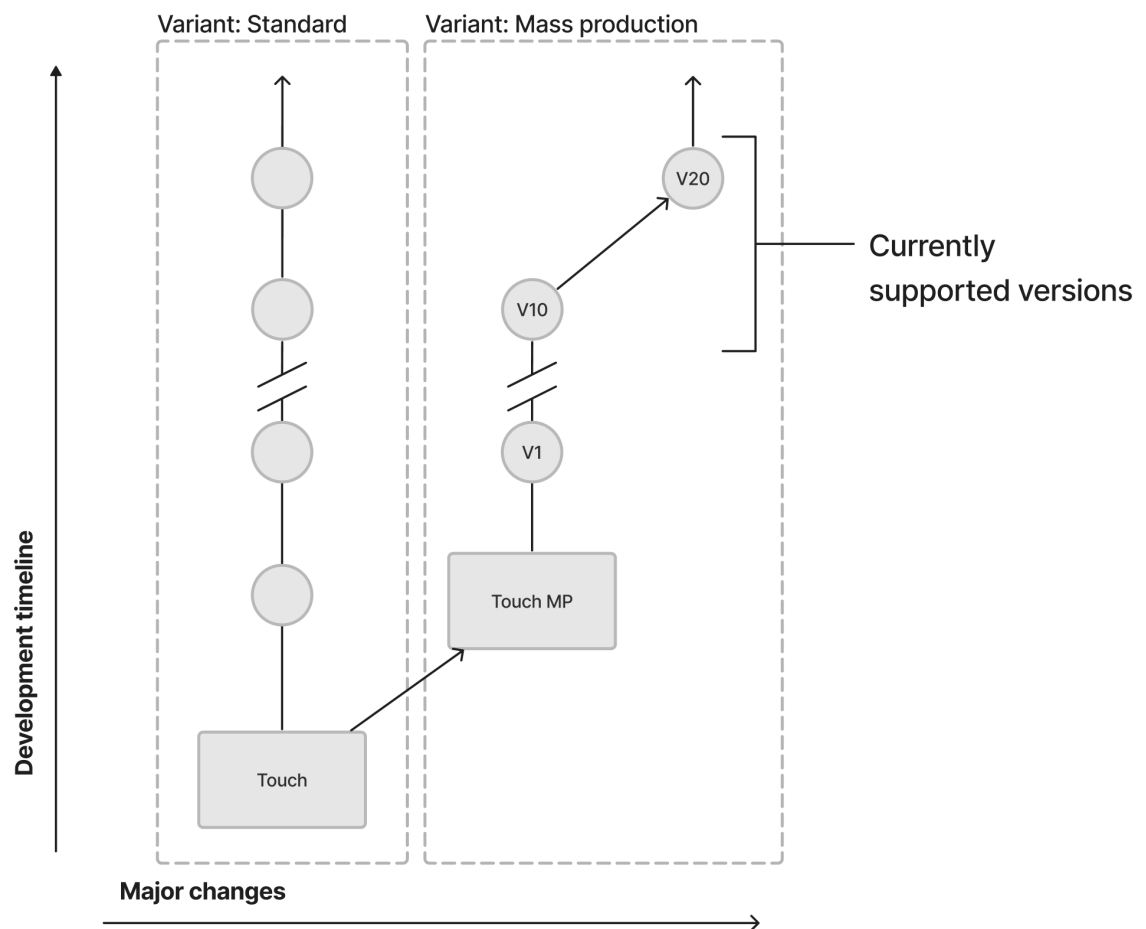


**Figure 3.1.** *OptoFidelity TOUCH test system's motion platform. (OptoFidelity Ltd n.d.)*

Basic use case for the system is to mount a device under test (DUT) on system's measurement table. Tests are prepared by positioning the DUT with a camera, to synchronize movements in DUT's display coordinates. Scripting API is used to control end effector movements on the display. Test movements include tap, press, swipe and many other gestures. Camera is used for measurements which include accuracy, jitter, linearity, reporting rate, resolution, latency, sensitivity and more. (OptoFidelity Ltd n.d.)

Subject of this work is a variant of TOUCH in a mass production (MP) environment testing end products. As illustrated in figure 3.2, the MP variant has been branched off from standard TOUCH development. TOUCH MP has been developed to its 10th version with minor improvements. Next version had major changes done for cost reduction and end of life component management, so version number incremented to V20 to denote the bigger change. Both versions are currently in use and supported by the same software.

Due to non-disclosure agreements some details of features are not published. The MP variant is branched off from TOUCH development. This work doesn't represent the state of standard TOUCH sold currently, still fundamentals are the same. Further discussions pertain to this MP variant.

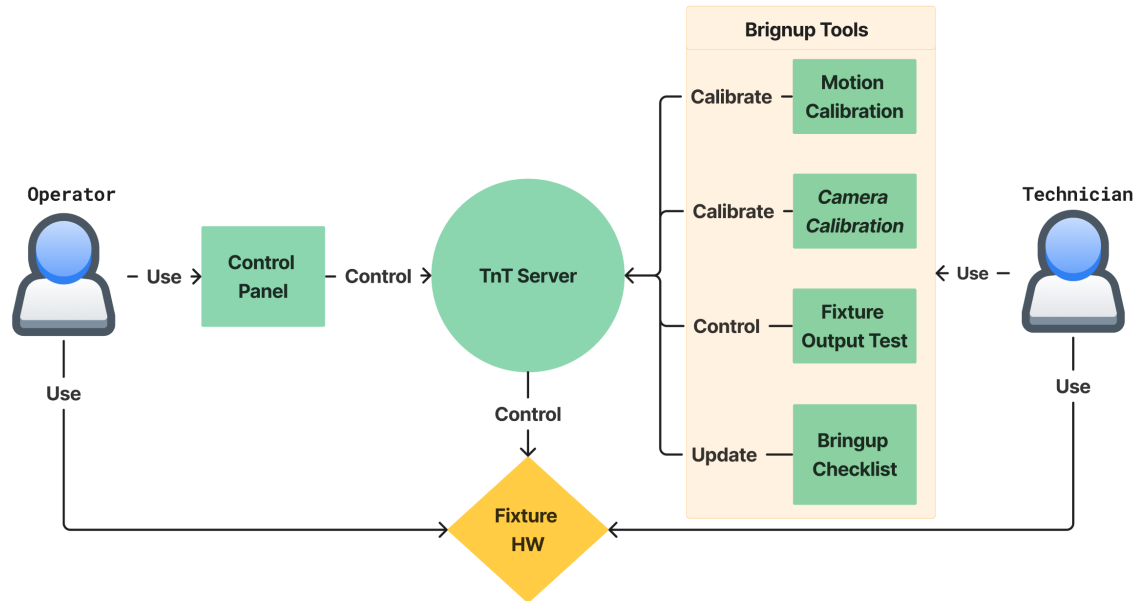


**Figure 3.2.** Touch system version development overtime

## 3.2 Software description

Before operator can start testing, system needs to be prepared by doing a bring-up. Bring-up consist of checking the system's functionality and performing calibrations. TnT software suite is formed from six different software tools. In figure 3.3 software tools are annotated in green and physical test system in yellow. TnT Server is responsible to use

devices like motion platform and camera base on commands it receives. On right side of the figure are tools that bring-up technician uses to prepare the system during bring-up process. On the left side is operator who uses control panel software to do DUT testing.

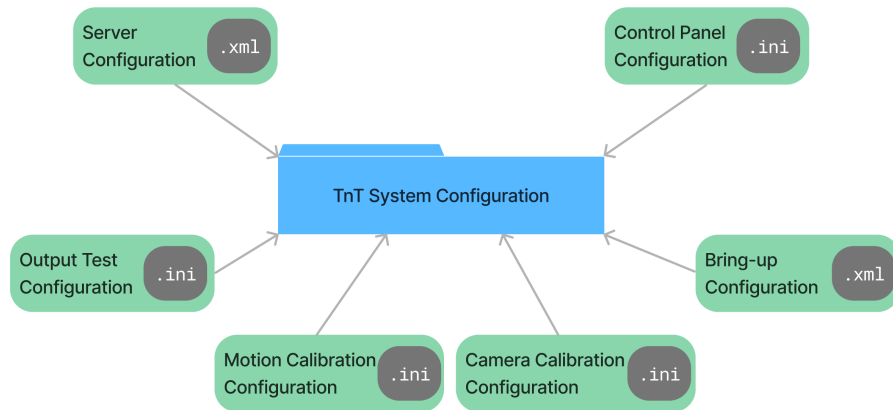


**Figure 3.3.** Components of TnT software.

TnT software needs to act differently based on what hardware it controls and what kind of DUT is tested. For example, TnT server's configuration file includes parameters for properties like

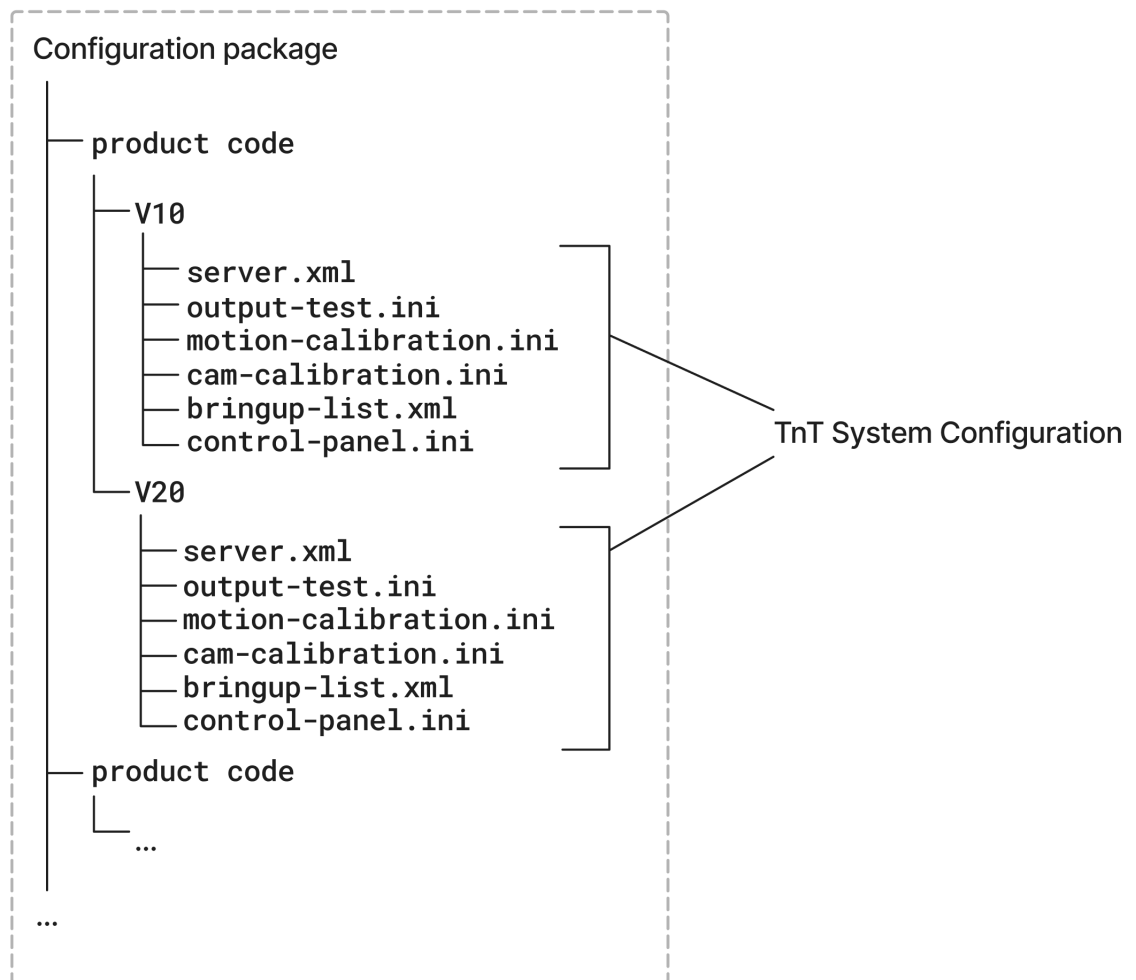
- motion platform dimensions,
- motor characteristics,
- motion controller drivers,
- DUT properties,
- end effectors in use.

