Improving the customizability of interfaces that communicate with different systems

Master's thesis in Software Engineering

Georgios Pseiridis Pseiras
Zhenyu Zhang

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2016

Improving the customizability of interfaces that communicate with different systems

GEORGIOS PSEIRIDIS PSEIRAS,
ZHENYU ZHANG

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2016

Improving the customizability of interfaces that communicate with different systems

Georgios Pseiridis Pseiras
Zhenyu Zhang

Supervisor: Patrizio Pelliccione, Department of Computer Science and Engineering
Examiner: Regina Hebig, Department of Computer Science and Engineering

Master's Thesis 2016
Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2016

Abstract

This thesis describes the process of handling the variability needs of a company which does not follow traditional product lines approaches and where the work of code maintenance is divided between the company and their customers. More specifically, this study focuses on improving the customizability of interfaces used for data transmission through different system, where misinterpretations can occur leading to problematic communication. The background specification, related work, trade-off analysis of available methods, the development of a prototype and its evaluation are presented in this study. We document the guidelines for decision making and suggest ways for future work. We pushed the prototype on the company's code repository, ready to be used by their employees and hopefully be proven useful. We hope that the findings of this study could support the customization endeavors of companies of similar technical contexts.

Keywords: Software customization, Variability Handling, Product Lines, Interoperability, Maintainability

Acknowledgements

We would like to express our sincere appreciation of Jeppesen for giving us the wonderful opportunity to work with them. More specifically, we would like to thank our supervisor at the company, Thekla Damaschke for her tremendous support on our study. She is the mind behind the original idea of this thesis topic and she was always willing to help.
We would also like to thank all our interviewees for making time from their busy schedule to engage in our study and answer our questions. Many thanks also to the line manager in the company, Cormac O'Brien for helping us get an office for our work and helping us with scheduling meetings, to Hans Andreasson for helping us identify key stakeholders, all the people from the IO team for helping us setting up our working environment and provide valuable feedback.
We would also like to express our gratitude to our supervisor in the university Patrizio Pelliccione for his support, encouragement and for showing us ways to continue with our work.

Georgios Pseiridis Pseiras, Zhenyu Zhang, Gothenburg, Sweden, August 2016

Contents

1
Introduction

Software variability is defined as the ability of a software system or artifact to be changed, extended or configured for use in a specific context. As the variability is improved, the system can be easier customized (Svahnberg, Gurp, Bosch 2001). Variability is a leading aspect of success in software engineering but at the same time a dominant reason for complexity augmentation (Rhein et al. 2015). To exploit the benefits of variability, requirements between different stakeholders need to be addressed and different software solutions should be proposed. There is not a single software package which could satisfy all customers. Customization is therefore often presented as a solution to the needs of the different customers (Weiss & Schweiggert 2013).

1.1 Purpose and objective

Handling variability is crucial for a company which places great emphasis on mass customization. There is a risk of problems occurring between different systems due to their inability of to interpret correctly the given parameters. Therefore, to mitigate this, effective variability management is required to enclose the activities needed to explicitly manage variability and handle the dependencies among the various software artefacts (Schmid & John 2003 p260).
In the concrete case of the Jeppesen Company, in each installation, their crew planning and tracking system is put into a unique system context that needs to exchange information with a varying set of other systems from different vendors. The timely and correct information exchange is crucial for the ability of the system to provide the expected services with expected quality. Especially for a tracking system, interoperability problems with surrounding systems can lead to severe impact on the airlines operation
Due to the high degree of variation which is required, interface development has so far been the domain of customization. These interfaces define the interpretation of the airline data. In the past, each airline had their own customer specific interfaces to communicate with the company's systems. The company is currently striving to introduce standard interfaces. In this way they are expecting to move away from customer specific interfaces and impose a standard behaviour which will be followed by all airlines. However, in the avionics field, it is a common phenomenon for different customers to use their own interpretation of otherwise well-defined standards and thusly violating the imposed standard behaviour.
Software product line engineering techniques cope with the high demand of varia-

tion among their products. Software product lines define a family of products which share commonalities and variations with each other and can be viewed as a family of products (Bosch Capilla & Kang 2013). Product line techniques suggest ways to minimize the costs of development, maintenance and evolution of the software products (Svahnberg, Gurp & Bosch, 2001 p2). The software companies are responsible for their products during all the life-cycles.
There is a significant area of research about handling software variability where a considerable amount of different methods have already been proposed. These methods suggest patterns, theoretical frameworks and tools, mainly in the context of software product lines. These methods work under the assumption that the maintenance of the software systems is being done completely by the company's side and that the required variation will live for very long.
However, this is not the case for Jeppesen, as part of their system's code is maintained by their customers. After the company delivers their systems, they lose control over customer specific source code. Their clients are responsible for partial code maintenance and compatibility of their systems. Therefore, the risk of incorrect interpretation of the provided parameters is high when one of the systems in the whole landscape is updated or replaced.
This study aims to support the customization efforts in a company which does not follow traditional product line engineering techniques. We asses and analyze the requirements, compare them with existing solutions and define whether they fit in our case. The outcome of this thesis are the guidelines to support future decision making for the customization needs of similar contexts based on the acquired knowledge, accompanied by solid reasoning.

1.2 Research questions

We have defined two research questions to determine a decision making strategy. These research questions are as follows:

RQ1: What are the variability needs and what are the limitations of the interfaces?

In order to suggest a method we needed first to understand the nature of the required variation of the interfaces. This research question's aim was to let us understand the context and provide a basis to design a method to address the variability needs. We therefore had to understand, why different airlines interpret well-defined standards in different ways, what causes faulty communication between different systems, which parts of the standard are still open for interpretation and how did the company so far managed the variability inside these interfaces. To answer these questions, interviewed experts inside the company about a specific standard interface. After the analysis phase was over, we tried to answer the second research question which was:

RQ2: What are the guidelines to support the customization efforts of a software system whose code maintenance is divided between the company and its customers?

This research question is concerned with providing the outcome of the thesis. Having obtained the context of the problem based on the first research question, we tried to suggest guidelines to effectively increase the customizability of their systems. These guidelines would be based on solid reasoning and evaluated to ensure the degree to which it is successful in facilitating the customization needs of the company under study and other companies with similar technical context.

1.3 Scope and limitations

This thesis focuses on the variability handling issues in the technical context of Jeppesen, located in Gothenburg, Sweden. Thus the conclusions of this study might not be applicable to other companies. Additionally, the main focus is placed on improving the customizability of the new system although we interviewed stakeholders working in the old system as well. However, the new system was supporting only one customer by the time this study took place. It is therefore hard for our method to be integrated right away since the possible emerging use cases are not well known right now.
The basis of the study was placed on a single, rather complex interface where several aspects are affected at the same time. It is concerned about timetable information and although it is standardized, airlines still interpret it differently with each other. Each field in the standard is translated into semantics which systems can be understood my Jeppesen's system. However, since some fields are open for interpretation, the processing of the semantics could lead to incorrect results.
Later during this study, we decided to expand the scope to another interface to investigate whether the suggested method for the first interface could be replicated to a similar context. This is the operational messages interface which can be considered as family of standards. Due to limited time, for this interface we took a more theoretical approach.

1.4 Method

The methodology followed the guidelines of design research as described in (Vaishnavi & Kuechler, 2004). We tried to first understand the problem and create knowledge by implementing our conceived idea. To gain an insight of the company's technical context we performed semi-structured interviews, a group interview and code analysis. To get the state-of-the-art of existing methods for variability handling, we performed literature review. The interviews were recorded, transcribed and analyzed further. The knowledge gained from the literature was documented and used to suggest a conceptualized solution.
We defined two research questions which served as the main guidelines of the research. In this way, we expected to come closer to the research goal by answering them. We later performed a trade-off analysis of the existing methods found in the generic literature. The purpose was to give solid argumentation of which method was more suitable in the problem under investigation. We later proceeded to develop a prototype and evaluate it.

1.5 Outline of the thesis

This thesis is structured as follows. In chapter 2 we present general information about the company and the systems they develop. In chapter 3 we present the key concepts of software customization and variability handling that this thesis is focused on and discuss the related work. In chapter 4 we present the methodology we followed to deal with the variability problem under investigation. In chapter 5 we discuss about the main variation points, the main indicators which influenced our decision making and a trade-off analysis of the main realization mechanisms. In chapter 6 we discuss the development process while in chapter 7 we present the evaluation of our approach and the guidelines for future decision making. In chapter 8 we state the threats of validity. The study concludes in chapter 9 and suggestions for future work is presented.

2
Background

Boeing was founded on July 15, 1916 by William Boeing and is the world's leading aerospace company (Boeing, 2016). This study took place in one of their subsidiary companies called Jeppesen founded in 1934. They offer a diverse set of products such as navigation charts, planning tools and crew management systems This study focuses on the last (Jeppesen 2016). In this section we discuss the background of the company, presenting an overview of their crew management systems, the interfaces we studied and the avocado model.

2.1 Crew management systems

An enormous amount of airlines expenses emanates from the costs of moving crew. Jeppense's office in Gothenburg specialises in providing crew management solutions to airlines so as to increase the crew productivity. The planning process is very complex and consists of a number of different stages.
The first one is the manpower process which is about the long term plans. Here, the general number of crew members, along with the required qualifications is defined. This can be considered as defining anonymous work-blocks for crew personnel, where they should be filled by real people later on. The companies can decide who should be hired, promoted, who needs training and how the leave should be divided.
The next process is called pairing. It is a set of flight legs, which start and end in the same destination. This is a sophisticated process where a large number of factors should be taken into consideration. When the crew comes back, the pairing is finished. There are a lot of restrictions such as the hours the crew can work, how many continuous days they are allowed to work, the different workforce laws of the country they come from to name a few.
The next process is called rostering and it is about assigning actual personnel. The rosters can be viewed as the actual work schedules. Again, here there are lot of restrictions. For example, a trip going to Brazil requires at least one crew personnel who can speak Portuguese. What is more, a crew member might wish to take a few days off and therefore another crew member should replace him or her for. Sometimes crew can not work together or they should work together.
Finally, there is the crew tracking stage. This is where the company tries to solve problems when they arise. For example, a crew member who got sick in the middle of his or her assigned trip has to be replaced by another member who is on standby in this city.
Valid and consistent information exchange between different systems is crucial for

the optimization system to provide the best results. However different airlines have different customization needs. Consequently, their systems vary with each other. This imposes a threat for the information interoperability as there is a risk for misinterpretations of the exchanged data.

2.2 Standard interfaces

International Air Transport Association (IATA) provides support for most airline companies throughout the world. The IATA manual defines the structure and behaviour of both interfaces of the study. We used the issue of March 2011 for the study.

2.2.1 SSIM interface

The Standard Schedules Information Manual (SSIM) defines the interface which is complied to the IATA regulations. It is used from airline companies to produce schedule data in the form of timetable information. The airline companies generate an SSIM file which is then distributed to multiple recipients such as airline reservation systems, timetable agencies, air traffic control authorities and so on (International Air Transport Association, 2011). Jeppesen systems use these files for crew and fleet management, with a strong emphasis on optimization of crew pairing, rostering and tracking. The need for the consistency of the data is therefore considered highly important in order to create valid and efficient solutions.
The interface under investigation has been selected because it is well-known by the people in the company and it is well-defined by the IATA manual. However it contains a number of variability points. This means, there are parts of the standard which are open for interpretation that the interface should be aware of. Therefore, it was considered a good candidate to provide a basis for this research, where the outcome could be replicated to other similar interfaces as well.
The manual provides explicit formatting rules to define the interface's syntax. Each byte in the syntax has its own meaning and corresponds to one of the supported fields. It should be noted that the company's systems do not read all the fields, since some of the optional fields such as meal service, have no impact on crew planning and are therefore discarded.

2.2.2 Operational messages interface

The second interface is about the operational messages, which could be considered as a family of formats. They are concerned with information regarding amendments of flight schedules, deviations from original schedules and aircraft movements. The information are being exchanged between airlines, aircraft and schedule aggregators (International Air Transport Association, 2011).
An identifier is placed at the beginning of the message to differentiate between each format. Information for permanent changes to basic schedules are transmitted through the Standard Schedules Message (SSM). Modifications of schedules are defined through the Ad-Hoc Schedules Message (ASM). Aircraft Movement Messages

(MVT) are concerned with messages regarding departure, arrival and delay of a flight which were produced manually. Aircraft Initiated Movement Message (MVA) are concerned with messages of the same nature as with MVT but produced by the aircraft itself (International Air Transport Association, 2011).
The manual provides guidelines for specifying a diverse set of actions for each of these formats, in the form of sub-messages. These actions, for example, can define behaviour for inserting a new flight designator, update information of existing flights and change of existing routing information. Each of these sub-messages are specified in a similar fashion as with the SSIM's case; each byte corresponds to a field which contain data for messages between the aircraft and the airport. Not all the messages are being used by each airline.

2.3 The avocado model

The Avocado model demonstrates how the company develops its products. This Avocado Model is shown on the figure below:

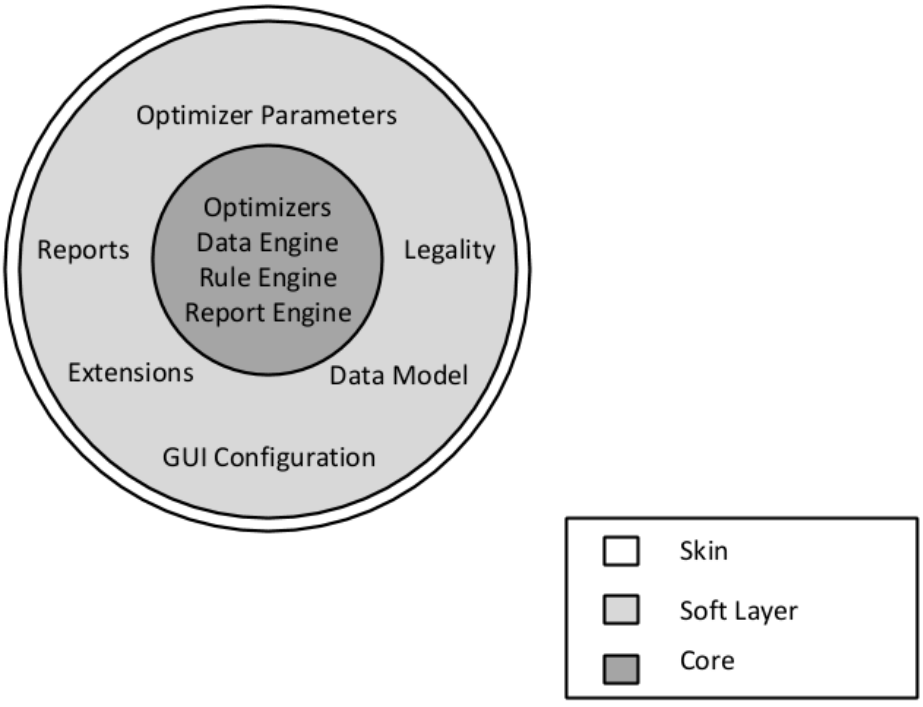

Figure 2.1: The Avocado model.

