CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Using software product line engineering to construct products with different certification levels

- An Industrial Action Research Study

Master's thesis in the program Software Engineering and Technology

Oscar Evertsson
Rebecka Reitmaier

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

Master's thesis 2019

Using software product line engineering to construct products with different certification levels

An Industrial Action Research Study

Oscar Evertsson
Rebecka Reitmaier

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

Using software product line engineering to construct products with different certification levels

Oscar Evertsson Rebecka Reitmaier

Supervisor: Jan-Philipp Steghöfer, Computer Science and Engineering
Advisor: Robert Engberg, 1928 Diagnostics
Examiner: Robert Feldt, Computer Science and Engineering

Master's Thesis 2019
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Gothenburg, Sweden 2019

Oscar Evertsson and Rebecka Reitmaier
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Background: Software product line engineering (SPLE) is used to derive multiple products from a common platform and has many industry examples of its benefits such as reduced development cost and decreased time to market. However, there is no research to our knowledge on how well it functions with agile development when there is a need to create both safety-critical products and non safety-critical products. The regulatory difference between these two is that the safety-critical products require certification to be sold.

Aim: This thesis investigates which SPLE variability approaches can be used to differentiate code associated to a safety-critical and a non safety-critical product. The research was done at the company 1928 Diagnostics, with the goal of finding the most suitable variability approach for the company and how this might affect their business, architecture, process and organization.

Method: We investigate variability approaches by looking at current SPLE literature and by using an action research methodology. The data is collected through interviews, focus groups, a mockup of a chosen variability approach and discussions at the company.

Results: We identify five variability approaches that can support differentiation: *design patterns*, *components*, *preprocessor*, *parameter-based* and *version control*. We found that the most suitable variability approach for 1928 Diagnostics was components. From the evaluation of the mockup we found that potential effects primarily would be related to the architecture and the assistance it could provide to the process later. Finally, we present a methodology for how to derive the most suitable variability approach.

Keywords: Software Product Line, Agile Development, Safety-critical system, Development Process, CE-marked products, Certification, Medical Device, Software Engineering.

Acknowledgements

We would like to start with thanking everyone involved in this project. In addition, we want to express our utmost gratitude to the following people for making this thesis possible.

Jan-Philipp Steghöfer, supervisor at Chalmers: For the guidance and academic expertise he has provided and for being more than anything you can expect from a supervisor.

Robert Engberg, QA Director and supervisor at 1928 Diagnostics: For the standard and certification expertise provided and availability for both questions and support.

Fredrik Dyrkell, CTO at 1928 Diagnostics: For the technical expertise provided and availability for questions.

The team at 1928 Diagnostics: For a warm welcome to your company, for letting us conduct this research and availability for questions.

Tobias Alldén: For reviewing and giving feedback.

Oscar Evertsson and Rebecka Reitmaier, Gothenburg, June 2019

Contents

Glossary

BAPO model a model which covers the four main concerns of software product line engineering: Business, Architecture, Process and Organisation (van der Linden, Schmid, & Rommes, 2007). 2

IEC 62304 a standard for medical device software which prescribes a set of processes, activities and tasks. 3

MedTech Medical Technology or Health Technology is the application of knowledge which can solve a health problem or improve quality of life (Organization, 2007). 1, 30

pathogen An organism or other agent that causes disease (Alberts et al., 2002). 1

safety-critical system a system where a potential fault in the system could lead to financial loss, damage to the environment or injury of people (Kasauli et al., 2018). 1

software product line engineering a way of working with software development where one wants to reuse code as much as possible to create an SPL. 2

software product line a collection of similar products which can be derived from a shared set of software assets. 2

technical documentation "The documented evidence, normally an output of the quality management system, which demonstrates conformity of a device to the Essential Principles of Safety and Performance of Medical Devices" (Force, 2011). 57

List of Figures

List of Tables

1

Introduction

In many industries today, the competition of being the first on the market with a product has never been more difficult. The world is more connected than ever with an estimated amount of connected devices to more than 75 billion by 2025 (Statista, 2019). This globalization means that companies who previously only had one local competitor may now have multiple competitors even if they are located time zones apart. Competition in the MedTech industry is no exception; 1928 Diagnostics[1] is a MedTech company which delivers a cloud platform that provides both infection tracing and diagnostics in a shared codebase.

The diagnostics the company provide can predict whether a *pathogen* is resistant to variants of antibiotics. This prediction can then be used to choose suitable antibiotics for patients. Infection tracing, on the other hand, is something that is currently only used for research purposes, to trace outbreaks both globally and at an individual hospital. Since the diagnostics provided by the company affect patient safety this part of the system is classified to be a *safety-critical system*. For a system to be considered safety-critical a potential fault in the system leads to financial loss, damage to the environment or injury of people (Kasauli et al., 2018). The last case, injury of people, could potentially be the case for 1928 Diagnostics if the system would erroneously diagnose a pathogen's antibiotic resistance.