These are parameters that are different between test system versions or different DUTs. Configuration files are used to change each software tool's behaviour. TnT system configuration is a collection of configuration files, shown in figure 3.4. Configuration consists of one configuration file for each tool, which are in different file formats. XML and INI files are used.



**Figure 3.4.** Test system's individual component's configuration files form configuration for the system.

TnT system configuration in software release is in configuration package, where installer copies them to correct directories for each software tool to use. Structure of configuration package is shown in figure 3.5. It's a tree structure divided by DUT product codes, which is followed by a test system version specific configurations for that DUT.



**Figure 3.5.** Structure of configuration package.

### 3.3 Challenge

The system needs DUT specific configuration files, so a new configuration is needed upon a new DUT. Most parts of the configuration, like motion platform parameters, remains the same between configurations. New DUT and calibration specific parameters are needed.

Current process of creating a new configuration is following

1. Files are copied from similar DUT
2. Parameters added or removed with text editor using diff utility
3. Dimensions and coordinates measured and calculated from 3D CAD model
4. Dimensions and coordinate parameter values are modified manually
5. Uploading new configuration to version control system
6. Creating alpha software release
7. Installing new software release into the test system for testing.

Main challenges of managing configurations with existing methods are

- calculating coordinates: Configuration includes both absolute and relative coordinates. Due to mechanical differences, absolute coordinate values between V10 and V20 versions are different which increases the amount of coordinate calculation.
- data duplication: Different software tools with separate configuration files need to act on the same hardware. This leads to having same information in more than one place. Misconfiguration happens if all parameters aren't updated.
- manual editing: Manually conducting changes to many places poses a risk for typographical errors.

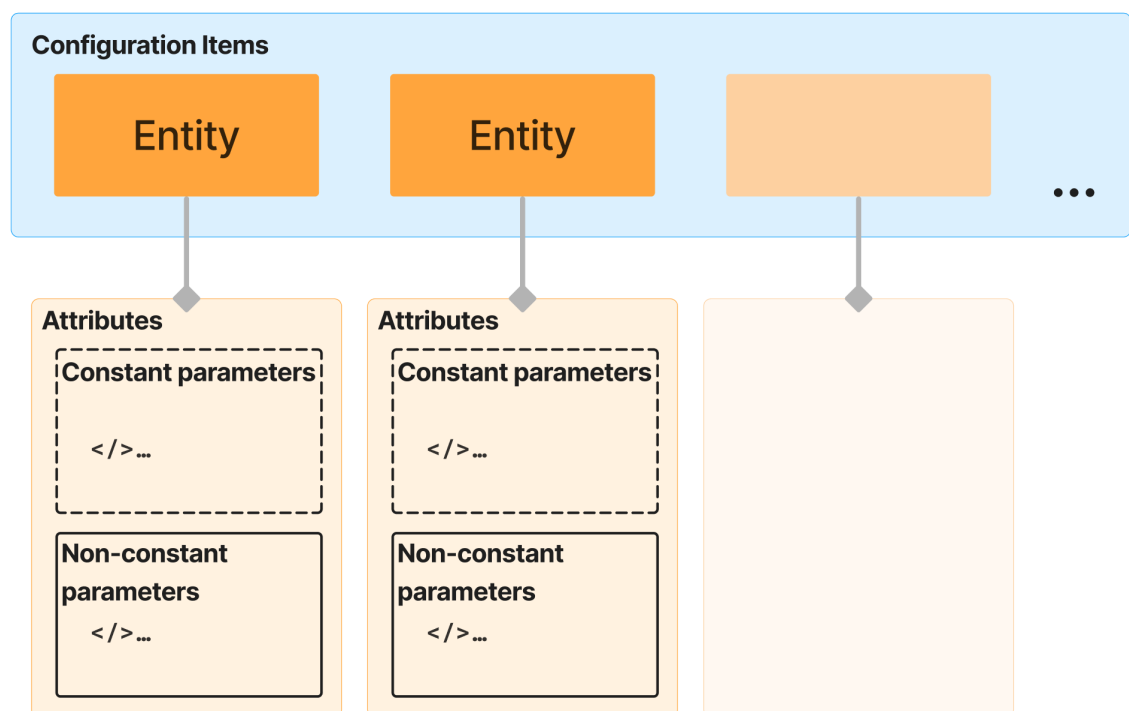
Additional challenge comes from aspect of understanding the system. Person creating a configuration needs a deep understanding of the system. Need of understanding extends to a single parameter level to understand what it does and how it depends on other parameters. Parameters in application level are documented, but information if the same data is in multiple files, is not. Lack of process to follow and trusting individual person's memory to set it in all necessary places introduces a big risk of misconfiguration. According to Miller 1956 human working memory is limited to around seven items.

Feasibility study was done before starting this project. It was estimated that manual creation takes 5 days to do on average. With configurator it's estimated to take half a day, which mainly consist of collecting needed information like physical measurements. In addition to just improve creation time, if validation rules are set correctly, it will greatly improve quality of a configuration release. Previously there has been very limited ability to test configuration parameter values.

### 3.4 Identifying configuration items

Before designing a configuration management system, it's needed to know what to configure. Existing configuration files are studied to identify configuration items and their properties. CI (Configuration Item) in this case was defined to be an entity that can change between configuration packages. Configuration package is a DUT specific set of configuration files that contains parameters for the test system. Configuration file is composed of parameters that are associated with some entity. Entity is something that can exists apart from others, like a calibration tool or a feature that writes an extra log file. Depending on the complexity of an entity, it can contain a single parameter or more.

Figure 3.6 shows an example where a calibration tool would be an entity that is a configuration item that needs to be configured. All its parameters don't need to be configured. In addition to the configurable parameters, it can have constant parameters that remain unchanged. Constant parameter is a parameter that doesn't change between configurations. These are attributes of the system or the DUT than doesn't change, like motion platform dimensions for example.



**Figure 3.6.** Configuration item can have constant and non-constant parameters.

#### 3.4.1 Method to identify configuration items

Parameters in configuration files weren't separated in a way that it would be clear to say what entities they belong to. The sections that they were separated to, serves purpose to the application itself, but not in entity configuration point of view. It was known from

previous configuration work, that majority of parameters are constant and don't need to be configured. Identifying all entities and selecting what's needed to be configured would have led to unnecessary work. Identifying all would be necessary if we were to build a complete configuration management system to control changes to the system, but goal here was to control common changes.

Method for finding configuration items was to look for the smallest element of a configuration item, a parameter that needs to be configured. After finding all parameters that are needed to be configured, they can be grouped to entities they belong to. These entities are the configuration items that the configuration software needs to control.

Three most recent configuration packages were chosen to be material for the analysis. Those three were proven in production and they covered most of the different features. Other configurations were either found to be less reliable or to be too similar with others. To find potential configurable parameters, configuration files inside different configurations were compared to each other using a text editor with diff-functionality. Signs of a parameter of interest was that it's value differs between configurations, or the parameter doesn't exist in all configurations.

### 3.4.2 Analysing configuration files

First step was to find all potential configurable parameters. Parameters and information about them was collected in a table, shown in 3.1. Columns to fill in data were:

- *Application*: Configuration packages are divided into separate files for each application to tell which file the parameter is located in.
- *Parameter name or section*: Parameters were divided into same sections from configuration file.
- *Parameter value in V10*: Values for both system versions.
- *Parameter value in V20*

Application name	Parameter name or configuration file [SECTION]	Parameter value V10	Parameter value V20
OF Test Station	[TEST]		
OF Test Station	use_plc	True	True
Positioning Tool	[GENERAL]		
Positioning Tool	calibrationAbsoluteFirstPositionX	138.0	187.0
Positioning Tool	calibrationAbsoluteFirstPositionY	175.0	226.0
Positioning Tool	blobChessboardOffsetY	-60.0	-71.5
Positioning Tool	[CALIBRATION]		
Positioning Tool	TrayOffsetX	-112.5	-112.5
Positioning Tool	TrayOffsetY	72	72
Positioning Tool	robotAbsoluteFirstPositionX	111.1	161.1
Positioning Tool	robotAbsoluteFirstPositionY	86.6	130.0

**Table 3.1.** A few potential parameters filled into the table.

After all potential parameters were found, a thematic analysis was done to find configuration items. Parameter documentation was used to categorise parameters into system level features. Motivation was to find what parameters were linked together in context of a system feature. It links all parameters related to a feature regardless of layer (HW or SW) or software application it's in. New column "System level feature" were added and all parameters were labeled with a system level feature it's associated to.

Second step was to find parameters that are associated with a distinct hardware component. New column "Hardware entity" was added to track this category. If a change in hardware component would cause a change in parameter, it was labeled. Third step was to label if a parameter was fixture version specific. New column "Common value between fixture versions" was added for labels.

Table 3.2 shows a few parameters. Using fourth lowest parameter *TrayOffsetX* as an example, we can see that its value is same between system versions. *TrayOffsetX* is needed for a system feature called *DUT specific feature 1* and it's an attribute of a hardware component *service tray*. It's value needs to be configured to match the physical

component *service tray*.