The core part consists of functionality that is the same to all of their customers. For example, all the optimization algorithms are embedded in the core. The translation of SSIM information to semantics is also inside the system's core.
Then, there is the soft layer that surrounds the core. This layer contains the business logic and it can be different from customer to customer. For example, union

contracts are different between the United States and Europe. Another example is how the graphical user interface is displayed from customer to customer. Even the translation of the SSIM can be modified to adhere to an airline's needs. This layer can be modified by the airlines companies alone, or with the help of Jeppesen engineers. The configuration can be done by utilizing a domain specific language developed by the company as well as by writing Python scripts which do not require the system to be recompiled. The soft layer therefore varies for each airline. Jeppesen does not have control over this layer after their system is delivered. It is up to the airlines to maintain it.
Finally, there is the skin, which is the outer layer of the avocado. This is where the system allows its users to define parameters. For example, how many days off in a month can a crew member have.

3

Theory and related work

In this chapter, we discuss foundations, key concepts behind the study as well as related work. There is a significant area of research to suggest ways and tools to manage variability. Most literature refers to product line engineering, which authors differentiate it from pure software engineering techniques. Typically, in software engineering approaches, the focus is placed on delivering single products. Product lines aim to deliver a family of products (Völter, 2009).

3.1 Challenges of Software Customization

Mass customisation is concerned with the production of goods that address the particular requirements for a number of different customers (Davis 1987). Dealing with software customisation requires significant investment and variability management from the software companies. Additionally, risk management is required as there is a number of potential problems that can arise. The systems need to be kept compatible with each other. Every time a system is updated, removed or changed, there is a risk that something might break. Consequently, maintaining low customization costs and the ability to assess risks of potential mismatches between software products are central challenges of mass customization.

3.2 Related Work

To the best of our knowledge, we identified a gap in the literature of case studies in similar contexts. That is, all existing literature we came across talks about managing variability from the company's side, not sharing the maintenance work among the company and their customers. There is a significant amount of case studies related to variability handling, management and customization efforts.
The authors in (Svahnberg, Gurp & Bosch, 2005) performed and analyzed a number of case studies in large IT companies for ways of managing variability. They proceeded to present guidelines for decision making based on explicitly defined indicators. They suggest thirteen different realization mechanisms as shown in figure 5.5 in the trade-off analysis subsection. However, the guidelines were meant to be followed by companies which utilize traditional product line approaches, where the maintenance is done by the company's side. Although our study was greatly influenced by their work, we could not be sure whether they could be applicable in our context too. For this reason we decided to expand their guidelines to a different context as well, following their indicators.

The authors in (Capila Bosch) present a systematic review of foundations and principles for effectively handling variability. Similarly, (Galster et al. 2014) present a systematic literature review for variability in Software Systems.
In (Renault 2014) the author shares his industrial experience of ways to explore reuse and variability management mechanisms with system engineering methods. He discusses the drivers for variability management, the drivers for reuse, the main existing practices and some gaps of existing approaches.
The authors (Galster & Avgeriou 2015) discuss their findings of case studies about variability handling in large scale systems. Again, here the authors suggest ways to handle variability. Their work does not focus on product lines but rather in large scale enterprise systems. They also compared their methods with the mechanisms discussed in (Svahnberg, Gurp & Bosch, 2005). They provide tables for mapping of variability types to levels of variability and to compare their suggested variability realization techniques against variability handling mechanisms.
In (Jansen et al. 2010) they discuss how can customization mechanisms in software product lines be applied in the context of multi-tenant web applications. They perform two case studies and discuss their findings. The variability realization techniques aim to solve functionality in this concrete context but they are inspired by the previously mentioned techniques in the taxonomy of (Svahnberg, Gurp & Bosch, 2005).

3.3 Software Product Lines

The ability of customers to demand products tailored to their requirements, shapes the software industry. To cope with this demand, software companies have introduced product lines (Bosch Capilla Kang 2013). Software product lines can be viewed as a family of products where commonalities and variations are shared between the products. The aim is to keep the development, maintenance and evolution costs to a minimum by exploiting asset reuse mechanisms (Svahnberg, Gurp & Bosch, 2001 p2).
The demand of individual products is steadily on the rise, while there is a continuous pressure to deliver these products as fast as possible so as to obtain a competitive advantage on the market. Product lines offer methods and mechanisms to exploit the commonalities and varying parts of each individual system (Svahnberg, Gurp & Bosch, 2005).
Companies define a domain software architecture that enables the development of a number of software solutions that can be configured to match the market needs. The individual products are built by borrowing a number assets from this domain architecture. When the development starts, there is an arbitrary number of possible systems that can be built. Later on, however, this number is constrained to only one run-time system by taking some design decisions. These design decisions are defined as variability points of the system (Svahnberg, Gurp, Bosch 2001).
A large number of papers discuss techniques that can be applied in the context of software product line engineering. According to (Svahnberg, Gurp, Bosch 2001) most design decisions are delayed until the later stages of the development life cycle. This implies that the product can choose which variants to include for each

variability point as a set of the available components which are selected in build time. Another way is to allow the users to change the system according their wishes during run time by choosing the available functionality provided by the system. Modelling the variability is considered essential to support the construction of an adjustable architecture (Griss, 2000). This is also supported by (Pohl, Böckle, Linden. 2005 p60) where they discuss that the first step towards an effective way to handle these points is by understanding them. They also suggest to use feature models which are used to represent the common and different features of a software product line. During the different development cycles, there are different types of modeling as they describe different abstraction levels.

3.4 Key concepts

This section clarifies the key concepts that are being used in the rest of this studies chapters. It should be noted that there is not necessarily a single definition for each of these concepts.

3.4.1 Variation Point

Variation point, also referred as variability point, expresses where differences between products exist (Linden, Schmid & Rommes 2007, p10). A Variability point implies that a particular artefact will be different among the family of products (Pohl et al. 2005, p61). An example of a variability point is the video capturing quality of a modern mobile phone. This can be different based on the phone's hardware or based on the the choices the user makes during the phone's operation.

3.4.2 Variant

A variant defines the available options for a particular variation point (Linden, Schmid & Rommes 2007, p11). An example of a set of variants is the available options for selecting the quality of the video recording functionality. These options can be can be for example low, medium, high and ultra-high quality. The variants, when selected, are bounded to the particular variation point of the video recording quality.

3.4.3 Run time and build time binding

An important dimension of variability management is the time of binding. It can either take place during build time or run time. Build time means the variability is bounded during the system's development, maintenance and evolution. During run time means the variability is bounded during the operation of the system (Svahnberg, Gurp, Bosch 2001 p5-6). An example of build time variability could be a family of products, such as surveillance cameras. One of these could be a fixed camera which includes 720p recording while another camera includes 1080p recording and 360 degrees view and zoom capabilities but is more expensive. The buyer then decides which camera is more suitable for his or her needs accordingly. Therefore, the buyers

can not change the variants of the system.
An example of run time variability binding could be the language selection of the user interface of the surveillance camera's software. The available languages are the variants and are all included in the system. The users then decides the language during the system's operation and can change it later if they wish. The user interface of the system is then changed dynamically based on the current user's requirements. Run time and build time variability play an important role on our decision making as we will discuss in the following sections.

3.4.4 Variability Types

We use the definitions presented in (Anastasopoulos & Gacek. 2001). The variability types can be:

- Positive: A functionality is added in the product
- Negative: A functionality is excluded from the product
- Optional: A certain code fragment or module can be added.
- Alternative: A certain code fragment or module can be replaced.
- Function: A functionality is altered.

3.4.5 State of variability

Following the taxonomy of (Svahnberg, Gurp, Bosch 2005, p710), the state of variability influences the decision making strategy. These indicators are also defined in (Svahnberg, Gurp, Bosch 2001, p6). The state of variability can either be

- Implicit: The variability point is part of a higher abstraction level but not part of the system.
- Designed: The variability point is considered explicit and design decisions are taken to suggest how it should be implemented into the system.
- Bound: The variability point is bound to a particular variant.

3.4.6 Software Features

The products vary with each other in term of the provided features. According to (Bosch et al. 2013 p.26- 29) product lines need to define the common and varying parts between each of their products. These are commonly referred as features. This is also supported by (Svahnberg, Gurp, Bosch 2001) where they refer to features as an abstract way to describe the user requirements. The authors suggest that the features can be grouped as follows:

- Mandatory, which means the feature should always be part of the system.
- Optional, where the feature may or may not be included.
- Alternative, which implies exactly one feature can be selected.
- External, which means the features are not part of the current system. This means that the system allows the use of functionality which is part of another system.

3.4.7 Interfaces

Interface is defined as a common boundary between different systems or between parts of a system, through which data are transmitted (IEEE 100 2000, p. 574). In the company's context, interfaces are being used to define airline related operations and properties. Information for crew, trips, rosters and aircraft are being transmitted through them. Most of these interfaces have been tailored to adhere to the needs of each individual airline company. For certain behaviours, each airline used their own interface which could not be reused by another customer. This resulted to a significant amount of customer specific interfaces. Furthermore, their customers were responsible for the maintenance of their interfaces and make sure they are compatible with the other systems and as a result the company was receiving a lot of support questions.
The company is now trying to move away from customer specific interfaces by introducing standardized ones with standard behaviour. The SSIM is one of these standard interfaces. Its well-defined format is expected to be followed in the same way from all of the airline companies. However, in practice, even standard interfaces are being misused in some way as there are still parts which are being interpreted in a different way by each airline.

3.5 Software Quality

Software quality is a broad topic in software engineering. Quality requirements specify how well the software product is expected to perform its functions. They are also known as non-functional requirements to distinguish with the actual functional requirements of the system (Soren 2002, p217). We follow the definitions of the (ISO/IEC 9126-1) standard which categorizes the different dimensions of software quality in six clauses. Maintainability and interoperability are the core quality requirements of our study. The reason is that the company wishes to implement a solution which is easy to maintain, without breaking the communication of the interfaces with different systems. Changeability is a categorized as a sub-quality of the maintainability clause. We analyze these qualities below.

3.5.1 Maintainability

According to (ISO/IEC 9126-1) maintainability is about the capability of the software system to be changed after it has been delivered. The sub categories of maintainability refer to the ease that the product can be analyzed, modified, tested and comply to maintainability standards. (Sommervile, p243-244) identifies three main categories of software maintenance. The first is about fixing bugs, the second is adaptation to a new enviroment and finally the inclusion of additional functionality. The additional functionality is usually the source of the highest maintenance efforts. Additionally, the cost of maintenance is considered as the most costly phase of software development as discussed in (Asadi, Rashidi, 2016) and (McKee 1984). The company stresses the importance that the solution to their customization issues needs to be easy to maintain. As an example which was raised during our inter-

views, the component which translates raw input data into XML might slightly be different from customer to customer. Should this be the case, the company might end up having thirty or more instances of the same component to maintain resulting to maintenance hell.

3.5.2 Interoperability

According to (ISO/IEC 9126-1) interoperability is defined as the capability of two or more software systems to communicate and exchange information correctly in order to achieve their goal. Different systems have different behavior patterns. A message sent from a particular system could be interpreted differently from another one, resulting to problematic communication. In some cases, the reason for this is the complexity of the required parameters for the effective communication (Rowely 1995). As an example, in (Garlan et al. 1995) found that integrating commercial off-the-shelf products (COTS) proved to be hard because each product had different behavioral model. The effort to make the systems interoperate proved to be futile and very costly. Another example of industrial interoperability problem in the aviation industry, would be the interpretation of the date field field. A system sends the date in local time to another system which in turn interprets it as UTC. This results in problematic communication, even if the semantics comply to the standards of the interface used.

3.6 Dimensions of variability handling

The authors in the systematic literature review by (Galster et al. 2014) identify two major dimension clusters of the variability handling. The first one is concerned with portraying and modeling the variability, the different requirement types and which software artifacts are affected. This dimension is intertwined with requirements engineering, where the specifications of the variants are introduced. Representing the functional requirements using feature models appears to be the most dominant approach.
The second dimension is related to the methods of realizing the variants inside the system. These methods are grouped on a higher level as follows:

- *Reorganization:* The structure of the system, along with its artifacts is reorganized to adapt to changes.
- *Selection:* Selecting among a given set of variants for each variation point. This appears to be a typical approach in the context of software product line engineering.
- *Value assignment*: A particular value is assigned to a variation point, usually by parsing parameters.
- *Code generation*: This refers to techniques used to generate customer specific artifacts based on a given input.

According to the authors, the most popular realization technique is the selection

from a pool of variants followed by the system's reorganization (Galster et al.). It should also be noted that the authors distinguish between software engineering and software product line engineering techniques. The variability handling strategies in the context of software product lines share similarities, something that does not exist in pure software engineering approaches (Galster et al. 2014 p297). In the following subsections of our report we will discuss about the available technical methods in more detail.

3.7 Methods for variability handling

In this section we discuss the identified methods for variability handling and management. Because of the large amount of different available methods, we will focus on the most popular ones that appear more often in the literature.

3.7.1 Generators

(Wölfl et al. 2015) (Bass, Clements & Kazman 2012 p402) (Pohl, Böckle & Linden 2005, p251) have discussed the concept of Generators as a widely used mechanism in software product lines to produce product specific architecture. According to a systematic mapping in (Mehmood & Jawawi, 2013) code generators rely on model-driven engineering and aspect-oriented techniques. Model-Driven Engineering uses models as primary artifacts which generate automatically the required components a software system needs. Aspect-oriented software development focuses on the separation of cross-cutting concerns.
(Jörges 2013, p11-35) presents the state of the art in code generation. Most generators use meta-models and domain-specific languages. Generative programming is a concept that implements a family of systems which is generated by a given requirements specification. Most of the available techniques make use of domain-specific models, developing domain specific and defining languages and transformation rules.
(Wölfl et al. 2015) present an experience report with making use of a novel code generation approach making use of aspect-oriented development. They combine model-based and product-line technology to create safety critical software in the technical context of Airbus Helicopters.