Due to the nature of having parts of a system that is considered safety-critical, 1928 Diagnostics needs to follow specific regulations to be able to sell a medical device product. Depending on the risk-level of the product and the country where the product is placed, there are different avenues for achieving this. These avenues span from self-certification to thorough assessments by the governing body which ensures that a product follows the regulations.

To aid companies in complying with regulations, several standards have been created which are considered state of the art. These standards often require documentation for safety cases and a need to analyze the safety requirements beforehand (Kasauli et al., 2018). To certify a product against a standard, therefore, results in an extra workload which would not be present otherwise.

From a certification point of view you talk about certifying a medical device product rather than the code alone. To achieve certification, 1928 Diagnostics need to handle code that will be a part of such products according to specific processes. These

[1] https://1928diagnostics.com/

processes differ from processes that they can use for code that is part of none medical device products. This thesis uses the terms “certified” and “noncertified” to identify code that is either part or not part of a medical device product.

As previously mentioned, 1928 Diagnostics work with both diagnostics which requires certification for which those products and with infection tracing that does not require certification. These two services share a lot of features which are located in the same codebase, even though their use cases and need for certification are different.

1928 Diagnostics would like to expand their pool of pathogens they analyse into more products where some will be used for infection tracing, some for diagnostics or a combination of both. The infection tracing and diagnostics consist of some shared features but also include variations on a pathogen basis. Certain pathogens can for example require additional typing. A possible solution to derive multiple products from a set of shared features is to create a software product line (SPL), primarily since it promises a shorter time to market and reduced costs for developing the systems (Apel et al., 2013).

One way to construct an SPL is to apply software product line engineering (SPLE). SPLE is a way of working with software development where one wants to reuse code as much as possible to reduce the development effort and thereby time-to-market (Vaccare Braga et al., 2012). SPLE affects large parts of a company, and there are four main concerns one needs to address: the business, architecture, process and organization aspects. These four concerns are together referred to as the BAPO model, see Figure 1.1 (van der Linden et al., 2007).

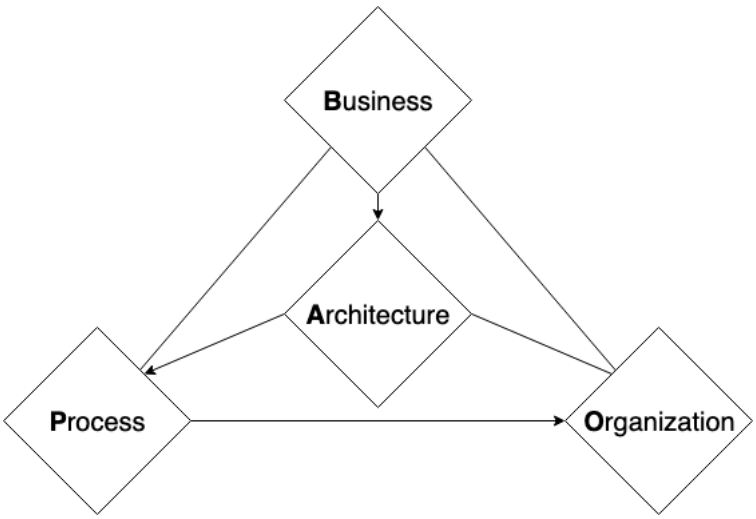

Figure 1.1: The BAPO model. The diagram is adapted from van der Linden et al. (2007)

SPLE does not take into account that the shared functionality might have different requirements with regards to certification. According to Vaccare Braga et al. (2012) it is important to know which parts of the codebase need to have a specific certification level; This so one can avoid unnecessary work for those parts which do not

require certification activities.

1928 Diagnostics develops its cloud platform following an agile methodology. In the Manifesto for Agile Software Development, the authors state that when working in an agile way the practitioner should focus on individuals and interactions over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation and responding to change over following a plan (Beck et al., 2001). The standard IEC 62304 is recognized as *state of the art* and commonly chosen when developing medical device software, as such it is also used by 1928 Diagnostics as part of fulfilling regulatory requirements for certification. This standard has requirements on the *software life cycle processes* which on the surface seems more to be in line with working in a more waterfall compared to their preferred agile methodology.

This thesis brings the areas safety-critical, agile and software product lines, certification together and investigates how they are interconnected.