Application name	Parameter name or configuration file [SECTION]	Parameter value V10	Parameter value V20	System level feature tag	Hardware entity tag	Value common between versions
OF Test Station	[TEST]					
OF Test Station	use_plc	True	True	DUT SPECIFIC FEATURE 1		COMMON
Positioning Tool	[GENERAL]					
Positioning Tool	calibrationAbsoluteFirstPositionX	138.0	187.0	CAMERA CALIBRATION	CALIBRATION GAUGE PROBE	UNCOMMON
Positioning Tool	calibrationAbsoluteFirstPositionY	175.0	226.0	CAMERA CALIBRATION	CALIBRATION GAUGE PROBE	UNCOMMON
Positioning Tool	blobChessboardOffsetY	-60.0	-71.5	CAMERA CALIBRATION	CALIBRATION GAUGE	UNCOMMON
Positioning Tool	[CALIBRATION]					
Positioning Tool	TrayOffsetX	-112.5	-112.5	DUT SPECIFIC FEATURE 1	SERVICE TRAY	COMMON
Positioning Tool	TrayOffsetY	72	72	DUT SPECIFIC FEATURE 1	SERVICE TRAY	COMMON
Positioning Tool	robotAbsoluteFirstPositionX	111.1	161.1	CAMERA CALIBRATION	CALIBRATION GAUGE PROBE	UNCOMMON
Positioning Tool	robotAbsoluteFirstPositionY	86.6	130.0	CAMERA CALIBRATION	CALIBRATION GAUGE PROBE	UNCOMMON

**Table 3.2.** A few parameters labeled with different categories.

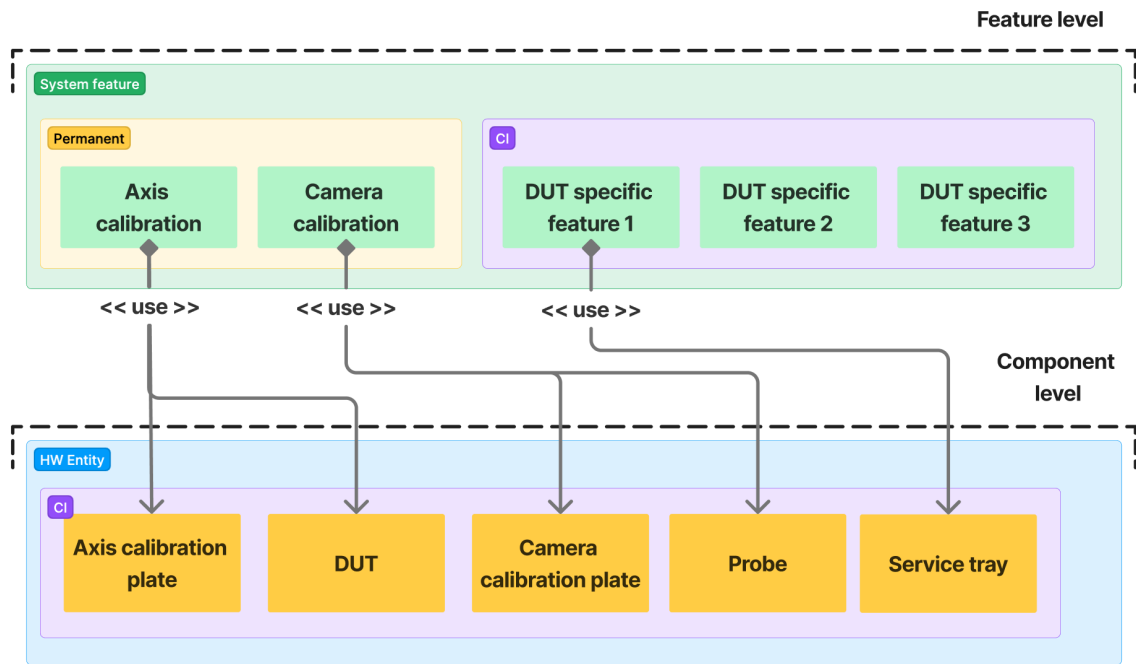
### 3.4.3 Analysis result

Labeled parameter categories was evaluated and adjusted to find suitable entities. Five system features and five hardware entities were found, shown in table 3.3. Two of the system features are always present. Three system features can be toggled based on need, making them configuration items. All hardware entities were configuration items because they had parameter(s) that need to be configured.

**Table 3.3.** Found entities.

Identified entities		
Feature	Type	CI
Axis calibration	System feature	No
Camera calibration	System feature	No
DUT specific feature 1	System feature	Yes
DUT specific feature 2	System feature	Yes
DUT specific feature 3	System feature	Yes
Axis calibration plate	HW Entity	Yes
Calibration gauge	HW Entity	Yes
DUT	HW Entity	Yes
Probe	HW Entity	Yes
Service tray	HW Entity	Yes

It was found that all parameters are related to a system feature, but not necessarily to a hardware entity. Hardware entities are physical components that are used by a system feature, so a system feature is a higher level concept. Relations and hierarchy are visualized in diagram 3.7. If system feature is not present, related hardware entity is not needed. Calibration related system features in left side of top row are always needed. Therefore calibration features are not configuration items and presence of these feature don't need to be controlled. Three other system features were configuration items. These are features that are chosen to be activated according to configuration needs. All hardware entities have parameters that need to be configured. If hardware entity is needed, then it's also needed to be configured. This means that all hardware entities are configuration items.



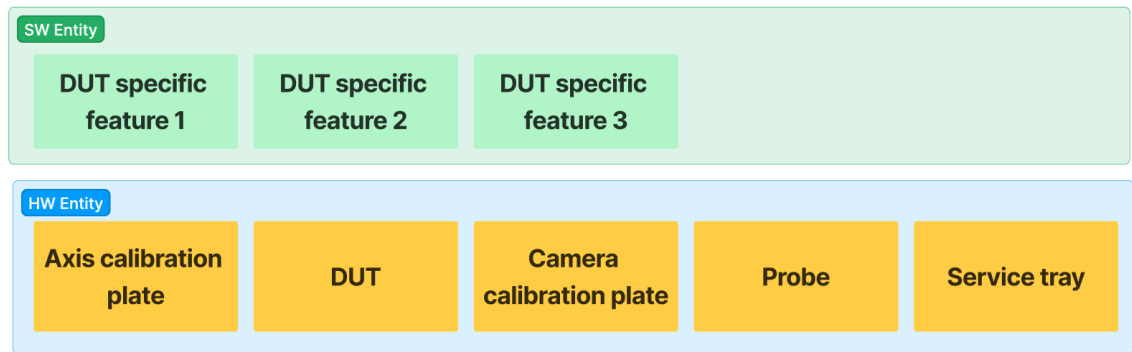
**Figure 3.7.** Found entities from parameters.

Two types of configuration items were identified, system features and hardware entities. System features were something that can be enabled when needed. It requires parameters to be present and set correctly. Diagram 3.7 shows only entities but not parameters. So even if *DUT specific features 2 & 3* don't depend on other entities, they still require constant parameters to be present.

All except one hardware entity is always needed to be defined. Special case is *service tray* that's needed only when *DUT specific feature 1* is enabled. In the other hand, if it's not enabled, presence of the extra parameters doesn't do any harm. Therefore *service tray* entity can be always present to simplify the rules and omit the need to control its presence. With this simplification, all hardware entities are always present.

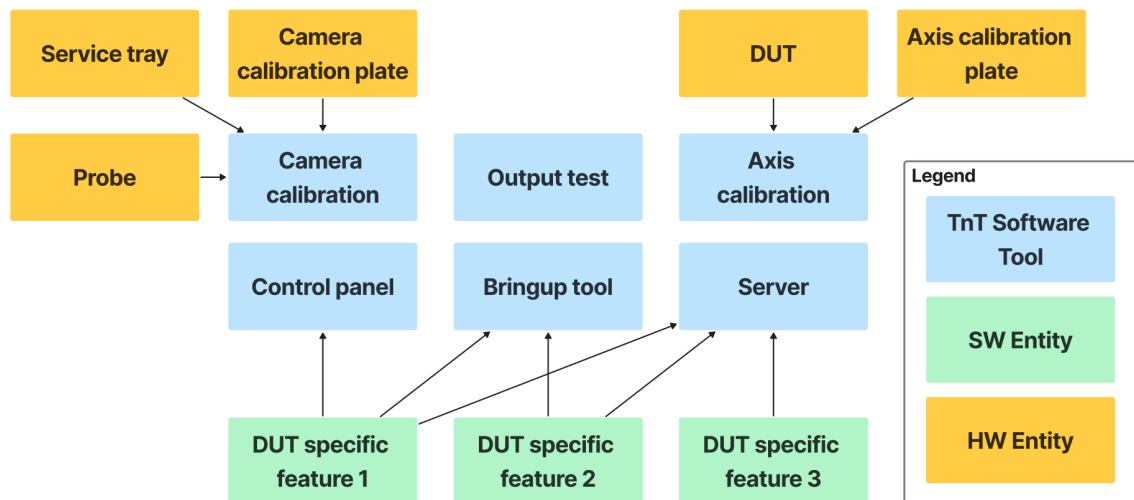
In figure 3.8 are the entities that were found. There are two types:

- Software entity: These are the system features. These are named as software entity, as they are something that can be thought to be toggled on or off with simple boolean parameters.
- Hardware entity: These are the hardware components that need to be configured. These are more complex in configuration point of view, there are many parameters that need value to be calculated.



**Figure 3.8.** Identified configuration items.

Figure 3.9 shows how these entities relate to TnT software tools. Blue rectangles in the middle are TnT software tools' configuration files. Hardware entities are marked with orange and software entities with green. Arrows are drawn from entity to TnT software tool's configuration file that contains parameters for the entity. For example, attributes of *Camera calibration plate* need to be defined only in camera calibration tool's configuration file. In case of *DUT specific feature 1*, three different configuration files need to be configured: control panel's, bring-up tool's and server's.



**Figure 3.9.** Configurable entity relations to TnT SW tools.

### 3.4.4 Result summary

One system version had 1000 lines of configuration. From there, 10% needs to be configured. Found configurable parameters were categorized into eight different configurable items. Configurable items were divided into two types, hardware and software entities. Hardware entities are always defined in single configuration file, but software entity can have dependencies on multiple files.

**Table 3.4.** *Identified configuration items.*

Configuration items		
Feature	Type	Defined in no. of files
DUT specific feature 1	SW Entity	3
DUT specific feature 2	SW Entity	2
Dut specific feature 3	SW Entity	1
Axis calibration plate	HW Entity	1
Calibration gauge	HW Entity	1
DUT	HW Entity	1
Probe	HW Entity	1
Service tray	HW Entity	1

## 4. DESIGNING CONFIGURATOR

In this chapter requirements for configurator is defined, then overall design done and finally software design proposal is presented.

### 4.1 Overview

Purpose of configurator application is to create new set of configuration files based on user's selections and input. Configuration items includes commonly configured features and components identified in section 3.4. Change to parameters outside identified configuration items is considered to be change to the system, requiring a change process. Target user is a developer familiar to the system. User needs knowledge of the system to use the application. Role of the application is to help user by doing the tedious part of configuration, but the user needs to be in control having knowledge what is to be done. In essence, the application will be a semi-automated software tool for a expert to use.

Main benefits of the semi-automated solution are

- parameter level knowledge is omitted from basic configuration work,
- parameter values such as coordinates are calculated from user input,
- configuration is validated automatically.

### 4.2 Requirements

Nature of the project was to find a good balance point between creating value and time usage. Requirements setting needed to take this into account. Since current target user is a developer and usage interval is around ten times a year, the application was chosen to be first developed as a console application. This allows quick deployment to usage. Overhead of changes to user interface can be avoided by creating it after the application is established and necessary changes are done. Most of the functionality is in backend development and the application is trivial in user interface perspective.