3.7.2 Configuration Management

(Bass, Clements & Kazman 2012 p402), (Svahnberg, Gurp, Bosch 2001) and (Svahnberg, Gurp, Bosch 2005) suggest configuration management tools as another powerful mechanism to manage variability.
Based on the definition in (Bachmann & Bass 2001 p7), this approach aims to develop customer specific products by invoking the required modules. The variants of each variation point are used as input and configuration items are produced to shape the product accordingly.
In (SEI 2000) the authors suggest a Capability Maturity Model of enhancing the effectiveness of these tools. Configuration management is concerned with the integrity of the different software products, making use of configuration control mechanisms

(SEI 2000, p72). Configurations form a baseline which defines the required grouping of various products (SEI 2000, p186).

3.7.3 Self-Adaptive systems

In (Galster et al. 2014) they discuss the concept of self-adaptive systems, which heavily rely on managing variability. Since our study focuses on the SSIM adapter, we investigated related work for self-adaptive adapters and connectors. The work of (Di Marco, Inverardi & Spalazzese 2013), (Ke & Huang 2012), (Van den Heuvel, Weigand, Hiel. 2007) addresses the issue of solving protocol mismatches between clients and servers. The main idea is the use of an adapter or connector which contains a strategy to select among a pool of algorithms the optimum one for each mismatch case.
The area of configurable adapters is currently not widely explored and the proposed solutions put a lot of emphasis on satisfying performance requirements. A controller retains a knowledge-base, in the form of a repository of scripts and based on the current state of the system it changes its behaviour dynamically.

3.7.4 Code level techniques

There are code level techniques for handling variability in both traditional software engineering and software product line engineering. These techniques range from either applying design patterns, or simply by putting conditions on constants or variables. We reviewed related work in both fields. It is noteworthy that hybrid approaches can exist, as the patterns in both fields can be used together.

3.7.4.1 Design Patterns

To increase the software's customizability, the software architecture needs to be able to accommodate the variation points. For this reason, we consider generic design patterns that should facilitate this purpose. We reviewed the design patterns originally presented by the Gank of Four (Gamma et al. 1995). These patterns are also explained by (Freeman, Sierra & Bates 2004) in an easier to understand way.
Design patterns are reusable solutions to recurring problems that arise in specific design contexts (Gamma et al. 1995). Their aim is to solve these problems in an elegant and effective way but most importantly in a way that is predictable since the risks and benefits of each design pattern are known beforehand.
The authors in (Mirakhorli, Mäder & Cleland-Huang. 2012) discuss the applicability of the design patterns and architectural tactics in different contexts. The aim is to enable the software architecture to accommodate variability points efficiently. Additionally, they provide figures and illustrative examples to enhance the understanding of the readers and aid their decision making.

3.7.4.2 Software product lines patterns

We reviewed a summary of product lines patterns as presented in (Lee, Hwang 2014). We also reviewed the patterns which this paper references, such as those presented in

(Anastasopoulos & Gacek. 2001). The taxonomy in (Svahnberg, Gurp, Bosch 2005) provides some code level techniques and their expected results for each of these. These papers provided an important insight of the current methods for managing variability to highly customized software systems by exploiting reuse mechanisms. (Völter, 2009) identifies three general patterns in software product lines. These patterns are removal, injection and parametrization. Removal instantiates a product without the inclusion of specific components. Injection includes adds up more to the product's core. Parametrization suggest ways where variants can be specified for certain components by providing parameters.
The methods are also affected by the different lifecycle phases of the product lines and the binding time of the variability. Lifecycles can be for example during requirements engineering or architecture design, while the binding time can be during compile-time, link-time or runtime. The different variability types, such as positive, negative optional are discussed as in (Anastasopoulos & Gacek. 2001) and presented in chapter 3.3.4 of this document.
Since the company our study took place in does not follow traditional product line approaches, we have reviewed an experience report of a study that took place in a company that did not make conscious use of software product line engineering techniques (Benavides & Galindo. 2014). The company manages variability by simply providing a base project, which contains all optional parts.

4
Methodology

This study follows the guidelines for design research in information systems, as presented by (Vaishnavi & Kuechler, 2004). In this section we describe the steps we followed in order to propose our suggestion and collect feedback. In summary, the steps we followed were as follows:

1. Understand the context of the problem by performing semi-structured interviews and code analysis.
2. Capture, document and analyze the extracted information.
3. Perform literature review to explore available solutions from the state of the art.
4. Provide trade-off analysis for each variability point.
5. Conceive a solution and realize a prototype for showing the idea.
6. Evaluate by collecting feedback from key stakeholders in the company
7. Iterate this process, refine the method, expand the scope and reevaluate

4.1 Research Method

The aim of this thesis was to focus on a particular problem and suggest a method to solve it. We followed the steps of the model proposed by (Vaishnavi & Kuechler, 2004 p.7), we first had to understand the context of the problem. Then, we explored the current available solutions and select the optimum by providing the reasoning behind. This enabled us to suggest a way to solve it by designing a prototype. We then proceeded to evaluate our suggestion based on the feedback of the company's stakeholders. Based on that we further refined our method. Additionally, to assess whether or not our approach could be applied to another similar context, we used the findings of the first iteration to suggest a way for another interface. Feedback from experts was once again collected and documented. The figure 4.1 shows the process steps we followed.

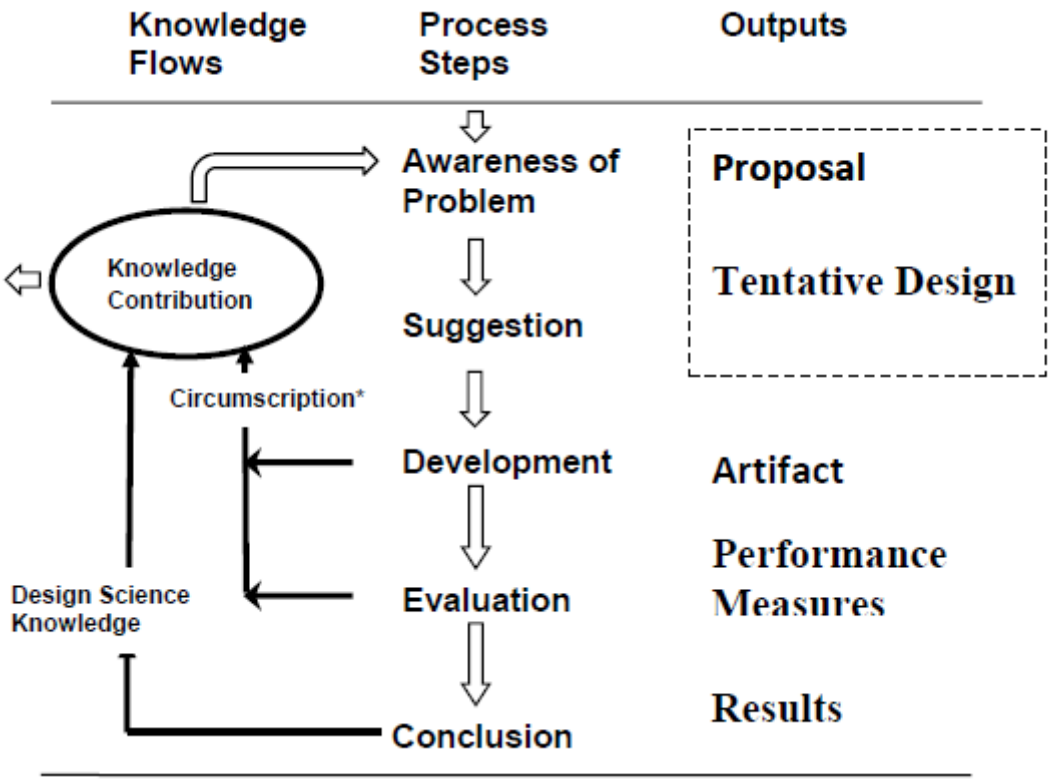

Figure 4.1: The process model for design research. Taken from (Vaishnavi & Kuechler, 2004 p.7)

4.2 Awareness of the problem

Initially, the focus was placed on the SSIM interface only. After concluding the first iteration with the evaluation we decided to expand the scope to a second interface. Due to time constrains and the required effort to build the required components, the second iteration had a more theoretical approach.

4.2.1 Interviews

In an attempt to extract the knowledge which was dispersed in several stakeholders of the company, we conducted a number of semi-structured interviews. In this way, we gained an insight about the context of the SSIM interface, the customization procedures and other relevant technical information. The identification of right interviewees was considered fundamental for the study. We wanted to extract as much relevant information as possible. For this reason we focused on stakeholders who worked with interfaces before, either as developers or by interacting with customers. Some interviewees had a hard time to give us relevant information that we needed but they helped us by pointing some other person who could help us more during our analysis phase. We only kept the information we believed was relevant for our study, after the interviews were complete. Irrelevant information for example could be about the difficulty to understand error reports when the parsing of SSIM files

fail, as it had nothing to do with improving the customization of this interface. Those stakeholders were in total twelve. We list them below:

- A developer who worked in the implementation of the SSIM parser in the old system.
- Three software engineers involved in the development of the new system, including the SSIM parser.
- A system's expert involved in the production of the new system who also gave us an insight of the new system's architecture.
- A system Architects of the Core Technical Development, involved in the development of the old system but also provides support in the new system's architecture.
- A process consultant who also plays the role of the Product Owner of the new system.
- A system expert who is also a Subject Matter Expert in pairing systems and with vast experience in the old system.
- Two service managers from the Service center, who come in contact with clients.
- A systems expert from the Implementation Department, who engage in customization and delivery procedures, sometimes with the help of customers.
- A Business Consultant involved in premarket and benchmark activities

The interviews followed a semi-structured approach by preparing questions beforehand. Each interview was planned for at least thirty minutes. However, some of the interviews took longer, with a maximum time of one hour and twenty minutes, as we allowed the interviewees to elaborate more based on topics that were considered relevant to the study. All the interviews were recorded and later transcribed for further analysis.
Additionally, in an attempt to mitigate the risk that we might have missed some important aspect or that the questions might have been misunderstood by the experts and also to ensure the consistency of our findings a group interview also took place. In the group interview engineers from both the new and old system were involved and we allowed them to discuss with each other for each point that was raised. The total time of the group interview was approximately sixty minutes.
The main focus of all interviews was placed on identifying the variability issues within the SSIM interface. However, to further facilitate our understanding of the general context that this interface operates in, other issues were also taken into consideration. Discussion about the high level architecture of the new system as well as the general demonstrations of the company's business goals and challenges took place during this phase.

4.2.2 Code and document analysis

Our study aims to allow the new system's integrator to become more configurable. We therefore considered code analysis of both the old and the new system's source code as an essential part of our study. Main focus was placed on the translation

process of the SSIM files, containing raw data, to semantics which the company's system can understand. We analyzed the code both of the old and the new system. Especially in the new system, code analysis was essential for the development of the prototype as it required understanding of the classes, their methods, their data structures and their dependencies.
For the code analysis of the old system an engineer who had worked a lot on the SSIM integration part was also involved. We used screen capturing software to record the code analysis in order to facilitate its documentation. In this way we could look into the code multiple time with the expert's commentary and grasp the issues of the old approach. He explained how the code was written and refactored in the past, the challenges they have faced and the causes of complexity. Moreover he proceed to demonstrate the functionality of the system by running some test cases. The code analysis of the new system was mostly done manually by us. However, support of the engineers was provided by answering our questions and providing short explanation of the reasoning behind their implementation. We have also been involved in the new system's team meetings where we kept notes of their discussions. These meetings discussed mostly the architectural aspects of the new system's integrator.
Finally, analysis of relevant company documentation was applied. These documents included the 2011 version of the IATA standard manual, architecture diagrams, official manuals and tutorials of both systems. Especially the manual of the IATA standard helped us a lot to understand the code of both systems and based on that we got a grasp of the customer specific use cases that have appeared in the past or might appear in the future.

4.3 Literature review

In order to get a background of the research problem, a set of research papers, journal articles and dissertations have been selected. The core focus was placed on the variability management and handling, software customization and design patterns. Our purpose is to adopt a method based on the state-of-the-art variability handling methods which we consider relevant to the company's technical context. In the previous chapter we discussed the related work based on the literature review. We started by reviewing a systematic literature about variability in software systems done by (Galster et al. 2014). We read the summary of variability mechanisms in product lines by (Lee & Hwang 2014) and also reviewed the referenced papers. We adopted definitions presented in (Svahnberg, van Gurp & Bosch, 2001), a paper which is cited many times. We also followed their taxonomy for variability realization techniques in (Svahnberg, van Gurp & Bosch, 2005) which is also referenced by many other papers. The keywords we used to obtain references from various digital libraries were closely related to variability management. Literature related to variation point models, patterns for software variability, variability modelling and software customization were selected. The appropriateness of the selected literature is evaluated by reading the abstract and introduction, the proposed tools and frameworks and their implementation or the overview of the method and finally the conclusion of the study. In the related work chapter we presented the main methods

we discovered.

4.4 Trade off analysis and development

We were looking for a solution which would adhere to the requirements of the company's stakeholders. We therefore wanted a solution with as minimal effort to implement as possible. During the trade off analysis we looked through the available methods in the literature; we assessed their applicability on the company's context and their expected strengths and weaknesses.
We documented the results of the analysis and discussed the different methods with stakeholders in the company. Based on their feedback we decided which of the methods were more appropriate for interface customization and we proceeded to develop a prototype. In chapter 6 we presented the trade off analysis for each method.

4.5 Development

Having concluded with the trade off analysis, we picked a method and went on to design a prototype. The initial design was on the core Java implementation. However, when the company changed to Python, we went a step back and tried another approach.
The idea was to observe whether or not we could support the customization of the use cases we found through the interviews. Each use case was dealt in isolation and we observed whether or not our suggestion could support them. The development was done in a test-driven development approach. We started by writing a test case which represented a use case and let it fail. We then wrote code which would make the test case pass. The prototype included a separate python module which would inherit and overwrite parts of the system's core code. The core code was left intact.