1.1 Statement of the Problem

1928 Diagnostics have four needs that they would like to solve to be as cost-effective and competitive as possible. These are:

1. Segregate the codebase in a way that allows only relevant parts of the code to be developed according to stricter requirements for medical devices.
2. To achieve certification they need to comply with necessary safety standards. These standards require artifacts whose generation 1928 diagnostics would like to automate as much as possible.
3. Derive multiple products from a shared codebase.
4. Work agile, to be able to deliver new features quickly and faster adapt to changes.

To be able to segregate and certify only parts of the codebase is something that 1928 Diagnostics believes could minimize the amount of time spent on procedures the standards prescribe, giving them a competitive advantage. For example, if the code is excluded from something that 1928 Diagnostics wants to certify it would not need to be included in the risk management which is a requirement for one of the standard *IEC 62304*.

To gain a competitive advantage 1928 Diagnostics would like to automate and generate as much as possible of necessary artifacts. Besides, they want to derive multiple products from a shared codebase and to work agile. An SPL could be beneficial for the company to derive multiple products. However, there are multiple SPLE variability approaches available and from the reviewed literature it is uncertain if they could be implemented to assist in certifying only parts of the codebase. It is also unclear if these variability approaches can be combined with automation and generation of artifacts, working agile and in a safety-critical context. 1928 Diag-

nostics are interested in the potential benefits of moving to an SPL but because of these uncertainties and how it might affect BAPO, they are hesitant.

1.2 Purpose of the Study

On a general perspective, the purpose of the study was to investigate what SPLE variability approaches could be used for differentiating between certified and noncertified code. Furthermore, from a 1928 Diagnostic specific perspective, the purpose was to investigate what variability approach is most suitable to solve their problems, see section 1.1, and how this might affect their BAPO. This thesis pushes the intersection of the three fields *agile software processes*, *safety-critical systems*, and *SPLE* forward by how the interaction might affect BAPO. To the best of our knowledge, there are no existing studies that investigate introducing SPLE in an agile environment for safety-critical systems.

1.3 Research Questions

Based on the background, the problem description and the purpose of the thesis, the following research questions were constructed:

RQ 1: Which variability approaches for implementing an SPL can be used for differentiating between certified and noncertified code?

RQ 2: In the context of 1928 Diagnostics, what is the most suitable way to derive multiple products from a shared codebase with different certification levels?

RQ 3: How does the chosen variability approach affect the organization's BAPO?

RQ 4: Can a methodology be constructed to derive the most suitable variability approach for differentiation between certification levels? If so, how?

1.4 Contribution

One of the broader contributions of this thesis is which variability approaches support differentiation between certified and noncertified code. This can be an aid for any company that would like to move towards an SPL and have a need to differentiate code. Further, the thesis presents a set of quality attributes for each variability approach containing both attributes coming from literature and through this research. An example of a quality attribute from literature is *separation of concern*. Another one, which comes from this research is *Transition in steps possible*.

The most significant contribution of this thesis is the methodology described in

Chapter 9 which presents a workflow to derive the most suitable variability approach to differentiate between certified and noncertified code. The methodology is based on the results in the other previous chapters. This methodology could be useful for companies not only in the MedTech industry but also for other industries where safety-critical products are to be constructed using an SPL.

A more narrow contribution by the thesis is which variability approach was most suitable for 1928 Diagnostics and why. Furthermore, the thesis presents the potential effects that the implementation of the variability approach could have on their BAPO. This could be relevant not only to 1928 Diagnostics but also to companies in a similar domain or setting which would like to move towards an SPL.

The thesis further shows that the implementation of the chosen variability approach could be extended through a tool to assist developers in their development process. This could be helpful for other companies that chose to implement components as a variability approach.

2

Background

This chapter provides an explanation of topics which are needed to understand the thesis. These topics include what a software product line is and how it can be achieved through variability management approaches. The chapter also introduces general knowledge about certification for medical device software and how 1928 Diagnostics' services work.

2.1 Software Product Line Engineering

A software product line is a collection of similar products which can be derived from a shared set of software assets (Apel et al., 2013). Software product lines emerged from a need to tailor mass-produced products for different customers (Apel et al., 2013). The differences between the customized products then need to be managed in some way, this is called variability management.

Variability management is the core of software product line engineering. One needs to find, model and implement the similarities and differences between the products (van der Linden et al., 2007). To effectively use these similarities between products is it important to develop assets for reuse and then use these reusable assets when creating new products (van der Linden et al., 2007).

There are many positive effects of moving towards SPLE if one knows that they need to develop several products which share some features. The main reason most practitioners use SPLE is that it reduces the development cost of systems (van der Linden et al., 2007). Other positive effects are the reduction of time to market and enhancement of the system's quality (van der Linden et al., 2007). These positive effects exist because there is less need to create and maintain duplicated code between several systems which share features.