Main function of the application is to set features and configure associated parameters. Findings from section 3.4 were, that there are two types of configuration items: software and hardware entities. Software entities need simple boolean parameters, but require

these to be set in one or more configuration files. Hardware entities are always defined in a single configuration file, but require parameter values to be calculated. To ensure that user's mistake in input doesn't lead to misconfiguration, validation is needed to catch configuration errors. Business rules of the system defines the validation rules for parameters. Configuration state should be easily visible, so metadata about configuration needs to be written. The application should be developed with company's coding conventions in mind. Coding conventions define things like code style, formatter used, documentation, setting up logging and unit tests. These kind of requirements doesn't affect design at this level but are needed at implementation.

#### Main functional requirements

- FR01 The system shall enable software entities based on user's selection
- FR02 The system shall set hardware entities based on user's input
- FR03 The system shall calculate parameter values
- FR04 The system shall validate user's input when configuring an entity
- FR05 The system shall validate output configuration
- FR06 The system shall write metadata of created configuration

#### Main non-functional requirements

- NFR01 Application should be integrable to a GUI
- NFR02 Implementation should follow company's coding conventions
- NFR03 Codebase shall be maintainable
- NFR04 Validation rules should be maintainable

#### Constraints

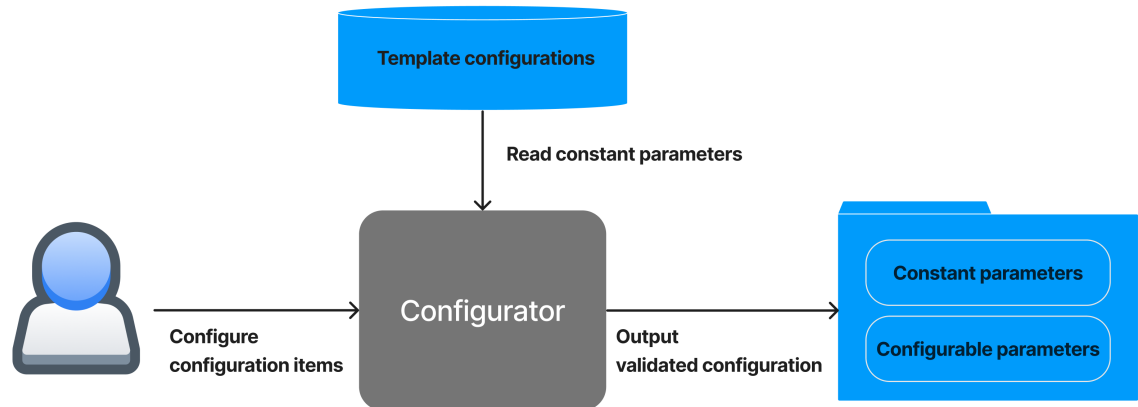
- Implemented in Python programming language

### 4.3 High-level design

It was found out in section 3.4 that only 10% of parameters inside of configuration need to be configured. Parameter information flow was split into two, constant and configurable parameters. It reduces complexity of data objects needed in the processing, because defining sections that don't have any configurable parameter, is avoided. Another benefit is clear separation between the two sets of parameters. Maintainer can view and edit validation rules separately which helps with non-functional requirements NFR03 and NFR04, defined in section 4.2, regarding maintainability.

Information flow through the application is shown in figure 4.1. Template configuration contains configuration files of each TnT software tool. Inside those files are the constant

parameters. Template configuration is read in from files. Configurable parameters are calculated based on user's input, which are then merged with the template configuration.

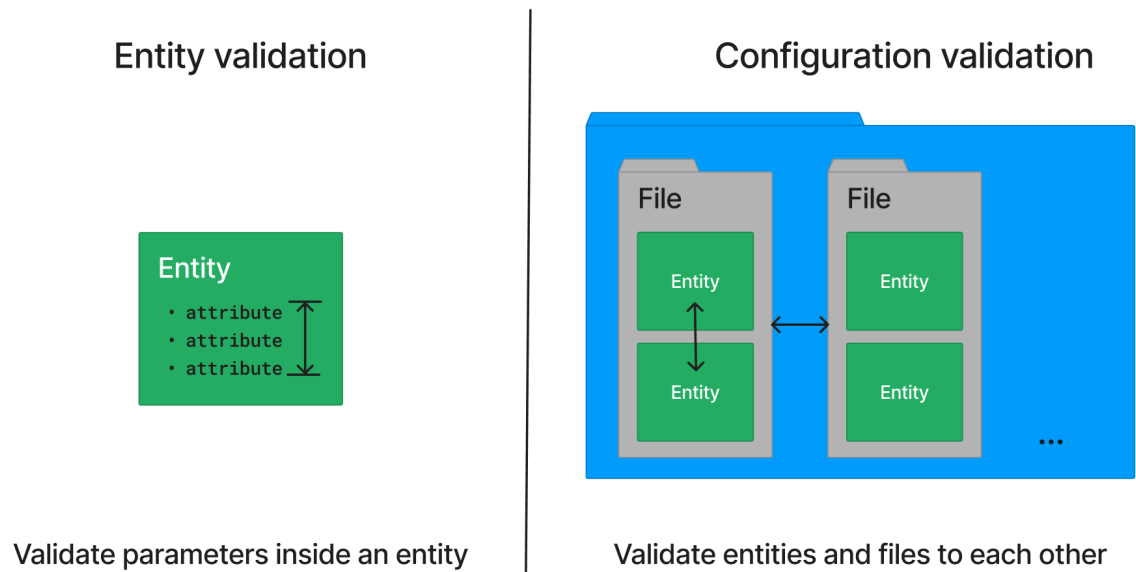


**Figure 4.1.** Configured parameters are merged with constant parameters to form a complete configuration.

Validation was done in two categories, entity validation and configuration validation, shown in figure 4.2. Validating attributes of an entity enables user to have validation feedback after a complete configuration item is configured, instead of showing all errors at once. It also separates validation rules of an entity and rest of the system, which helps with maintaining them. User input is not validated as is, instead it's used to calculate parameter values of a entity first and then the entity is validated in context of parameter values. This is to avoid translating parameter constraints to the context of primary values. It could induce a possible source for errors when defining the rules in primary value context. Giving errors in primary value context would enable to give user more readable error messages, but since target user has expertise to the system, it's not a must have.

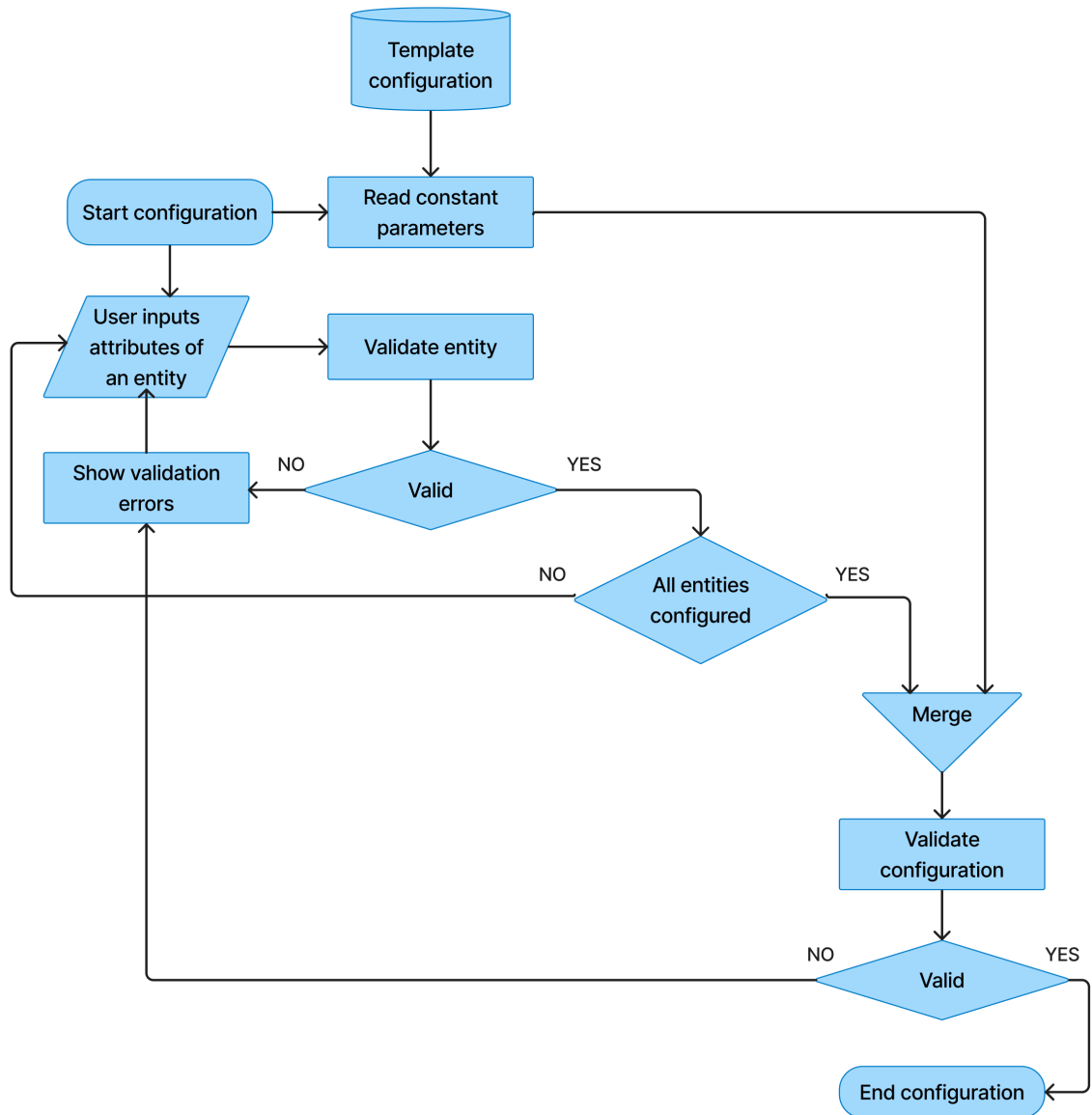
Unlike hardware entities, software entities don't need entity validation. They are features that user can control in only two states, enabled or disabled. Unlike hardware entities that have user defined attribute values, software entities' parameters are set by a setter function into a predefined state.

Configuration validation is validating a complete configuration. It's done after configurable parameters from validated entities are merged with constant parameters from template. System level validation rules are checked in this phase, because it allows to compare any parameters inside the configuration against each other.



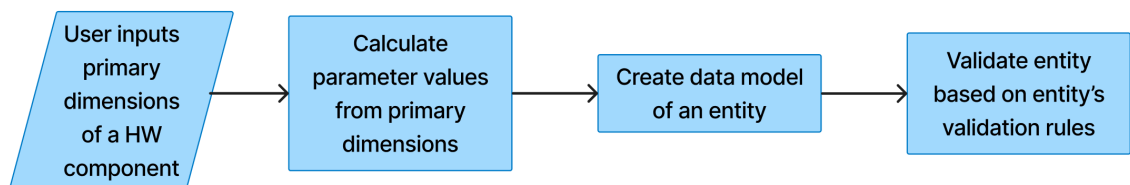
**Figure 4.2.** Validation was divided into two modes, entity and configuration validation.