4.6 Evaluation

After the completion of the prototype we proceeded to evaluate it. We wanted to get early feedback to see whether or not our method is aligned with the company's customization needs so as to decide how we would continue. For this purpose, we invited experts from the company to collect initial feedback. The stakeholders involved were a system expert of the new system and a service manager from the service center department. The system expert was the person who wrote the parser's code in Python. The person from the service center was selected because he comes in contact with customers.
The evaluation started by asking permission to record the evaluation. We later used slides and performed a live demonstration of our prototype. We asked their opinion of the strengths and weaknesses of our approach. We transcribed and analyzed their feedback. Based on it we would then proceed to refine our method further.

The final evaluation took place after the refinement of our original approach. The stakeholders involved were engineers of the new system, one of the new system's architect, an architect from the core development product, a line manager and a stakeholder from the implementation team. The presentation included slides which gave an overview of our work, the problem at hand and the flow of logic of how we concluded to our results. After the presentation we encouraged the employees to give us their opinion. We stressed the importance of why they think this way. We encouraged them to elaborate on the reason they think our approach is good or bad and to discuss with each other.

5

Towards effective variability handling

The first step towards an effective way to handle variability is by understanding it. This is also required if we want to answer the first research question regarding the variability needs and limitations. For each variability point, we identify the variable item, why different airlines have different needs and which are the required variants, as suggested by (Pohl, Böckle, Linden. 2005 p60).
In this chapter we present the main variation points of SSIM and Operational Messages interfaces. We then present an overview of how the old and the new system parse SSIM files. Finally, we discuss the indicators which influenced our decision making and provide a trade-off analysis of variability realization mechanisms.

5.1 Variation inside standard interfaces

5.1.1 Main Variation Points of SSIM

The results of the analysis phase led to the identification and grouping of the main variation points of the investigated interface. We use the term variation point to describe the parts of the standard which are open for interpretation from customer to customer. The system needs to be aware of these fields and their required variants, providing a mechanism to handle the possible use cases.
Some of the issues of more technical nature were not considered as variation points. These issues were mainly related with the inability of some customers to produce proper SSIM files complying with the IATA standard. For example, a customer might include four bytes in a field that is meant to contain only two, ruining the syntax for the rest of the fields. It is then up to the customer to fix the syntax of the corrupted files and resend it again.
During our interviews we observed that the various stakeholders had some difficulty to remember all the use cases which appeared in the past. During the group interview, the significance of each variation point was discussed. The general feeling was that the standard should be the same for everyone and there should be no room for interpretation. However some stakeholders strongly believe that this can hardly be the case and it is very likely that airlines will impose their own use cases, as it happened before. By the time our study started, the new system was supporting only one customer and the requirements for the next ones were not known.
We summarize the main variation points of the SSIM in figure 5.1 as follows:

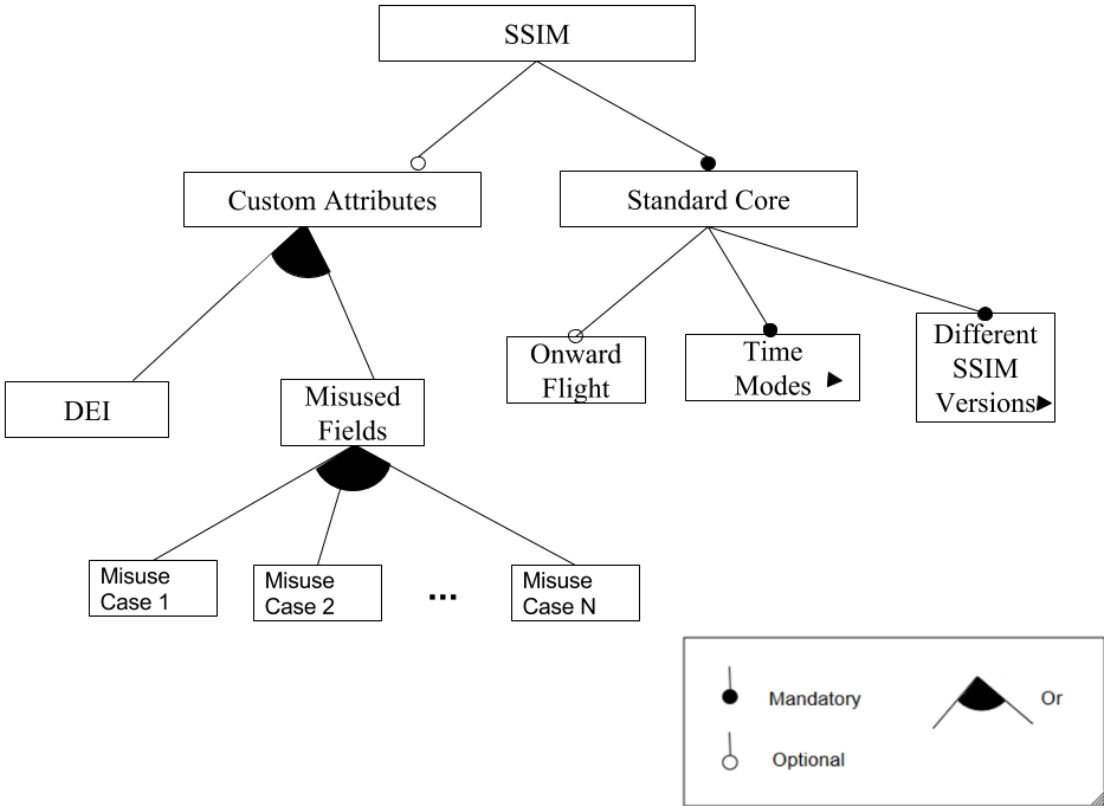

Figure 5.1: Main SSIM variability points using feauture diagrams

The SSIM standard contains two main parts. The standard core part which contains the fields shared by all the customers. We found three variation points inside the standard core. The first one is an optional field for the onward flight information. The second one is the different time modes. Lastly, there can be different versions of SSIM followed by different airlines. We illustrate further the variants of the variation points for the time modes and different SSIM versions in figure 5.2.
The second part is about extending the standard through custom attributes where the customers can add more functionality. Custom attributes can take the form of data element identifiers (DEI) which are defined by the IATA manual. They can also take the form of misused fields, where customers violate the syntax in some way.

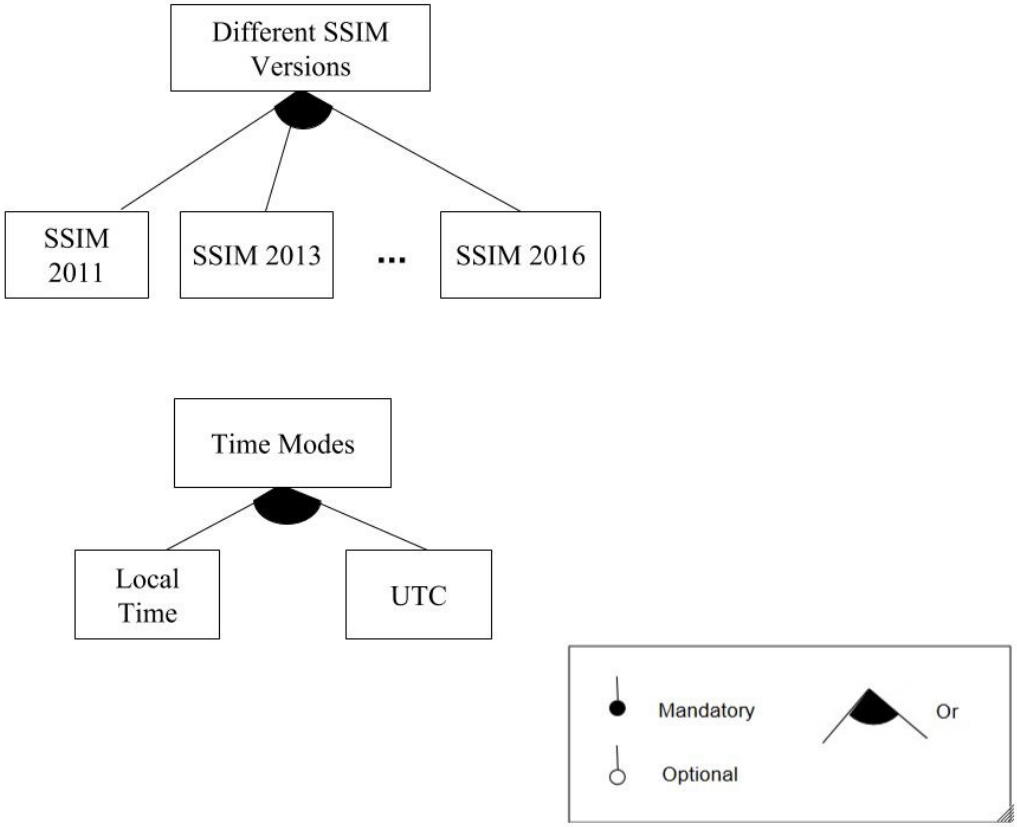

Figure 5.2: Different SSIM Versions and time modes

Depending on the IATA manual issue which an airline uses, different SSIM versions can exist. Some airlines might even use very old versions. Time modes can be either in Local Time or UTC. In the following sub sections we include more details for each variation point.

5.1.2 Data element identifiers

We begin with the Data Element Identifiers (DEI) as the first identified variation point. These can be used to extend the SSIM interface. This way customers can address their individual and distinct needs. DEIs take the form of an integer in a specific field inside the interface. Another string field corresponding to this DEI contains information to address an airlines specific requirements. The DEIs are described by the IATA manual and each integer has a different meaning. Currently there were only four DEIs supported inside the core code of the new system, as their total number can be a few hundred different cases.

5.1.3 Misused Fields

Moreover, there can exist customer specific use cases which can not be handled by the use of DEIs.The system needs to know how it should handle each customer's individual case. An example of such case is code-share issues, where an airline might wish to replace the subsidiary carriers with the main carrier. Another example is the

departure time shown to the customers and the actual departure time of the aircraft. This might occur because in some airports the passengers are being transported with buses to the aircraft, so there might be a few minutes variation. However, from crew planning perspective the aircraft time is what is important. As a last example, some customers use the flight service type, such as cargo only, in the field that is meant for the flight suffix. Finally, there could exist a case where an airline might want to use a field inside the SSIM interface which is usually not read by Jeppesen´s systems. Therefore the system should be aware of this case and act accordingly. During the group interview all the stakeholders agreed this was the most urgent variability point.

5.1.4 Different SSIM versions

In addition to the custom attributes, different SSIM versions exist. The IATA standard releases a new version of their manual twice a year. During our investigation we used the 2011 version as a reference. Airlines whose systems use a very old version might not be compatible with the newer ones. Throughout the different versions, lots of fields have new meanings. Some unused or empty fields start being used. Some of the fields that the company normally does not read might be required by some customers. However, according to some interviewees in the group interview, this is rarely a problem as it is very rarely changed in a major way. Furthermore, some clients might have a completely new use case which results on a modified version of the SSIM standard.

5.1.5 Time modes

Another issue is related with the time mode an airline company is using. More precisely, for the date of operation of each flight leg can be specified either in UTC or in Local Time. Although the IATA standard states that the time shall be in UTC, some customers still might want to store it in local time because this was the time format their systems were using. It should be noted that the handling of this issue is responsible for a significant percentage of the old system's code complexity, according to engineers in the company. As an example, a flight from an American airport can take place late at night. If the airline company sends the time in Local Time, the day of operation is different in UTC time because it is a different day in Europe. This can cause confusion, especially since the day of operation for each flight should be unique according to the IATA standard.

5.1.6 Onward flight

Finally, stakeholders raised the issue of an optional field containing information about the onward flights. These flights are concerned with the next leg flown by the same aircraft. The information about these flights can either be missing, or it can be included and still not be consistent. The system should be able to handle these cases as this information is necessary for planning purposes.

5.1.7 Main variation points of Operational messages interface

We identified four variation points for the operational messages interface. The first variation point is concerned with whether or not the aircraft rotations need to be updated. Different airlines have different scheduling process. In this interface, the company can choose to either update the onward aircraft rotations by performing an action or keep the same information provided by the SSIM. If the airline actually wishes to update the onward information, functions which swap flight legs or assigning new pointers in the rotations can be called.
The next variation point is about the time mode for the date of origin. The manual for the operational messages says the time should be in UTC, but some airlines especially in America still use local date of origin because it is easier to keep unique. However, the time obtained from the SSIM interface can still be in Local Time. As an example, a flight starting late at night in USA which uses Local Time, appears to be the next day in Europe. The system therefore needs to know how to properly adjust the date accordingly.
Another variation point is about the diversions from original schedule. When a flight leg can not land to the original arrival station it might have to land to another airport. This can occur, for example, because of harsh weather conditions. The follow up messages might be different between airlines. The aircraft might have to go back to its original destination, or continue to the next destination. The variation lies on the assumption of what the aircraft should do, as long as not further message arrives or opposite, for what situation additional messages are expected
Finally, the last identified variation point is concerned with the reliability of the different sources the system gets messages from. If an airline company has an internal consolidation system, then this requirement is not needed at all. An example of different sources reliability would be the messages for the actual and estimated time of departure. For actual time of departure the system would typically trust more the messages sent from the aircraft, while for estimated messages it would trust those sent from the airport.

5.2 The old and the new system

In this section we present the difference between the old and the new system for when it comes to parse SSIM files.

5.2.1 The old system

The variability binding takes different approaches in the two systems. The old system provides a form that allows its users to define many parameters when they wish to import the raw data from SSIM files, giving them the capability to configure the system in run time. These parameters correspond to one more flags of the tool. Some of them are taken from the input form, some are taken from the configuration files. The files are then converted to a specified format and handled by the same component of the tool.

The original code, which was written in C language, was closely following the IATA format and it was expected to be used in the same way by all the customers. In reality, the algorithms had to be adjusted to accommodate more customer specific requirements. This resulted in an extremely complex code which was hard to read and manage, as described by the company's experts. Although it works correct and satisfies the expected functional requirements, it is hard to maintain as there is a high risk of problems occurring when the system is updated.

5.2.2 The new system

The new system divides the work of interpretation and logic handling between the adaptor and the back end, as shown in Figure 5.3

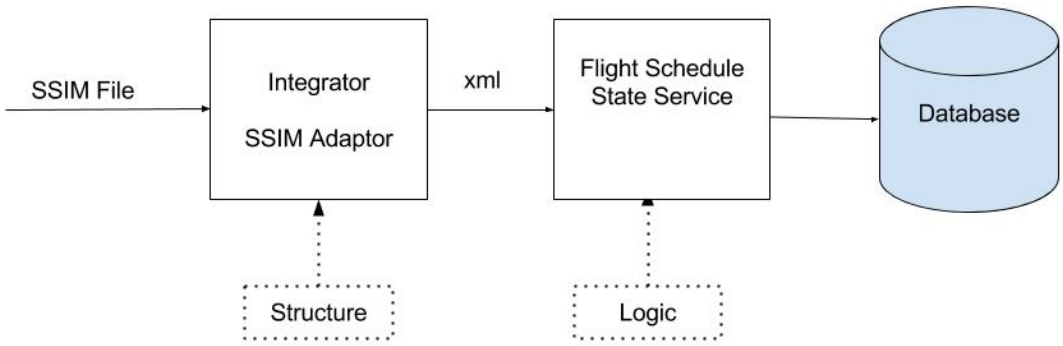

Figure 5.3: General overview of parsing an SSIM file and storing it in the database

The aim is to have separation of concerns between these two components. The SSIM file is generated by an airline company and used as an input to the integrator. The integrator, consecutively, uses an adaptor to define the structure for the SSIM standard in XML form. The "Flight Schedule State Service" component imports this XML file and applies some well defined computations to handle the given semantics accordingly. Finally it stores in the database the converted information.
The integrator's code of the new system was originally written in Java. However, during our analysis phase, the company changed the programming language to Python. Although the logic stayed somewhat the same, we had to perform additional code analysis and change our documentation appropriately. For example, the class diagrams had to be redrawn. Based on these diagrams we later placed our proposed solution.
An important aspect of our decision making was, for each variability point, which of these two components had to be selected. Whether or not it was a structural or logical problem. Finally, it should be mentioned that the old system is managing variability in run time while in the new system the variability is bound in build time. This means, the old system allows the users to change the interpretation of the parsing of SSIM files by providing an input form containing a number of parameters. However, the new system was initially designed and built to manage customer

specific use cases. Later, when they changed the technology from Java to Python, they still provided plug-in scripts which extent the system's core architecture that address customer specific issues.

5.3 Indicators for selecting a method

There is a number of indicators that greatly influences our decision making for selecting one of the available methodologies. We tried to suggest a way to handle each variation point independently. The first step to manage each of these, according to the guidelines provided by the taxonomy of (Svahnberg, van Gurp & Bosch, 2005 p708), is to identify the variability and where it is needed. These are the variation points which we identified during the context's analysis phase.
The next step, based on these guidelines, is to constrain the variability. A variation point which can include a significant number of possible alternatives, is often preferable not to include all of them in the first version of the system. The system should include those that are more likely to appear and allow the possibility to accommodate more in the future (Svahnberg, van Gurp & Bosch, 2005 p710).
Additionally, the current architecture plays a significant role to our decision making, as there is currently a single fully-working instance of the system. There is a separation of concerns between the parsing of the data and the actual logic handler. We therefore need to decide, for each variability point, which component should we use for our implementation. The technology of the system also is taken into consideration along with its strengths and limitations.
The next step, according to (Svahnberg, van Gurp & Bosch, 2005 p714) is to populate the variant feature, that is, how the variant should be created and integrated inside the existing components of the system. The population can either be implicit or explicit. Implicit population means the system does not recognize the available variants. Explicit population means the system actually provides functionality to support the required variants. As an example taken from the authors of the taxonomy, an *IF-Statement* is implicit population, since the system can not usually add more *ELSE-Statements* during run time. This decision indicates when should the binding time of the variants be done. This can either be in build or run time.
In the old system, some of the variants were supported during run time system. The users were able to select the variants they wish during the system's operation through the provided input form when they were loading the SSIM files. There was also a build time option where users could parse a configuration file along with the SSIM files to specify their additional requirements. These files included code which corresponded to the DEIs, as specified in the IATA manual. In the case of the new system though, the engineers prefer a build time solution. However, run time variability might be supported in the future for some of the variation points.
Another very important factor is the decision of who is expected to populate a variant feature by writing or compiling code (Svahnberg, van Gurp & Bosch, 2005 p715). The end users can still make use of the customization layer to write their own scripts. After all, Jeppesen allows their customers to customize their system according to their needs through domain specific language and by Python scripts. As mentioned

before, the customers are able to customize the system on the customization layer. For compiling and building the system, it is up to the engineers of the company. Finally, a decision should be taken of how to bind the variants. This can either be done internally or externally (Svahnberg, van Gurp & Bosch, 2005 p717). Internal binding means that the system provides the required functionality for a particular variation point. External binding means the system makes use of an external entity that implements the binding, such as an external module, another system, an external tool or even a person.

5.3.1 Summary of indicators

The indicators are based on the feedback of the engineers of the new system during the analysis phase of our study.
Below we present the summary of the indicators:

- The collection of variants should be populated *implicitly*. The company does not wish to include or maintain customer specific code. Therefore, the system needs to include code to only satisfy a particular customer's use cases.
- The system binds the variability during build time, where it is designed and handed over to the customer.
- When it comes to who needs to write the required code, the customers are expected to modify the system to handle their variability needs in the form of Python scripts.
- This means that the functionality should be bounded *externally*.

Some of the variability points can be handled differently. The initial assumption of our research was that the collection of the variants had to be done explicitly, which implied that run time variability and internal binding was required (Svahnberg, van Gurp & Bosch, 2005). While we choose not to follow this path for the rest of our study, mainly due to the feedback we got from the company's stakeholders, some of the variation points can still be handled in run time. The main reason for this argument is that this was the case in the old system.

5.4 Trade-off Analysis of Variability Realization Mechanisms

We have discussed in a previous section about the main realization mechanisms for variability, found in the generic literature. In this section we present the strengths and weaknesses for each of these mechanisms.

5.4.1 Generators

According to an experience report for variability management in the field of avionics (Wölfl et al. 2015) generators can follow an asset-based development dividing the work to domain and application engineering and generating source code. This

approach is more suitable when a large number of variation points is expected. This is also supported by (Bachmann & Bass 2001 p7) The benefit is that the quality of code increases and only the generator is needed to be maintained and evolved to support new variability points and also enhances the understandability of the system. The weakness of this technique is the upfront investment that is required since generators imply the development of meta models, domain specific languages and transformation rules, as mentioned in (Wölfl et al. 2015) and (Bachmann & Bass 2001 p7).

5.4.2 Configuration management systems

Additionally, this approach is efficient when dealing with alternative implementations and it is easier to maintain. However, it is difficult to manage optional features and, like the generator, it requires upfront investment although probably not as expensive as building a generator (Bachmann & Bass 2001 p7). This approach implies that the company maintains customer specific modules from its side. An alternative to this approach is simply making use of version control systems to keep track of the different implementations of a module. (SEI 2000 p53)

5.4.3 Self-Adaptive systems

According to (Van den Heuvel, Weigand, Hiel. 2007), this approach is more suitable for the automation of solving interoperability conflicts through the use of learning mechanisms rather than handling variability issues. Strong emphasis is placed on addressing performance issues. The effort required to define the algorithm which chooses the best of the available options require some significant effort.

5.4.4 Code level approaches

Code level, or lower level approaches, require less effort to implement but due the manual effort which is required, they might not be suitable for a large number of requirements (Wölfl et al. 2015). As mentioned in previous chapter, there are many ways to make use of code level techniques. Object oriented mechanisms include techniques such as inheritance, extensions and polymorphism (Anastasopoulos & Gacek. 2001) (Lee, Hwang 2014). Design patterns where the benefits and drawbacks are known beforehand (Gamma et al. 1995) can influence our decision. Furthermore, code level techniques can provide hybrid approaches. In the variability realization mechanisms of the taxonomy in (Svahnberg, Gurp, Bosch 2005) some of the approaches make use of combination of design patterns, mainly to handle run time variability. Additionally, for build time binding, there are approaches which make use of configuration management tools and architecture reorganization.

5.4.5 Summary

In figure 5.4 we summarize the main available methods with their strengths and weaknesses.

Method	Description	Strengths	Weaknesses
Generators	Generates customer specific artifacts based on a given input	Powerful tool for large number of use cases	Upfront investment
Configuration Management Systems	Produces customer specific modules.	Efficient when dealing with alternative Implementations Easier to maintain.	Requires maintenance of customer specific modules
Self-adaptive systems	Selection from a repository of scripts, dynamically, based on the state of the system.	Automatically solves protocol mismatches. Strong emphasis on performance.	More suitable for networked systems. Requires sophisticated algorithms
Code level methods	Software engineering and product line engineering patterns.	Easier to introduce. Effective when the number of use cases is not large.	Not suitable for too many use cases

Figure 5.4: Summary of strengths and weaknesses of the main methods

A generator would require too significant upfront investment. Additionally, there are currently not enough identified use cases. Therefore, the return of investment can not be easily estimated.
To make use of Configuration Management Tools include a cost of their purchase. Alternatively, the existing version control systems of the company could suffice. The problem of this approach is that maintaining a history of changes of each customer specific components and configuration still requires effort (Bachmann & Bass 2001 p7). This technique could be a viable approach if there were more concrete use cases and there was an urgent need to maintain all these different components from the side of Jeppesen.
Self-adaptive systems is a more viable approach for networked systems. However, maintaining a knowledge-base where the appropriate script to handle a certain situation could be proved viable, especially for run time binding.
Code level techniques appear to be more suitable in the case of the investigated interface. The main argument to support this decision is that the number of possible customer specific requirements is not expected to be too high. There is a significant number of different techniques. We choose to follow the general grouping discussed in (Völter, 2009), which is the removal, injection and parametrization as we discussed in 3.7.4.2. For adopting a more concrete realization mechanism, we choose to follow the taxonomy of (Svahnberg, Gurp, Bosch 2005). In figure 5.4 we borrow their summary. The indicators which we discussed previously can help us choose among those methods.

Section	Name	Introduction time	Open for adding variants	Collection of variants	Binding times	Functionality for binding
3.1	Architecture reorganization	Architecture design	Architecture design	Implicit	Product architecture derivation	External
3.2	Variant architecture component	Architecture design	Architecture design Detailed design	Implicit	Product architecture derivation	External
3.3	Optional architecture component	Architecture design	Architecture design	Implicit	Product architecture derivation	External
3.4	Binary replacement—linker directives	Architecture design	Linking	Implicit or explicit	Linking	External or internal
3.5	Binary replacement—physical	Architecture design	After compilation	Implicit	Before runtime	External
3.6	Infrastructure-centered architecture	Architecture design	Architecture design Linking Runtime	Implicit or explicit	Compilation runtime	Internal
3.7	Variant component specializations	Detailed design	Detailed design	Implicit	Product architecture derivation	External
3.8	Optional component specializations	Detailed design	Detailed design	Implicit	Product architecture derivation	External
3.9	Runtime variant component specializations	Detailed design	Detailed design	Explicit	Runtime	Internal
3.10	Variant component implementations	Architecture design	Detailed design	Explicit	Runtime	Internal
3.11	Condition on constant	Implementation	Implementation	Implicit	Compilation	Internal or external
3.12	Condition on variable	Implementation	Implementation	Implicit or explicit	Runtime	Internal
3.13	Code fragment superimposition	Compilation	Compilation	Implicit	Compilation or runtime	External

Figure 5.5: Summary of realization mechanisms. Taken from (Svahnberg, van Gurp & Bosch, 2005 p742)

We preferred a solution which does not require major architectural restructuring, as this would interfere with the company's current approaches. For this reason, we focused on two techniques, *Variant component* specializations and *Optional component specializations* as they comply with the identified indicators. In the next chapter we provide more details about the method we choose to follow and the reasoning behind it.

6

Development and implementation of proposed solution

The aim of our study is to create knowledge by attempting to solve a concrete customization problem. In this way, we tried to answer the second research question, that is, in which way we could increase the system's customizability. We proceeded to develop a prototype so as to show the idea. We started with the custom attributes variation point, which was considered as the most urgent from the company's stakeholders.
In this chapter we present the steps we followed during the development phase of our study. The discussion of the feedback is described in the next chapter.

6.1 Java implementation

The integrator's code was initially entirely written in Java. Since there was only one customer, there were not any known customer specific requirements for the new system. Our initial assumption was that the system would require run time binding, because this was the way it worked in the old system. Based on the taxonomy of (Svahnberg, van Gurp & Bosch, 2005 p735-737) and the figure 5.5 we considered two code-level realization techniques. The first one was run time variant component specializations and the second was variant component implementations. These techniques make use of design patterns in code level to manage variability. The reason we choose these techniques was based on the indicators which stated that the binding should be done in run time, the collection of variants should be explicit while the system provides internal functionality.
Our original suggestion was to apply behavioural design patterns. Behavioural patterns are linked with the assignment of responsibilities between objects and the handling of diverse algorithms. These patterns favour object composition over inheritance (Gamma et al. 1995 p221).
For the case of custom attributes, we tried to to make use of the strategy pattern. As originally presented in (Gamma et al. 1995 p315), this pattern defines a family of algorithms, encapsulates each one and makes them interchangeable. Strategy lets the algorithm to vary independently from the clients that use it. In this way we expected that we could encapsulate the required behaviours for the different misuse cases inside separate classes.
We tried to design the parser in a way to make use of this pattern by creating subclasses. These sub-classes would include the different algorithms for managing the

different misuse cases. In this way, we tried to separate the parts of the application that vary and the parts that belong to the core of the product, that need to stay the same. We consider the handling of customer specific use cases as different algorithms that affect the behaviour of the system.
Our assumption was that not many misused cases would appear since this was the general impression we obtained during the group interview. Otherwise, should their number become too big it could lead to class explosion, resulting in increased system complexity.

6.2 Change of technology

The company changed the programming language of the integrator from Java to Python. This was done in order to handle the parsing of multiple SSIM files at the same time. This functionality was also present in the old system.
There is now a main core application written in Java. This application calls python scripts which support the translation of the SSIM input to XML. Furthermore, the company provides customer specific scripts. These scripts inherit and overwrite parts of the core implementation. This way they adjust the system to adhere to each of their customer requirements. The customers can also edit these scripts since there is no need to recompile the system.