Validation logic flow diagram is shown in figure 4.3. Constant parameters are read from template configuration file and configurable parameters from user's input. User's input is validated after each HW component is configured. After required input is given with valid values, configured parameters and constant parameters are merged to a complete configuration which is then validated.



**Figure 4.3.** Flow of validating and merging configurable and constant parameters.

User input is simplified in figure 4.3 and shown in more detail in figure 4.4.



**Figure 4.4.** Validation flow input detail.

Implementation of validation process was identified to be major part of the application that dictates rest of the code. It affects information flow, definition of data objects and validation rules. Developing whole validation from a scratch should be avoided if suitable existing solution can be applied. Implementing own solution would greatly increase development

effort of the application, increase risk of erroneous result and likely be less maintainable in validation rules point of view.

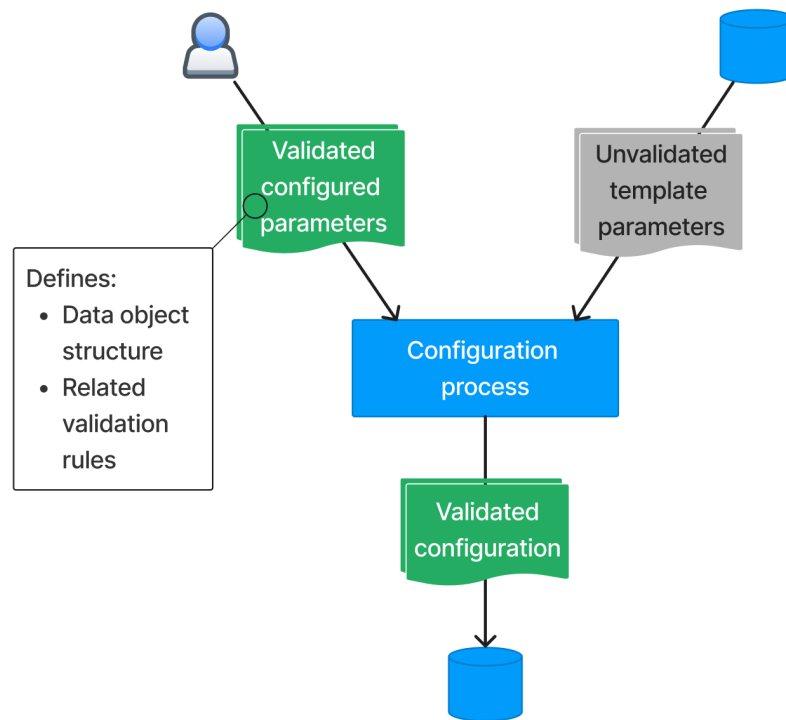
Pydantic library is a popular choice and has been on surface in many software engineering medias lately. Suitability of Pydantic was studied first, then few other options were checked how they would fit. Pydantic were chosen to be the overall best for this application. Main drivers for the choice were nested models and easy to read validation schema. It seemed to be the safest choice due to popularity and active maintenance, which indicates that there are valuable amount of usage examples and less bugs to be found during development.

Beneficial features of Pydantic library for this use case are (Pydantic Services Inc. 2023b)

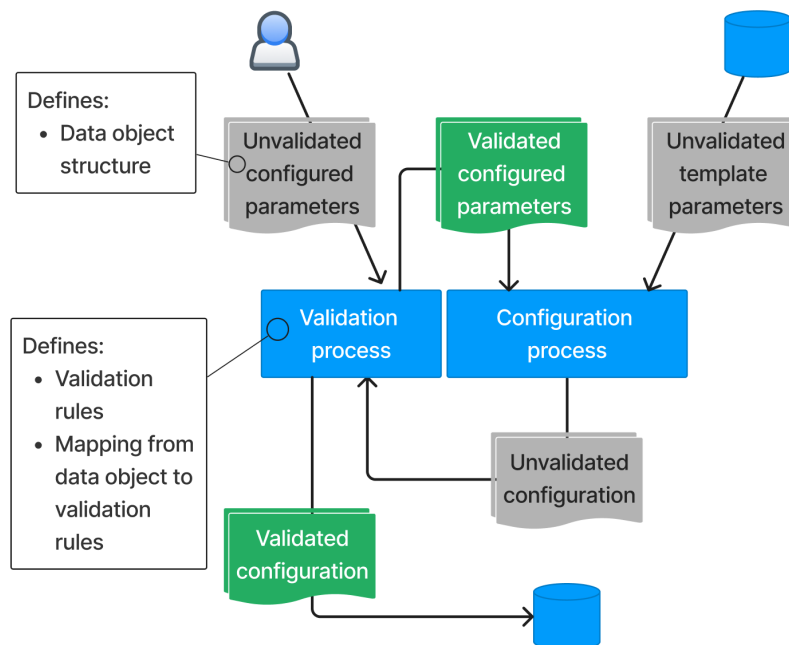
- Pydantic model is a data class: Model can act as a data structure to be passed through the program.
- Allows nested models: Validated data object can contain other validated data objects.
- Validation rules are defined in class declaration: Provides clear view of validation rules.
- Validation rules for a parameter: Enables validation rules for a parameter.
- Validation rules for a object: Enables validation rules to use every parameter to dependencies between parameters.
- Parameters can be read in to the model: Parameters not defined in the model can be appended. Parsing parameters into model is easy.
- Parameters can be exported from model: Parsing parameters from model is easy with existing functionality.
- Actively maintained.

Validation of Pydantic model is done at initialization of the class by default, based on validation rules defined inside the object, show in theory chapter 2.2.5. Using validated model has advantages of having definition of data class and it's validation rules in the same place. It also omits need to pass data through a separate validation step.

Figures 4.5 and 4.6 show a comparison to data flow between validated data class and validation module design approach. Using separate validation module 4.6 would need extra steps in the program flow going through the validation. Data object passing information trough the program and validation rules would be coupled, because relation of parameters and validation rules need to be defined. E.g. adding a new validated parameter requires changes to more than one file. Validated object method 4.5 makes data flow simpler and validation process easier to maintain. The validation rules are defined in class constructor of the validated object, like was shown in theory section 2.2.5.



**Figure 4.5.** Validation object.



**Figure 4.6.** Validation module.

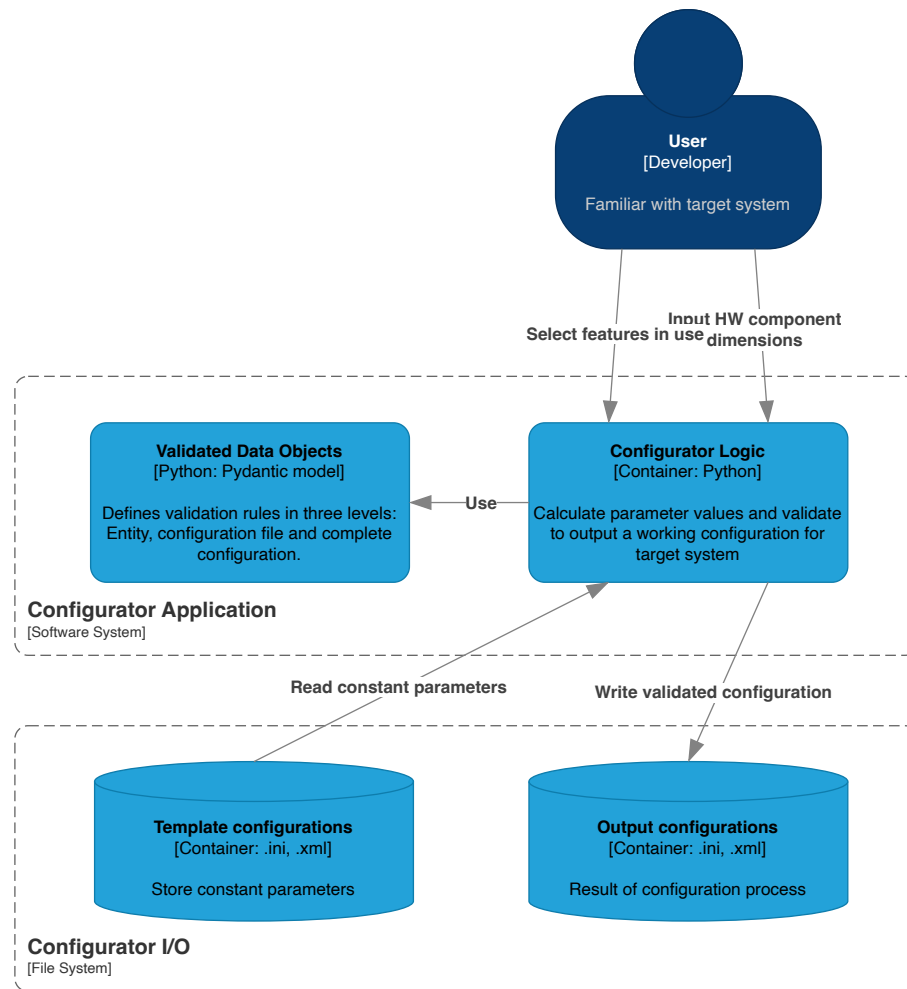
## 4.4 Software design

Architecture design is usually presented in context of bigger software projects. It was clear that this was a small software project, something that a single developer can do in a time scale of weeks. Architecture doesn't play such a big role in a small application compared

to more complex software systems, where clear separation between system components is critical for development, maintenance and even for system performance. Common factor is still separation of concerns and that can be done in the design of the small application. Advantages of building well structured code is a good way to accomplish non-functional requirements NFR01 and NFR03, show in section 4.2, about maintainability.

Design of configurator is shown using C4 model. The highest level of a C4 model, context diagram, showing interaction to users and systems outside has been presented in previous section 4.1.

Container diagram of configurator is shown in figure 4.7. Container diagram should show separate deployable units in the C4 model. Because this is a single application, the diagram is adapted into slightly more detailed to show the main components of configurator application. Division into deployable units would be dashed boxes in the diagram, *Configurator Application* and *Configurator I/O* using file system. Conceptual division to units was added to distinguish the main areas of configurator application: *Validated Data Objects* and *Configurator Logic*. Logic contains business logic of calculating parameter values from user input, enabling features by setting parameters, combining configured parameters with constant parameters and output a working configuration. Validated data objects are used to pass information between different components of the application. File system is used to store constant parameters and to save result configuration.



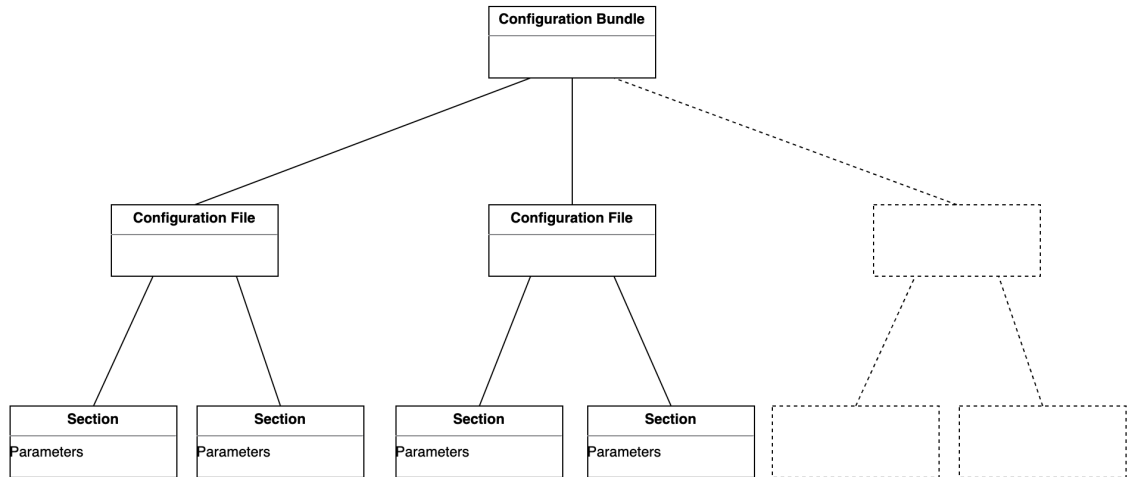
**Figure 4.7.** SW container diagram.

Component diagram 4.9 shows a more detailed view what previously mentioned containers consist from. Arrows show information flow between components. Application uses mediator design pattern to avoid tight coupling between different components. Components are coupled only to *Mediator* class with a concise interface to input and output data to the component. Business rules of configuration process is divided into *Input Processor* and *Configuration* classes.

Input and output data between components is passed in *Configuration Bundle* data structure. Shown in a abstract level in figure 4.8. *Configuration Bundle* is a representation of TnT software configuration. It consist of data objects representing configuration files of TnT software. Configuration file is divided into sections that contain parameters itself.

Information flow and between components and component functions are shown in figure 4.9. **Mediator** is the application's main method, so it can be integrated into graphical/command-line interface, or used as a script. It collects information and use other components to refine it into a result configuration. **Input Processor** has setter methods to configure hardware entities. Setter methods input dimensions of hardware components which are

used to calculate all parameters related to that component. Entity validation is done at this stage to make sure that information of a component is valid. Result of entity validation is passed to *Mediator*. After all entities are configured correctly, *Mediator* gets *Configuration Bundle* containing all configured parameters of hardware entities. **Parser** component contains input and output parsers. Input parser reads constant parameters from a template configuration into a *Configuration Bundle*. Output parser writes result *Configuration Bundle* into a output directory. Configuration files are using both XML and INI formats so it needs to support different parsing libraries. **Configuration** component gets configured hardware entities and constant parameters to merge them into a complete configuration. Setting of system features needs access to different configuration files, so it's done at this stage. Finally the resulting configuration is validated and passed to *Mediator* to output it via *Parser*.



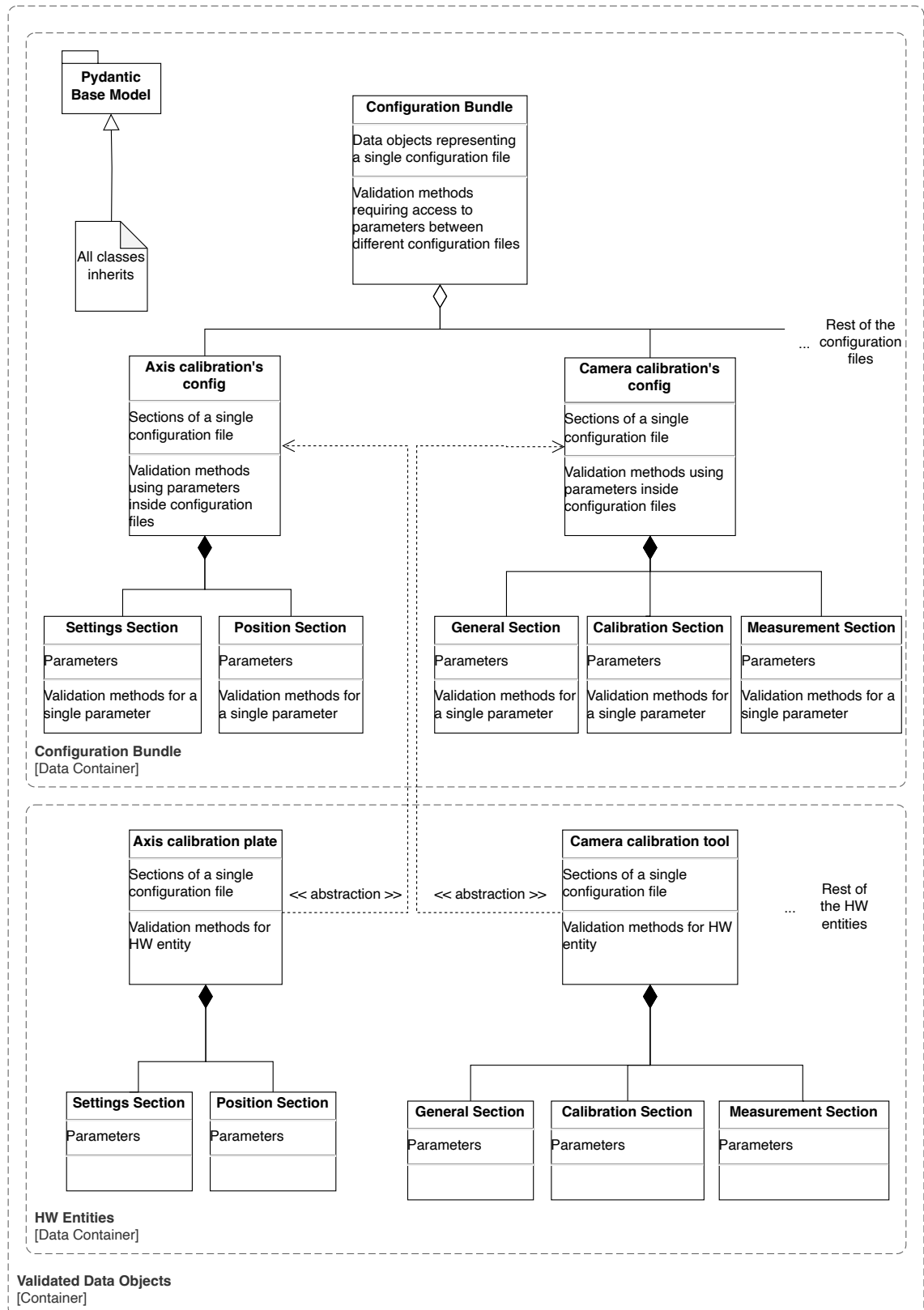
**Figure 4.8.** Config bundle overview.

Structure of *Validated Data Objects* container in component diagram 4.9 are shown in more detail in figure 4.10. Every class in the diagram is inheriting validation capability from a Pydantic library's *BaseModel*. **Configuration Bundle** is a container that can store every configuration file of a TnT system and represent a whole system configuration. It's an aggregation of configuration files. Configuration files are composition of sections within a configuration file. Sections are storing parameters. Separate sections are needed to

provide context for parameters that are having a same name. It was found in chapter 3.4 and shown in figure 3.9 that every hardware entity is defined in only one configuration file. Therefore ***HW Entity*** data objects share the same structure as a configuration file they belong to. Hardware entity objects contains only parameters of that entity and implement validation methods specific to them.

Validation can be done in every level. Section object is suitable for validating parameters values e.g. with type or value range constraints. Configuration file level is suitable for validation rules that needs to compare different parameters of a single file. Configuration level validation methods have access to every parameter in the configuration.

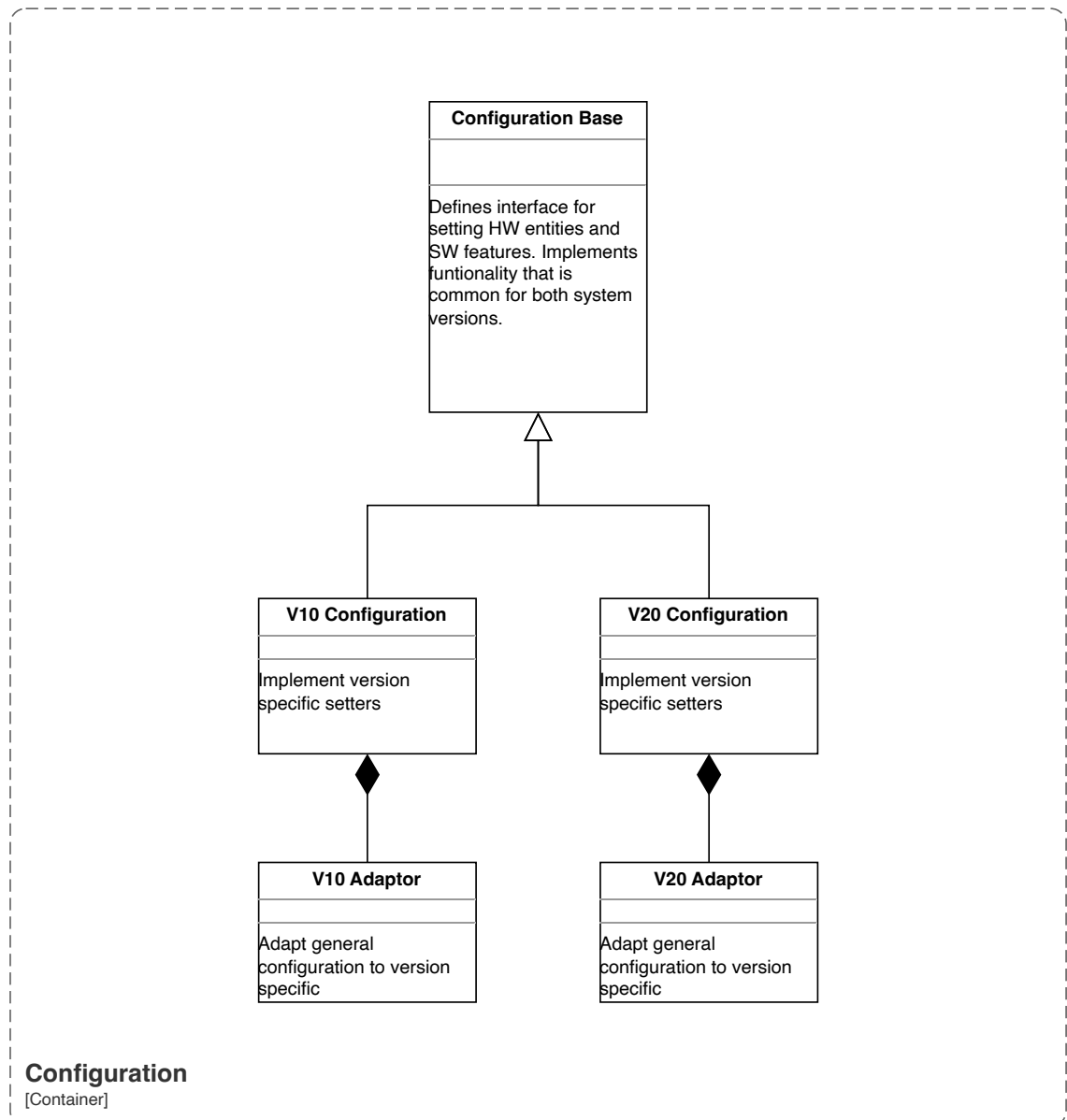
This structure combines domain model of TnT system configuration and applies that for hardware entities as well. Benefit is that instances of *HW Entity* data classes can be stored in *Configuration Bundle* container, which makes merging hardware entities to template configuration simple and main program can work only with *Configuration Bundle* type. Other benefit is that validation rules of hardware entities stays in separate places from the rest.