6.2.1 Code analysis results

The main logic stays somewhat the same with the Java approach. The core of the integrator uses python modules to define the structure and the general parsing rules. The module is called *ssim.py* which is included in the *std_interfaces _adaptors* core package.
As shown in figure 6.1, there is a distinction between the core and customer specific layer. The core part, which is on the right of the figure, handles the generic issues of transforming the SSIM into an xml file. The left part includes scripts which handle the concrete issues that are specific for a specific airline company. The DEIs are handled inside the *ssim.py* inside the package *std_interfaces _adaptors* of the core part. The IF-statement which handles the data element identifiers is relatively small when our research was performed but it could still significantly increase in size in the future.
The airline specific module *ssim.py* is included inside the *as_adaptors* package of the customer specific layer. The script inherits and overwrites parts of the super class it adheres to, without the use of interfaces or abstract classes. The size of this script is limited to less than fifty lines of code and handles two concrete use cases. This approach binds the variability in build time. There is clear separation between the core and varying customer specific solutions. In this way, the company increases the maintenance of the system since only the additional scripts need to be maintained which cause no ripple effects to other modules.

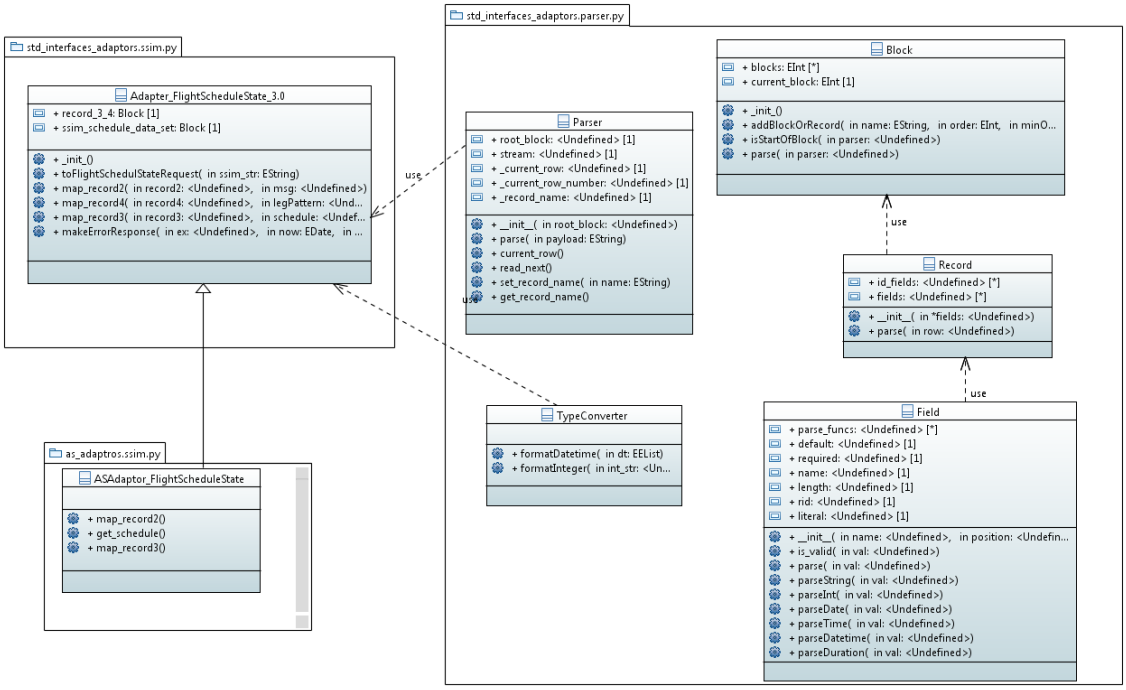

Figure 6.1: Class Diagram in Python

6.2.2 Issues of the current appraoch

The approach the company is following is quite efficient. The core stays the same for all customers and they are responsible for editing scripts to customize it further. They are also responsible for their maintenance and keep it compatible with their systems; Jeppesen loses control over the code.
During our study we raised some issues. Hard-coded conditions were included inside the customer specific scripts. The follow-up question was whether or not we could somehow reuse parts of the code. In this way, we hoped to decrease the amount of code required to handle use cases which appear more than once.
Another issue we raised was to what extent could the airlines write quality code by themselves. After all, they need to focus on airline related problems than programming related problems. Writing a Python script requires understanding of the system's core code as well as writing test cases to ensure the code provides the expected results.
Finally, there was the issue concerning what would happen when all the company's customers migrate to the new system. The maintenance is done by the airlines side which means they are responsible to keep compatible their systems. According to some stakeholders who come in contact with customers, they receive support calls when their customers can not figure out how to fix their systems and even sometimes they complain that they receive too much code to maintain.
By the time this study was performed, there were only one airline supported in the new platform and the company was talking with the second customer. The goal of this study was to look further ahead and suggest a way to mitigate emerging risks.

6.2.3 Suggested approach

Contrary to our initial approach, we decided our method should aim for build time binding, implicit collection of variants and external functionality. This decision was made by asking if run time binding was needed. This question was asked to the developers of the new system after the Java-to-Python transition was made.
Based on the above indicators, we looked into different techniques of the taxonomy, such as the variant component specializations and optional component specializations (Svahnberg, van Gurp & Bosch, 2005 p733-735). These choices follow the figure 5.5. and based on the indicators we discussed in section 5.3, we selected two methods as follows:
Variant component specializations suggest a method where a particular component is adjusted to addresses different variability issues. This is realized by creating a number of independent classes and selecting those who are required to be included for each case.
Optional component specializations is about including or excluding the relevant behaviour of a component for each case. This can be realized by developing a separate class which encapsulates the optional behavior (Svahnberg, van Gurp & Bosch, 2005 p733-735).
Inspired by those methods, our new approach suggested a grouping of the recurring methods in a separate module. This module would inherit and overwrite parts from the core implementation but the customers would only need to import this module on their scripts and use the available methods by simply parsing parameters.
The main idea was to support the customization by allowing the users to select among a set of provided functions instead of customizing those functions. In this way, we hoped to reduce the code duplication for handling these case as we expect that only the provided parameters will change in the future. We also hoped we would increase the understandability of the system.

6.3 Development process

To achieve this, we developed a new Python module which contains methods to handle misuse cases. We called this python module as *script_repository.py* and we created two children scripts, *chalmers.py* and *chalmers2.py* that made use of the provided scripts to handle a number of concrete use cases. In figure 6.2 we present the class diagram of our approach.

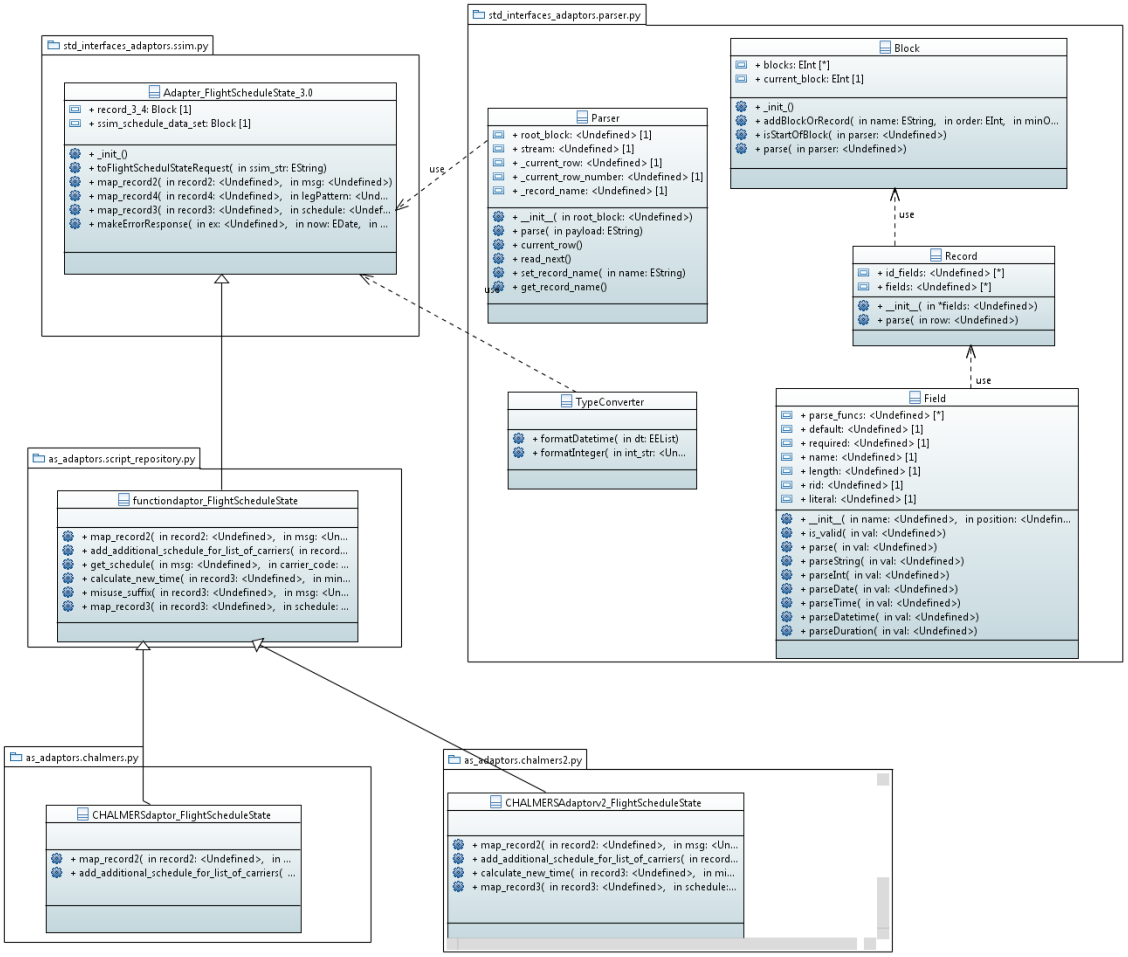

Figure 6.2: Approach using a script repository in Python

In order for the script repository to define algorithms for each use case, different parts of the core implementation had to be taken into consideration, such as the provided methods and data structure definitions.
Our prototype's was built in a test driven development approach; we first wrote a test case, we let it fail and we proceeded to write code to make it pass. For this reason, we have written a new file explicitly for writing our own test cases which were satisfied by the modules *chalmers.py* and *chalmers2.py*. Moreover, we have written a test case that only makes use of the core implementation in order to make sure we have not altered anything in this part and that the core is working as expected. The *script_repository.py* is the module which needs to be maintained and extended. The children modules only needed to call these methods and provide the appropriate parameters.

6.3.1 Example use cases

To show the idea behind our approach, we tried to support a few use case. Two of these use cases were making use of a certain airline's requirements, which the engineers have solved but in a hard-coded way. These use cases were related with

replacing the codes of the subsidiary carriers to to the main carrier and the inclusion of additional schedules to these subsidiary carriers. Our approach would only require the customer to import our module and call these methods with the corresponding parameters.
We also tried to solve some conceptualized use cases, based on what we discovered during our interviews in the analysis phase. The first use case had to deal with the time variation of the departure time. If the departure time had to be adjusted a few minutes, the script would again only require to import our repository and call the corresponding method with the appropriate parameters. Finally, we tried to handle the misplacement of the service type suffix, when some airlines use the operational suffix instead of service type field. We handled this case in the similar fashion as above, by calling the corresponding method. In this case, however, no parameters were required.
Our approach follows our initial assumption that not too many customer specific requirements would emerge. Should this be the case, there could be an explosion of methods. Additionally, these methods are only viable if they are required from at least more than one customer. In the next chapter we provide a more detail discussion about this approach.

6.3.2 The rest of the variation points

We have observed that not all of the issues can be handled with the same method. For each individual variation point a decision had to be made on which part of the system it should be handled. More precisely, for each variation point we had to decide whether it could be solved as a structure or logic issue. Therefore, the adapter or the logic-handler modules would had to be selected respectively.
The focal point of was placed mostly on the different misuse cases. We have investigated further of how to handle the rest of the variation points to decide whether or not it could be handled with our approach or in a different way. Our findings suggest that not all points could be handled with our suggestion.
Local time and UTC time adjustments require special logic handling. The algorithms responsible for this handling have already implemented in the logic component of the new system. The only requirement is that the inputted XML file provides the correct semantics to define whether the time is Local Time or UTC time. In this way, there is no need to adjust the system for each customer as it is able to handle all cases.
Similarly, the onward flight has been handled in this way. The algorithms to handle inconsistent or missing onward information is handled for all customers in the same way inside the logic part of the system. According to company's experts, there could still be a way to provide a generic method to handle this issue. However, the required parameters would probably be too many and the time needed to investigate how this could be implemented would probably be immense. We therefore decided that this variation point is out of the scope of our thesis.
We have verified that the DEIs are being effectively handled in the core part of the system. After all, DEIs follow are defined by IATA standard and do not misuse the SSIM syntax. We have validated this functionality by writing tests and compare

the expected output with a given input based on SSIM files we obtained from the company.
For special cases of certain DEI numbers, an IF-Statement in the core part suffices. This method is also described in the taxonomy of (Svahnberg, Gurp, Bosch 2005 p739) as *Condition on variable* which intent is to include functionality to support multiple operations. There are currently only a small subset of all the possible integers a DEI can have, as not all of them are required by all the customers right now. For each new customer who actually wants to use a new DEI, a new ELSE-Statement would be added. Therefore the overall IF-Statement could become potentially big and since it is currently included in the core part of the parser, it would might add up to its complexity. If this should be the case, we suggest to include this IF-Statement into a separate module and call it through the main core implementation by parsing the corresponding DEI.
Finally, when it came to handle the different versions of the SSIM interface, we observed that this could be handled in a similar way as with the misuse cases. If there are major differences between the supported version and the version an airline is using, it is implied that there is a need for adjustment. Company's experts have also agreed that this case could be considered as a special case of syntax misuse. As an example, a field of an older version could have a different meaning than the current version. Should this be the case, then a method from our repository could be called and adjust the system in the appropriate way.
However, during our investigation we have observed that if a customer wants the system to read a field that the company usually does not support, this can not be simply handled with our approach. We tried to write a method to handle this issue but it was futile as the system would require to be rebuilt and recompiled to become aware of this field. Our conclusion, supported by the developers of the new system, was that this can only be solved manually by the company's side.

6.3.3 Operational messages variation points

We tried to discuss ways to deal with the variation points of the operational messages interface. The suggestions include high-level methods to handle variability in this context. The aim to see whether or not the suggestions made for the SSIM interface could be replicated in this context. The discussion involved the system architect of the new system. Evaluation of this discussion was performed during the last presentation in the company which also involved engineers of the new system. Everyone seemed to agree with these suggestions. We assume that the company would follow the same logic of separation of concerns, where the raw input is translated to XML semantics and is being handled in the backend.
For the updated aircraft rotations variation point, there is a need to indicate whether or not the flight legs need to be updated. The decision whether or not to update these rotations could therefore be considered as a structural issue. Depending on an indicator, the corresponding function could be called. By default it would assume that there is no need to update the rotations at all. The functions would be written by Jeppesen engineers but the selection could be done by the customers. Therefore the algorithms could be part of the core and encapsulated in the backend. The

customers could simply edit a script to include an indicator to call these functions. In a similar fashion the date of origin could be managed. A flag in the translated xml file could indicate that a system is using local time. The recalculation is done in the backend part of the core. Therefore it is shared among all customers and the functions are written by the company's engineers. The selection among the functions could simply be done by the customers.
The reliability of the different priorities is a rather complex variation point. It could be done entirely in the customization layer and it would require explicit functions which would overwrite parts of the core. The system could prepare support for priority handling, either in the core or in the customization layer.
Finally, the diversions variation point could be viewed as a structural issue. The point is to define which would be the next flight leg. This could be handled by editing a script and calling the corresponding function and parsing parameters. This is intertwined with the repository of scripts suggestion of this study. However, if the functions are proven to be rather hard to replicate because of vast numbers of parameters, documentation and coding standards could facilitate this purpose.

7
Results and discussion

Our suggestion was heavily influenced by the company's customization needs. Therefore, our suggested method needed to be easy to introduce, implement, maintain and extend. In this chapter we will discuss the results of our findings by discussing the strengths and weaknesses of the method and the knowledge gained from this endeavour.

7.1 Validation

The use cases handled by the prototype provided some satisfying results. In order to use the script repository, an airline company only needs to import our module and call one of the available methods by parsing parameters. The total amount of lines of code a customer needs to write in order to use a method from the repository approximately four or five, including the import of our module in the beginning. Including an additional method requires at maximum three extra lines of code. The company's current approach required approximately ten to fifteen lines of code for developing a customer specific method which inherits and overwrites part of the core implementation.
Additionally, we developed our own test cases to validate that we were getting the expected results. The test cases compared a string of raw data input in the form of SSIM format to a string of the expected XML file. Their customers do not require to write the test cases by themselves. We have included commented test cases in a package where they can adjust fields of the SSIM and XML accordingly. In this way we hope that we can guarantee that our methods will work.
Finally, there is no need for airlines to understand how the core's code or the methods inside the script repository work. The algorithms and data structures needed to handle these use cases as they have already been implemented from the company's side.

7.2 Evaluation

After the completion of our prototype, we proceeded to evaluate it by asking for feedback from company's stakeholders. Initially, we invited two key stakeholders to evaluate our approach. Based on this evaluation, we went on to discuss with a systems architect for refining our method. In this way we identified the guidelines for decision making which we describe in a later subsection on this chapter. After that, we invited stakeholders to our final evaluation where we presented our work, our

suggested method for both SSIM and Operational Messages interfaces, the decision making process and ideas for future work.

7.2.1 Preliminary feedback

We performed a presentation with two key stakeholders of the company to evaluate our suggested method. The first stakeholder was the developer who wrote the Python code of the integrator in the new system. The second stakeholder was a person who works in the service center of the company. We used slides to show the idea behind our approach. These slides also included code parts which compare our method with the current implementation. The evaluation was recorded and later transcribed. We present the main points from the feedback we got in this section.
At first, our approach received was treated with scepticism. The main reason was that they did not intend to maintain customer specific code at all. According to the first stakeholder, it is up to the clients to edit Python scripts to modify the system according to their needs, especially since Python language does not require to recompilation of the system.
The second reason was whether or not they could reuse the methods to handle customer specific issues. According to the first stakeholder, this approach would add up to the complexity of their system's modules. There were currently no concrete evidence to support the argument that customer specific methods could be reused. The reason was the use cases were unknown to them since the new system supported only one customer at that moment.
Additionally, he argued that this repository of methods could become potentially large and hard to be used; the clients might feel lost of which method they need to use, especially if some methods had to be used together. The second stakeholder then also stated that he was sceptical for the degree on which the methods could be reused. As an example, he talked about the code sharing issue; whether it could be handled in the same way for multiple airlines or each airline had their own interpretation.
On the other hand, our approach received some positive feedback as well. The person from the service center stated that our approach simplifies the system's use from the client perspective. He argued that their clients are airlines and not IT companies.
More importantly, he argued that in this way they would enable their customers to focus on actually solving their real life problems, airline related problems, rather than programming in the system. The methods are viewed as black boxes from client's perspective and might be hard for them to edit them by their side.
Finally, he stated that this would facilitate the overall understandability of the system. As an example, after a long period of time someone wants to dig back into the code, separating customer specific issues into a separate module would be easier to get an overview of the integrator.
The new system's developer also argued that our approach would be a good idea if more than one clients needs these functions. Furthermore, he added that, should there be a case where they realize a code is duplicated to multiple clients, providing some kind of utility would actually be a good idea. Additionally he stated that if

they could find recurring patterns which are being used by many customers in other parts of their system this could be proven a good practice. However, he insisted that they currently prefer to duplicate code as they are not concerned with maintaining their customer's scripts from their side.
An interesting comment was that the issue of suffix misuse could be embodied in the core implementation as it should be the same for all customers. This method did not take any arguments, it was simply called to fix a problem. Instead including it in a separate module, it could be manifested in the core instead.
We have also obtained feedback from another stakeholder, a system's architect who is involved in the support team of the new system. According to her, the main argument to adopt our approach is whether or not the functions can be used from more than one customers. She also added that one of the expected benefits of our approach would be the reduced number of calls the company receives for tech support. We summarize the results in the following table.

Strengths	Weaknesses
• Effective approach for handling multiple recurring issues	• Requires additional maintenance. Might increase complexity
• Increase system's understandability	• Tricky to handle too many methods
• Increase usability from client's perspective; focus is placed on solving their actual issues instead of writing code • Can reduce tech support requests	• It should not be included in the core

Table 7.1: Summary of the preliminary evaluation

7.2.2 Final Evaluation

The final evaluation was performed after the refinement of our method based on the initial feedback. It took the form of a presentation and a number of stakeholders were invited to attend. These stakeholders were in total six; two software engineers and an architect of the new system, a system architect of the core development involved in both systems, a systems expert of the implementation department and a line manager. The list of people we invited, however, was longer but only those could attend our presentation that day.
The presentation lasted almost an hour. We had prepared a few questions to ask beforehand. We kept notes of their answers highlighting their main points and

immediately after the presentation was over we proceeded to documented the results. We started by discussing the identified variability management mechanisms from the generic literature and the variation points of the SSIM interface. We then discussed the current approach of handling the customer specific requests, its strengths and possible risks. We used two examples to illustrate what could go wrong with the current method.
Next, we introduced our suggested method and presented the preliminary feedback summarized in a table. At this point, we asked the stakeholders in the room for their opinion and why they think this way. At this point, a software engineer expressed his scepticism. He explained that although he understands the idea behind our approach, he was still unsure whether this would effectively contribute to their customization efforts as the company has already provided a solution for the SSIM case. The system architect replied to him that the company has solved the customization needs of the SSIM already multiple times and the point was that they do not want to solve it again and again for every customer. She therefore believed that our suggested approach has potential to facilitate the customization needs in the long run. The software engineer then seemed to agree with her and so did the rest of the stakeholders in the room.
At this point, we asked our audience whether they think our approach would increase the code quality from their customers side. More specifically, we asked whether they think our approach would assist their customers to write a better and cleaner code. The line manager stated that they always consider code quality as something very important and that he believed our suggestion could support this purpose. Additionally, the system architect stated that minimizing the dependencies and writing less lines of code for doing the same thing is always an improvement in the overall code quality.
We continued by presenting the guidelines for decision making to integrate a variation point based on its characteristics, as discussed in section 7.4. These ways of integration were as a shared component in the core, by instantiating only the required functions for a customer or in the form of coding standards. The audience seemed to agree. Someone pointed out that sometimes, whether a component belongs to the core or in the customer layer is sometimes debatable. Especially in the new system's architecture, it is even harder to know what is customization and what is core. They try to push more into the core and less in the customization layer, especially for tracking system and therefore he does not see a problem reusing parts as long as they appear to more than one customers.
We then discussed the variation points for the Operational Messages interface. We briefly described each variation point and its variants along with suggestions of how each of these could be handled in our way. The audience seemed to agree, although they did not give any concrete feedback.
We finally presented some ideas for future work. These triggered discussion among the stakeholders of how the company could support its customization needs more efficiently. One of these ideas was to maintain a knowledge repository. The stakeholders discussed about how sometimes do not know whether something should be in the customization layer or in the core, as there have been cases that were proven as more complex than originally thought or something appears second time and so

on. Therefore, maintaining the knowledge of each project could support future decisions. However, they pointed that no one usually wants to work on it as it is viewed as a tedious task. The second idea for future work was an interactive questionnaire whose purpose would be to support the requirements elicitation by providing the right questions to ask their customers.
In summary, the stakeholders agreed that maintaining from their side a repository of scripts could provide benefits in the long run. However, they do not plan to integrate it right away as there are still not concrete customer specific use cases and in general, they do not like to increase the overall maintenance by their side unless it is absolutely needed.
The general consensus for the decision making guidelines was that they could support their future decisions. Finally, the proposed future work received an overall positive feedback as it is intertwined with the company's current customization needs.

7.3 Integration

Although the company's stakeholders stated that they can not integrate a script repository right away due to the lack of concrete use cases, we decided to pushed our prototype in the company's code repository. The prototype includes a few example use cases and their respective test cases and it is ready to be used.
Additionally, based on the initial feedback we received, we proceeded to discuss with one of the architects who is involved in both the new and the old system's architectural design, about other ways to integrate our approach. We identified three different ways of integration. The selection among the above mechanisms is based on the expected level of reuse of each function.
The first one is by making use of a shared component. These components are shared between all clients. The customers simply choose which function they need to call every time. This approach would only be useful if those functions are being used by more than one customer otherwise there is a risk of ending up containing dead code. This is aligned with the idea behind our prototype.
The second way would be to instantiate the required functions for a particular project only. The functions are embodied in the customization layer of the system, including only the functions required by each individual customer. In this way customers are not required to maintain functions they do not need. The maintenance is done by the company's side; the customers simply copy a subset of the functions they need.
Finally, these functions might not be included at all to any component. Instead, they could take the form of documentation and coding standards. This could support the case where a function which can not be replicated to different customers and small adjustments need to be made each time. Coding standards could simplify the writing of those functions. In this way, there could be a common documentation instead of common code.
It should be noted that these suggestions were part of the final evaluation where stakeholders agreed about them. As described later in the future work, a knowledge-base would facilitate the decision making of how each function should be integrated.

7.4 Guidelines for decision making

The company does not follow traditional software product lines methods, as the customers are expected to edit and maintain code in the customization layer while the company is mostly responsible for the maintenance of the core implementation. In every customization case, decisions need to be taken. The SSIM interface gave us a strong basis to perform our study. Based on it, we document the guidelines in an attempt to support future customization decisions.
Each variation point needs to be dealt independently and in isolation. The characteristics of each point need to be identified; based on these, the decision of how it is going to be integrated in the system is taken. We therefore create a list of questions to ask so as to understand these characteristics, followed by a list of variability handling mechanisms.

7.4.1 Understanding the nature of the variation point

The first question is what is the likelihood that the variation point will appear more than once. Is it a special case required by only one customer or does it appear to multiple customers?
The second questions is concerned with the collection of the variants and subsequently, the binding time. If the variation point is shared by all the customers, then the collection is explicit and it is bound in run time. However, if the collection is implicit, the system is modified accordingly in build time.
The third question is about the problem itself, whether it is a structural or a logical problem. A structural problem is concerned with translating the raw input in the correct semantics. A logical problem is concerned with how an algorithm manipulates the provided semantics.
Finally, one important question is about the estimated complexity of the code to deal with a variation point. This means, it could either be a rather simple to write script from scratch, or it could require a deeper understanding of the system's core code and strong programming skills.

7.4.2 Implementation of a variation point

Having defined the characteristics of the variation point, the next step is to decide how and where it should be implemented. Again, a few decisions need to be taken. The first one is about who is expected to write code. If the expected complexity to implement a customer specific use case is expected to be low and it is unlikely that it will appear more than once, then it is up to the customers to write code. However, if the complexity is expected to be high, or the variation point is likely to appear more than once, then the company's engineers would need to develop a high-quality function.
Furthermore, if the problem type of the variation is a structural issue, the corresponding function could be encapsulated in the parser. If, on the other hand, it is a logical issue, an algorithm which provides diverse behaviour based on the given semantic input seems more of a reasonable option.

Finally, a mechanism which allows the users to access the variation point should be defined. If the collection of variants is explicit and their binding is done in run time, then the variation point should be embodied in the core of the system. In the opposite case, where the collection of variants is implicit and the binding is done in build time, the variation point could be part of the customization layer. It could be implemented as a new function, part of a repository of scripts, where clients simply choose those needed for their needs. If the variation point can be simply replicated by calling a function but requires to modify the function itself, the variation point could take the form of documentation and coding standards.