**Figure 4.10.** Data objects diagram.

Configurator needs to configure both supported versions of the system, V10 and V20. All version specific functionality in the code is defined in *Configuration* component. Outline

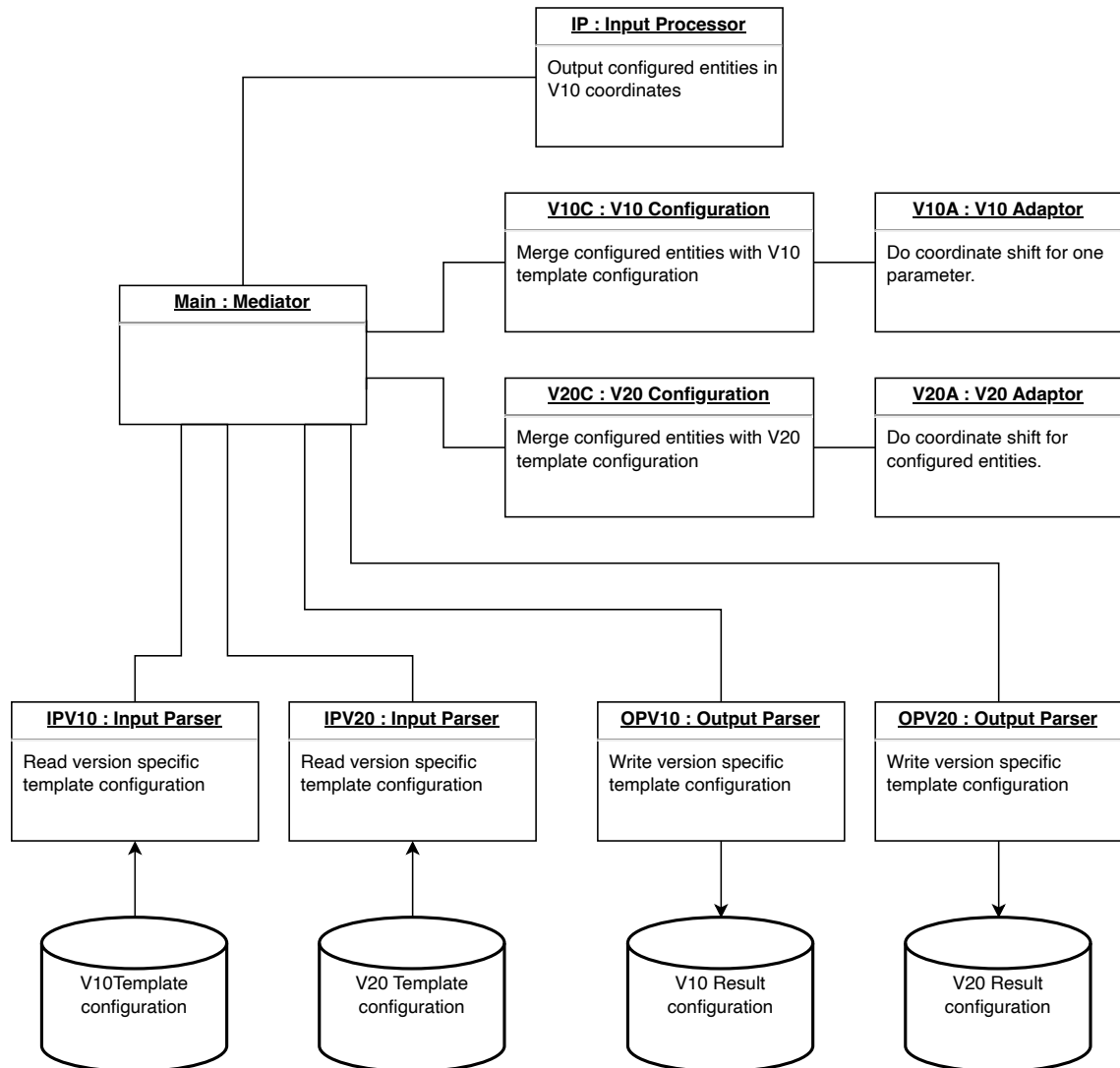
class diagram of it is shown in figure 4.11. Base class for *Configuration* defines interface for setting entities and getting result configuration. It also implements common functionality between versions. Version specific classes does their own implementations for setter methods. The application uses general configuration to configure hardware entities in *Input Processor*, which is then adapted to a version specific in the version specific class using designated adaptor class. New versions can be introduced by implementing new version specific *Configuration* and *Adaptor* classes.



**Figure 4.11.** Diagram of Configuration component.

Main differences between V10 and V20 configuration packages are coordinate shift in absolute coordinates and some of the parameters are different or have a different value. Previous figure 4.11 showed how version difference is handled in code level. Object diagram 4.12 shows how it's handled in application logic level. There are version specific

template configurations. Those go to version specific *Configuration* objects. *Input Processor* uses V10 coordinates to configure hardware entities. Class *V20 Configuration* needs to do coordinate shift to adapt those for V20. There was one special case that a parameter value is changed at run time in V10 system if certain feature is enabled. That is why class *V10 Configuration* needed it's own adaptor even though both, configured entities and template configuration are in same coordinate system.



**Figure 4.12.** Object diagram of configurator.

## 4.5 Example of validated data model usage

Listing 4.1 shows a definition for *Calibration* and *Measurement* sections for positioning tool's configuration. These are equal to the classes named in same manner in figure 4.10, under the *Camera calibration config*. A few parameters are defined with validation using Pydantic's *Constrained Types*. *confloat()* is used to attach parameters with greater than and less than comparisons, to restrict assigning parameter value outside of this

range. These data classes need to inherit Pydantic's *BaseModel* class.

```
class CalibrationSection(BaseModel):
    """General section in positioning tool's configuration file."""

    TrayOffsetX: confloat(ge=-200.0, le=150.0)
    TrayOffsetY: confloat(ge=-70.0, le=19.0)
    robotAbsoluteFirstPositionY: confloat(ge=80.0, le=250.0)
    robotAbsoluteFirstPositionX: confloat(ge=15.0, le=260.0)


class MeasurementSection(BaseModel):
    """General section in positioning tool's configuration file."""

    TrayOffsetX: confloat(ge=-200.0, le=150.0)
    TrayOffsetY: confloat(ge=-70.0, le=19.0)
    robotAbsoluteFirstPositionY: confloat(ge=80.0, le=250.0)
    robotAbsoluteFirstPositionX: confloat(ge=15.0, le=260.0)
```

**Listing 4.1.** Sections of positioning tool's configuration.

Listing 4.2 shows how the previous sections are used to form a validated data class to model positioning tool's configuration. It has the previously defined sections as instance variables and it also inherits *BaseModel* to access validation functionality. Validators to validate data between sections are defined here. Listing 4.2 gives one example of this using *root validator* decorator to access all class attributes. It gets the *TrayOffsetX* & *Y* values for both section and compares that they match. If the parameter values don't follow these rules during object creation, validation error is given. Data models can be validated also after creation.

```
class PositioningToolConfiguration(BaseModel):
    """Pydantic data model of positioning tool's configuration file"""

    Calibration: CalibrationSection
    Measurement: MeasurementSection

    @root_validator
    def equal_tray_offsets(cls, values):
        """Check that TrayOffsetX and -Y match within sections."""
        calib_tray_x = values[Sections.CALIBRATION].TrayOffsetX
        calib_tray_y = values[Sections.CALIBRATION].TrayOffsetY
        meas_tray_x = values[Sections.MEASUREMENT].TrayOffsetX
```

```
meas_tray_y = values[Sections.MEASUREMENT].TrayOffsetY

if calib_tray_x != meas_tray_x:
    raise ValueError("TrayOffsetX_doesn't_match_with_"
                     "Calibration_and_Measurement_section")
if calib_tray_y != meas_tray_y:
    raise ValueError("TrayOffsetY_doesn't_match_with_"
                     "Calibration_and_Measurement_section")

return values
```

**Listing 4.2.** Definition of validated data model with a validation.

## 5. DISCUSSION

To restate research questions from chapter 1:

- RQ1 What is needed to be configured in TOUCH MP test system?
- RQ2 What software design is suitable for TOUCH MP test system configurator?

### 5.1 Findings

Main findings to RQ1 were from configuration analysis in section 3.4. 10% of the parameters inside TnT system configuration needed to be configured. Configurable parameters could be categorised into eight different configurable items. Configuration items were found to be two types, hardware and software entities, needing a different process to configure. Hardware entities need dimension input from user, from where parameter values are calculated. All parameters of a hardware entity are defined in a single configuration file. Parameter values of software entities were simpler boolean values, but they are needed to set in one or more configuration file.

Results to answer RQ2 was presented in chapter 4. The design was influenced by the findings from configuration analysis. The ratio of configurable to non-configurable directed towards a design with two pipelines for parameter data. Configured parameters are calculated from user input, while constant parameters are read in from templates and merged together. A domain-specific data structure was created using Pydantic library, which includes a model for each configuration file of the TOUCH MP test system. The data structure serves as a container for passing data within the application, and as a place to define validation rules. The software design phase resulted in a high-level design for implementing configurator application. The application tasks were divided into different modules to separate responsibilities of the application. Formerly mentioned data structure is used to pass information between the modules.

Concrete result of software design phase was a high-level design how configurator could be implemented. Different tasks inside the application was divided into different modules. Input Processor handles user's inputs and calculates parameter values. Parser module reads in template configuration and writes output configuration. Configuration module merges configured and constant parameters and enables selected features. Previously

mentioned data structure is used to pass information between the modules. Separation into modules with defined responsibility helps implementation and maintenance of the software. E.g. if format of a configuration file is changed in the future, changes need to be only done to Parser module.

Validation was needed in stages, hardware entities need dimension input from user and user should have validation feedback after each hardware component is configured. Validation in other stages is done to make sure that parameters inside a complete configuration file and between different files are valid. This provides layered views to manage validation rules from a component to configuration file and complete configuration. Software entities don't need parameter information to be passed, only information if it's present. Software entity parameters could be set with setter functions after configurations are merged. Validation for software entities was not needed because there is no value input from user. Adding new configurable parameters in the future is fairly straightforward. If its parent configuration file and section are already in the system, it's a matter of adding it into a method that sets the value and defining validation rules. Even if the correct configuration file data class is missing, it's easy to extend *Configuration Bundle* data structure to have a new one.

## 5.2 Further development

With configurator it's estimated to take half a day, which mainly consist of collecting needed information like physical measurements. In addition to just improve creation time, if validation rules are set correctly, it will greatly improve quality of a configuration release. Previously there has been very limited ability to test configuration parameter values.