8

Threats of Validity

The value of this study is subjected to a number of validity threats. We classify the different types of threats of validity as described in (Wohlin et al. 2000).
Internal validity threats are concerned with how the treatment influences the outcome. There is a threat of internal validity in regards of selecting the right people to interview. To alleviate this issue, whenever we suspected the interviewees could not provide relevant for our study information, we asked if they could suggest some other stakeholder. Additionally, the interview questions might not have been understood clearly by the interviewees. To mitigate this risk we performed a group interview with stakeholders from the service center, the old system as well as in the new system. We also hoped that in this way we could encourage these stakeholders to elaborate more on the problem under study. Furthermore, the limited time to carry this study only allowed us to focus mostly on a single interface and limit the number of times we could evaluate the suggested methods.
Construct validity threats are concerned with the theory and the results of the study. The number of customer specific use cases were not clear. The stakeholders had rather hard time to remember concrete variability issues in the old system and there was only one customer in the new system by the time our study was performed. It is therefore not clear whether or not our suggestion will provide adequate support to manage all the future use cases in the long run. To alleviate this threat, we dealt with each use case individually. In this way, we strove to understand their characteristics and based on these to suggest ways to manage them. We document the decision making process and we suggest future work to further improve this study.
External validity threats are concerned with the generalizability of the findings beyond the scope of the study. This study was placed only on one company, located in Gothenburg, Sweden. The way this company handles variability is different than other companies who follow traditional software product lines. Therefore, the methods might not be easily replicated to other industrial contexts. Additionally, the study mainly focuses only on one interface which has a few known variation points. To alleviate this issue, expanded the scope of the study a little bit to investigate and suggest ways on a higher level to manage variability issues on another interface.
Conclusion validity threats are concerned with the ability to draw the correct conclusions. The conclusion whether or not the outcome of this study is a better approach than the current one the company is using has not been proven yet through integration. The evaluation is based on feedback we received by the company's stakeholders.
The preliminary evaluation involved an engineer of the new system and a person

working on the service center. Additionally, during the last presentation, where we presented the refined version of our approach and discussion about supporting the second interface, we involved more stakeholders. These stakeholders were a line manager, two software engineers of the new system, a system architect involved in the development of the new system, a systems expert of the implementation department and a system architect of the core technical development who is involved in both the new and the old system. The feedback was mostly positive. Although we had a bigger list of people to attend our presentation, not all of them could be present at that time due to work obligations. Finally, to reduce the risk of objectivity of the feedback received during the last presentation, we asked to state their reasons why they think something is a good or bad idea. Additionally, we encourage stakeholders to communicate with each other in order to hear a different perspective. We noticed before that sometimes the received feedback varied depending on individuals perspective. We therefore hope the subjectivity of the feedback was reduced by enabling stakeholders to openly state their opinion and hear other people's opinion as well.

9
Conclusion

Software customization has an impact throughout all the software life cycle such as requirements elicitation, the design of the system, development, deployment and maintenance. The majority of the existing research is suggesting ways of how to support customizability where a company follows traditional product lines approaches and the variations will live very long.
In the case of the company where this study was performed this was not the case. The company controls only the core part of the system and loses control of customer specific code. The customers are responsible for part of the code's maintenance. By the time the study took place they hardly had two customers using their newest system.
The aim of the study was to look further ahead and mitigate the risks which could potentially arise in the future. We explored the existing literature for methods, patterns and practices to suggest a way to handle variability in an easy, cheap and efficient way.
We provided a trade-off analysis of the available methods. Furthermore, we documented the work flow and how we conceived the suggested method in our study. We ambition that the findings of this work will provide a strong basis to facilitate customization efforts by companies of similar technical contexts.
The overall feedback we received from the company's stakeholders suggested that our method will be integrated in the future by the company. We therefore encourage validation of the suggested methods through integration and the expansion of the guidelines of the decision making process. For this purpose, we have already pushed a repository containing a number of sample functions, ready to be used by the employees of the company.
Additionally, we suggested ways for future work in section 9.1. We do not expect that the findings of this study will either solve all the customization problems of the company, nor will they reshape the vast landscape of software customization. We hope, however, that it will serve as a step forward to the vision of handling variability needs in an optimum way and add up to the existing knowledge for problems of similar nature.

9.1 Future work

The evaluation of the proposed methods in this study were based on the feedback of various stakeholders of the company. To justify whether or not these methods would actually provide the expected benefits, integration with the company's new

system is required. We hope that the prototype which we included in the company's code repository will be used by the employs and be validated further.
Additionally, the guidelines as well as the lessons learned of this study and similar projects should be retained in some way. A knowledge-base could therefore facilitate the purpose of continuously improve and refine existent knowledge. Each project could be viewed as an opportunity to expand this knowledge repository. Furthermore, a knowledge repository would provide robust guidelines to drive future customization decisions. Especially in combination of our method, this preserved knowledge could help the engineers to decide which method should be included in a script repository and which should be part of the customization layer for example. This knowledge-base could also be expanded by asking people who worked with similar interfaces about their past experiences, worst cases they had and how they came up with certain solutions.
Finally, another idea would be to develop some sort of interactive questionnaire. The purpose of this tool is to support the requirements elicitation for interfaces in a structured way, by suggesting the appropriate questions to ask their customers. It could guide the identification of important information such as the different use cases and their frequency of change. The initial content of this tool could be based on the findings of this study and by performing some interviews about the general desired requirements information and their priority. It could be extended with more questions based on the experience gained from interfacing projects.

Bibliography

[1] Svahnberg, van Gurp, Bosch. (2001) "On the Notion of Variability in Software Product Lines," Proc. Working IEEE/IFIP Conf. Software Architecture, pp. 45-54.

[2] Bosch, Capilla, Chul Kang. (2013) Systems and Software Variability Management, Springer, New York. p. 3 – 4 Kitchenham, B., Pickard, L., and Pfleeger, S.L. (1995) Case studies for method and tool evaluation. IEEE Software, 4(12), 52–62.

[3] Rhein, Thümc, Schaefer, Liebig, Apel. (2015), "Variability Encoding: From Compile-Time to Load-Time Variability", Preprint submitted to Elsevier, Germany

[4] Svahnberg, van Gurp, Bosch. (2005), 'A taxonomy of variability realization techniques' in Software-Practice And Experience , Wiley InterScience, Sweden, p. 705-754

[5] S.M. Davis. (1987), "Future Perfect", Addison-Wesley, Boston, Massachusetts

[6] Au tili, Inverardi, Mignosi, Spalazzese Tivoli. (2015), "Automated Synthesis of Application layer Connectors from Automata-based Specifications", in Language and Automata Theory and Applications, Springer International Publishing, Switzerland p. 3–24

[7] Rowely. (1995) 'Understanding Software Interoperability in a Technology-Supported System of Education', United States Air Force, p. 20 – 26

[8] Schmid & John. (2003) A customizable approach to full lifecycle variability management, Fraunhofer Institute for Experimental Software Engineering (IESE), Germany, p. 259 - 282

[9] Vaishnavi, V. and Kuechler, W. (2004). 'Design Science Research in Information Systems', January 20, 2004; last updated: November 15, 2015. URL: http://www.desrist.org/design-research-in-information-systems/

[10] Pohl, Böckle, van der Linden. (2005) 'Software product line engineering: foundations, principles, and techniques', Springer, Berlin Heidelberg

[11] Svahnberg, van Gurp, Bosch. (2005), 'A taxonomy of variability realization techniques' in Software-Practice And Experience , Wiley InterScience, Sweden, p. 705-754

[12] Bachmann & Bass. (2001). 'Managing Variability in Software Architectures' in ACM SIGSOFT Software Engineering Notes, ACM, USA, p. 126-132

[13] Anastasopoulos & Gacek. (2001), 'Implementing Product Line Variabilities' in ACM SIGSOFT Software Engineering Notes, ACM, Toronto, Ontario, Canada, p. 109-117

[14] Lee & Hwang. (2014), 'A review on variability mechanisms for product lines' in International Journal of Advanced Media and Communication, Inderscience Publishers Ltd, Daejeon, Korea, p. 172-181
[15] Galster, Weyns, Tofan, Michalik, Avgeriou. (2014), 'Variability in Software Systems—A Systematic Literature Review' in IEEE Transactions on Software Engineering, IEEE, p. 282-306
[16] Gamma, Helm, Johnson, Vlissides.(1995), 'Design Patterns: Elements of Reusable Object-Oriented Software' in Addison-Wesley professional computing series, Addison-Wesley
[17] Freeman, Sierra & Bates. (2004), 'Head First Design Patterns', O'Reilly, Available here
[18] Mirakhorli, Mäder, Cleland-Huang. (2012), 'Variability Points and Design Pattern Usage in Architectural Tactics' in SIGSOFT 20th International Symposium on the foundations of software engineering, ACM, p. 1-11
[19] Di Marco, Inverardi, Spalazzese. (2013), 'Synthesizing Self-Adaptive Connectors Meeting Functional and Performance Concerns' in International Symposium on software engineering for adaptive and self-managing systems, IEEE Press, p. 133-142
[20] Ke & Huang. (2012), 'Self-adaptive semantic web service matching method' in Knowledge-Based Systems, Elsevier B.V, Nanjing, Jiangsu, China, p. 41-48
[21] Van den Heuvel, Weigand, Hiel. (2007), 'Configurable Adapters: The Substrate of Self-adaptive Web Services' in Proceedings of the ninth international conference on electronic commerce, ACM, Minnesota, USA, p. 127-134
[22] Wölfl, Siegmund, Apel, Kosch, Krautlager, Weber-Urbina. (2015), 'Generating Qualifiable Avionics Software: An Experience Report' in International Conference on Automated Software Engineering (ASE), IEEE, p. 726-736
[23] Benavides & Galindo. 2014 'Variability management in an unaware software product line company: an experience report' in Proceedings of the Eighth International Workshop on variability modelling of software-intensive systems, ACM, p1-6
[24] Bass, Clements, Kazman. (2012). 'Software Architecture in Practice' in SEI series in software engineering, Addison-Wesley, 3rd edition
[25] Jahangir (1990). 'An Asset-Based Systems Development Approach to Software Reusability', The Society for Information Management and The Management Information Systems Research Center of the University of Minnesota, p.179-198
[26] International Air Transport Association. 2011, Standard Schedules Information Manual, Issued Montreal — Geneva Ref. No: 9179-21
[27] Wohlin, Runeson, Höst, Ohlsson, Regnell, Wesslén, (2000), Experimentation in Software Engineering, Springer, Berlin, Germany p. 66 – 73
[28] Griss. (2000), 'implementing Product line Features with Component Reuse', in Proceedings of 6th International Conference on Software Reuse, Vienna, Austria, June 2000
[29] Völter. (2009) 'Handling variability' in the CEUR archive of conference proceedings and for Hillside Europe website, Available here
[30] Linden, Schmid, Rommes. 2007, Software product lines in action, Springer Berlin Heidelberg New York

[31] IEEE 100. (2000) The Authoritative Dictionary of IEEE Standard Terms, 7th Edition, Standards Information Network IEEE Press, Available here
[32] International Organization for Standarization. (2001) ISO/IEC Standard 9126: Software Engineering – Product Quality, part 1, Switzerland
[33] Sommerville, 2011, "Software Engineering 9th edition", Addison-Wesley
[34] J. R. Mckee. 1984, “Maintenance as a Function of Design”. in Proceedings AFIPS, National Computer Conference, Las Vegas, pp 187-93
[35] Asadi and Rashidi. 2016, "A Model for Object-Oriented Software Maintainability Measurement" in I.J. Intelligent Systems and Applications, Modern Education and Computer Science Press
[36] Garlan, Allen and Ockerbloom. 1995 Architectural Mismatch: Why reuse is so hard, in IEEE Software pp 17-26
[37] Soren. 2002 "Software Requirements Styles and Techniques" 1st Edition, Addison-Wesley
[38] Mehmood & Jawawi, 2013 "Aspect-oriented model-driven code generation: A systematic mapping study" in Elsevier Science Ltd p395 - 411
[39] Jörges. 2013, "Construction and evolution of code generators: a model-driven and service-oriented approach" in Lecture notes in computer science, Springer
[40] Software Engineering Institute. 2000, "Capability Maturity Model Integration", Version 1.1 CMMI for Systems Engineering and Software Engineering Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University
[41] Boeing.com. (2016). Boeing: The Boeing Company. [online] Available at: http://www.boeing.com/ [Accessed 7 Sep. 2016]
[42] ww1.jeppesen.com. (2016). Jeppesen – Transforming the Way the World Moves. [online] Available at: http://ww1.jeppesen.com/index.jsp [Accessed 7 Sep. 2016]
[43] Renault and PLM Industry Expert 2014 "Reuse Variability Management and System Engineering" Poster Workshop of the Complex Systems Design & Management Conference CSD&M 2014.
[44] Galster & Avgeriou (2015), ’An industrial case study on variability handling in large enterprise software systems’ in INFORMATION AND SOFTWARE TECHNOLOGY, pp 16-31.
[45] Jansen, Slinger, Geert-Jan Houben, and Sjaak Brinkkemper. 2010 "Customization realization in multi-tenant web applications: Case studies from the library sector." International Conference on Web Engineering. Springer Berlin Heidelberg

A

Interview and evaluation questions

A.1 Semi-structured interview questions

1. What is the translation process of the SSIM to semantics that your company understands?
2. In your experience, can you give us examples where a field inside the SSIM was used in a different way from customer to customer? That is, customers had their own interpretation of that field? And why does this happen?
3. What about with misinterpretations of the semantics, what could go wrong? And how often can it occur?
4. What would you ask your customers to elicit their use cases?
5. Tell us, according to your experiences, about parts of the IATA that could be misinterpreted, that could maybe be handled differently from different customers?
6. How did you manage these issues in the past?
7. How are you handling things right now? Do you think there are issues with the current approach?

A.2 Group interview questions

For each of the following variation points we found during our analysis phase:

1. Do you agree that they are indeed a variation point?
2. Is there something else that we might have missed?
3. What do you think is the most urgent to deal with?

A.3 Code analysis questions

1. Can you guide us through the SSIM integrator's code of the old system?
2. Can you talk a little about the special cases the integrator is handling?
3. Can you elaborate more about special cases of your customers?
4. Do you have other issues with the interpretations of the semantics?
5. How do you detect these misinterpretations? and how do you handle them?

A.4 First evaluation questions

1. The functions that you wrote for your current customer in the new system, do you think they can be reused by some other airline?
2. Do you think, in general, that certain use cases might appear to other customers?
3. (after presenting our suggested method) Do you think this approach is good or bad? Why?
4. Are there some other ways that you could suggest for handling customer specific use cases?
5. Do you think this method could be replicated to other contexts as well?
6. Would you consider integrate this?

A.5 Final evaluation questions

1. What do you think of this approach? And Why? Do you agree with the preliminary feedback?
2. Do you think use recurring customer specific use cases could emerge for the SSIM interface? What about other interfaces?
3. What guarantee you have that your customers will write quality code and that their systems will be compatible with yours?
4. How do your customers understand the system's core and how do they test it?
5. Do you agree with these guidelines? And Why? Is there something missing?
6. (after discussing variability management for the Operational Messages interface) Do you agree with this approach? And Why?
7. What do you think about possible future work? What other ways could support the understanding and effective management of variability?