Future development for the application could include integrating it with a version control system where the finished configuration is uploaded, to avoid manual work. After configurator is implemented and biggest part of configuration process is automated, further development for the process would speed it up. Filling out information gaps and reducing communication time between stakeholders would be reduced by defining what information is needed before starting the work. Changes to the configuration files were not allowed in scope of this work, but that could be one form of development. In simpler way, refactoring the parameters with more suitable names. One option needing bigger changes could be a centralized configuration implementation to manage all software from a single source. Main benefit of centralized approach would be to implement a configuration service that would remove the need for having configuration files specific to the used test system version. Given that it's not a small change and configurator application is now on the way to improve current situation, it doesn't seem feasible to do. Codebase for this product has been stable and seems to continue to do so in the future, so configurator is a good solution for now.

### 5.3 Validity and generalizability

Validity of this work might be limited due to following factors. Only recently released configurations were used in the analysis. Some parameters needed to be configured might have been missed. On the other hand, the design of configurator was required to allow easy extension for new parameters and validation rules, so those should be trivial to add later. Validation rules for the parameters were not defined, so future work on the validation rules could expose some mode of validation that can't be executed. Nature of the configuration analysis was that it highly depends on domain expertise of the person doing it. In this work, mainly in the way that the person identifies correct category for parameters. I had experience of creating configurations for the system successfully, so I'm fairly confident that the result is a good fit. Still it can't be known until it's implemented and tested. Software design with its concepts was a fairly new field for me. At the start, literature had been misinterpreted or applied not following industry standards. It was encountered that some concepts were used in the design, just to later understand that it wasn't correctly understood and applied. On the other hand, some concepts still seem like they have much room for interpretation. Mistakes made have been corrected, and I think that it's a proof that I learned a lot. Example of this could be *Adaptor* classes in figure 4.11, that I first thought to be using Adapter design pattern, shown in theory chapter 2.2. But actually it's just manipulating data inside the data class, not adapting different interfaces.

This design was a fully custom approach to create configuration files for specific system. It can't be used to configure other systems, but the solution can be generalized in higher level to apply into different system.

## 6. CONCLUSION

The need was to find a solution for creating a new configuration for a test system. Purpose of this study was to answer what is needed to be configured, and what kind of software design is suitable for the configurator.

Main findings from configuration analysis was that 10% of the parameters inside TnT system configuration needed to be configured and those could be categorised into eight different configurable items. Ratio of configurable and non-configurable parameters directed towards a design with two pipelines of parameter data. Configured parameters are calculated from user input and constant parameters are read in from templates, then merged together. It avoids defining most of the parameters in the code which helps maintaining business rules of configuration process.

Domain specific data structure with validated data classes using Pydantic library were used to model information through the application. It's composed of a validated model for each configuration file. Models of hardware entities shared the same structure as the parent configuration file it belongs to, which provides easy way to merge them together. The data structure combined functionality of being

- explicit place to define validation rules,
- a container to pass data inside the application between different components.

The configurator should address the issues found in manual configuration work and set new standards for speed and quality of the configuration delivery pipeline of TOUCH test system.

## REFERENCES

- Alexander, C., Ishikawa, S. and Silverstein, M. (1977). *A Pattern Language: Towns, Buildings, Construction*. Center for Environmental Structure Berkeley, Calif.: Center for Environmental Structure series. OUP USA. ISBN: 9780195019193.
- Astera Software (2023). *What is Data Validation?* Accessed 2023-10-29. URL: <https://www.astera.com/knowledge-center/what-is-data-validation/>.
- Babich, W. A. (1986). *Software Configuration Management: Coordination for Team Productivity*. Addison-Wesley. ISBN: 9780201101614. URL: <https://books.google.fi/books?id=j6ZQAAAAMAAJ>.
- Bass, L., Clements, P., Kazman, R. and Safari, a. O. M. C. (2021). *Software Architecture in Practice, 4th Edition*. SEI series in software engineering. Addison-Wesley Professional. ISBN: 9780136885979.
- Bigelow, S. J. (2023). *Definition: configuration file*. Accessed 2023-10-28. URL: <https://www.techtarget.com/searchitoperations/definition/configuration-file>.
- Bosch, J. (2000). *Design and use of software architectures : adopting and evolving a product-line approach*. eng. Harlow: Addison-Wesley. ISBN: 0-201-67494-7.
- Brown, S. (n.d.). *The C4 model for visualising software architecture*. Accessed 2023-10-29. URL: <https://c4model.com/>.
- Buschmann, F. (1996). *Pattern-oriented software architecture : a system of patterns*. eng. Repr. Chichester: Wiley. ISBN: 0-471-95869-7.
- Crnkovic, I., Askund, U. and Dahlqvist, A. P. (2003). *Implementing and integrating product data management and software configuration management*. eng. Artech House computing library. Boston: Artech House. ISBN: 1-58053-498-8.
- Dorfman, M. and Thayer, R. (1990). *Standards, Guidelines, and Examples on System and Software Requirements Engineering*. IEEE Computer Society Press tutorial. IEEE Computer Society Press. ISBN: 9780818689222.
- Fisher, E. (2017). *Software and Firmware Parameterisation: A Basic Requirement for Modular Long-Term Usage*. Accessed 2023-10-29. URL: <https://www.software.ac.uk/blog/software-and-firmware-parameterisation-basic-requirement-modular-long-term-usage>.
- Freeman, P. and Wasserman, A. (1983). *Tutorial on Software Design Techniques*. IEEE Catalog number. EH0205-5, IEEE Computer society order number. 514. IEEE Computer Society Press. ISBN: 9780818645143.

- Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1994). *Design patterns : elements of reusable object-oriented software*. eng. Addison-Wesley professional computing series. Reading (MA): Addison-Wesley. ISBN: 0-201-63361-2.
- Goguen, J. A. (1984). Parameterized Programming. *IEEE Transactions on Software Engineering* SE-10.5, pp. 528–543. DOI: 10.1109/TSE.1984.5010277.
- Gomaa, H. (2011). *Software modeling and design : UML, use cases, patterns, and software architectures*. eng. Cambridge: Cambridge University Press. ISBN: 1-139-03597-5.
- Johner, C. (2019). *Parameterization of Software*. Accessed 2023-10-25. URL: <https://www.johner-institute.com/articles/software-iec-62304/and-more/parameterization/>.
- Joshi, B. (2016). *Beginning SOLID Principles and Design Patterns for ASP. NET Developers*. eng. Berkeley, CA: Apress L. P. ISBN: 1484218477.
- Kenlon, S. (2021). *What is a config file?* Accessed 2023-10-25. URL: <https://opensource.com/article/21/6/what-config-files>.
- Kerner, S. M. (2022). *Definition: Data Validation*. Accessed 2023-10-29. URL: <https://www.techtarget.com/searchdatamanagement/definition/data-validation>.
- Koskimies, K. (2000). *Oliokirja*. fin. Helsinki: Satku. ISBN: 951-762-720-3.
- Koskimies, K. and Mikkonen, T. (2005). *Ohjelmistoarkkitehtuurit*. fin. Helsinki: Talentum. ISBN: 952-14-0862-6.
- Leffingwell, D. and Widrig, D. (2003). *Managing software requirements : a use case approach*. eng. 2nd ed. The Addison-Wesley object technology series. Place of publication not identified: Addison Wesley Professional.
- Martin, R. and Martin, M. (2007). *Agile Principles, Patterns, and Practices in C#*. Robert C. Martin series. Prentice Hall. ISBN: 9780131857254.
- Martin, R. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. eng. 1st ed. Robert C. Martin Series. Hoboken: Pearson Education, Limited. ISBN: 0134494164.
- Miles, R. and Hamilton, K. (2006). *Learning UML 2.0*. eng. First edition. Beijing ; O'Reilly. ISBN: 1-306-81467-7.
- Miller, G. A. (1956). The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. *Psychological Review* 63.2. DOI: 10.1037/h0043158.
- Moreira, M. E. (2010). *Adapting configuration management for Agile teams balancing sustainability and speed*. eng. West Sussex: Wiley. ISBN: 0-470-97083-9.
- OptoFidelity Ltd (n.d.). *OptoFidelity TOUCH*. Accessed 2023-04-14. URL: <https://www.optofidelity.com/offering/products/touch>.
- Pfleeger, S. L. and Atlee, J. M. (2006). *Software engineering : theory and practice*. eng. 3rd ed. Upper Saddle River (NJ): Pearson Prentice Hall. ISBN: 0-13-198461-6.
- Piipari, T. (2013). Dynamic configuration management. MA thesis. Tampere. URL: <https://urn.fi/URN:NBN:fi:tti-201303211098>.

- Pressman, R. S. (2001). *Software engineering : a practitioner's approach*. eng. 5. ed. McGraw-Hill series in computer science. Boston: McGraw-Hill. ISBN: 0-07-365578-3.
- Pydantic Services Inc. (2023a). *Pydantic Documentation*. Accessed 2023-10-29. URL: <https://docs.pydantic.dev/>.
- (2023b). *Pydantic Documentation*. Accessed 2023-10-29. URL: <https://docs.pydantic.dev/latest/concepts/models/>.
- Qin, Z., Xing, J.-K. and Zheng, X. (2008). *Software Architecture*. eng. 1st ed. 2008. Advanced Topics in Science and Technology in China. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN: 978-3540743422.
- Refactoring.Guru (n.d.). *Mediator pattern class diagram*. Accessed 2023-10-09. URL: <https://refactoring.guru/design-patterns/mediator>.
- Rintala, L. (2021). Architecture design of a configuration management system. MA thesis. Tampere. URL: <https://urn.fi/URN:NBN:fi:tuni-202012118739>.
- Shalloway, A. and Trott, J. (2004). *Design Patterns Explained: A New Perspective on Object-Oriented Design*. eng. 2nd ed. The software patterns series. Hoboken: Pearson Education, Limited. ISBN: 9780321247148.
- SoftwareTestingHelp (2023). *Data Validation Tests For ETL And Data Migration Projects*. Accessed 2023-10-29. URL: <https://www.softwaretestinghelp.com/data-validation-tests/>.
- Sommerville, I. (2016). *Software Engineering*. Always learning. Pearson. ISBN: 9780133943030.
- Taylor, R. N., Medvidoviâc, N., Medvidovic, N. and Dashofy, E. M. (2010). *Software architecture : foundations, theory, and practice*. eng. 1st edition. Place of publication not identified: John Wiley. ISBN: 0-470-16774-2.
- University of Helsinki (n.d.). *Java Programming course*. Accessed 2023-10-06. URL: <https://java-programming.mooc.fi/part-7/1-programming-paradigms>.
- Villegas, F. (n.d.). *Thematic Analysis: What it is and How to Do It*. Accessed 2023-04-20. URL: <https://www.questionpro.com/blog/thematic-analysis/>.
- Wieggers, K. E. (2009). *Software requirements*. eng. 2nd ed. Pro-Best Practices. Sebastopol: Microsoft Press. ISBN: 0-7356-3518-8.