SPLE is divided into two parts: development *for reuse* and development *with reuse* (van der Linden et al., 2007). Development for reuse is called *domain engineering*. The idea of domain engineering is to develop reusable assets that provide the necessary range of variability for all wanted products. When that is complete the development with reuse take place which is called application engineering. The application engineering focuses on the development of the individual systems on the existing platform provided by the domain engineering. Both domain and appli-

cation engineering include development processes as requirements analysis, design, implementation, and testing. An outcome of the domain design phase is a reference architecture (van der Linden et al., 2007). The reference architecture is important since different components have to be able to communicate with each other through generic interfaces in all product variants (van der Linden et al., 2007). A reference architecture that captures similarities between products can ensure that components are created efficiently. By doing so, components which work in a similar manner can be brought together to one instead of having two separate components (van der Linden et al., 2007).

To be able to create an SPL the codebase does have to have some degree of variability (Apel et al., 2013). That code has variability results in that there is a possibility of creating different end-products from it. Apel et al. (2013) describe that there are three ways of characterizing variability in a system: binding time, language-based versus tool-based, and annotation versus composition. These three ways will be presented in the following sections individually.

2.1.1 Binding Time

Apel et al. (2013) describe the concept of binding time. Binding time referers to the point in time when features are included in a product variant, one *binds a decision*(feature). They present three different binding times: compile-time binding, load-time binding, and run-time binding. The difference between these is when the features are selected.

Compile-time binding is when features are selected before developers compile the program. Features which are not selected are not compiled and because of this is this alternative a good option if optimization is a wanted outcome. Since the features have to be selected before compilation it is the developers who choose the features for the product variant which gives a very limited user-flexibility. Two approaches for implementing an SPL with compile-time binding are to use preprocessors or feature-oriented programming.

Load-time binding is when features are selected at the program start. With this binding time option users are able to choose what features they want to use. Load-time binding comes at a cost since all features have to be compiled and shipped to the end-user. Load-time binding therefore often comes at a cost of memory and performance.

Run-time binding is similar to load-time binding since the features are selected after compile-time but features can be changed when the program is running. Since features can be changed during run-time these types of programs change features depending on direct input from users. Since run-time binding also compiles all features it does have the same negative effects as load-time binding: memory and performance. Two ways of implementing load-time and run-time binding are to use parameter-based variability or context-oriented programming.

2.1.2 Language-Based versus Tool-Based

Language-based approaches are implementation techniques that use mechanisms provided by the programming language while *tool-based approaches* are those that derive products based on tools which operate on software artifacts. The benefits of language-based approaches are that both the implementation of features and variability management are located in the source code; This makes it easier for developers to understand and reason about the product line and its implementation. The use of language-based approaches is however not only positive. Depending on the implementation approach, feature boundaries and keeping track of what features exist in the codebase can be hard (Apel et al., 2013).

Tool-based approaches prefer a clear separation between the implementation of features and variability management to derive products. The thought of doing so is to achieve a better code structure but forces the developer to have knowledge of more artifacts and find them (Apel et al., 2013).

2.1.3 Annotation versus Composition

Two ways that are widely used in practice to construct a product line are *annotation-based approaches* and *language-based composition approaches*. For annotation-based approaches, the code is marked to which feature it belongs to. This enables annotation-based approaches to remove the deselected features at either compile-time or ignore them at run-time to finalize the product. The benefits of using annotation-based approaches are that they are easy to use and is often supported in programming languages. However, there are not only positive aspects of annotation-based approaches. Preprocessor-based and parameter-based implementations for annotation-based approaches are often criticized for potential complexity, lack of modularity and reduced readability (Apel et al., 2013).

Language-based composition approaches on the other hand structure features in composable units such as a file, a container or a module. Ideally, each unit only consists of one feature to make it easier to include in a final product. An example is a framework that can be extended with plugins where each plugin, in this case, is a file, a container or a module. The main issues with language-based composition approaches are to keep the mapping between features and units easily understandable, traceable and to keep the relationship between units and features one-to-one (Apel et al., 2013).

2.2 Variability Management Approaches

According to van der Linden et al. (2007) there are three general techniques to gain variability:

- *Adaption* - With this technique there is only one implementation available but

the implementation provides interfaces which can be used to adjust behaviour. The authors mentions three examples of adaption: a configuration file, runtime parameterisation and patches of source code.

- *Replacement* - With this technique there are several implementations available which all comply with the requirements from the specification. When a product is to be made one can choose from these available implementations which yield different results.
- *Extension* - With this technique the architecture does need to supply interfaces which plugins can attach to. With different combinations of plugins can one create different products.

There are several available implementation techniques for variability. In the below sections have the variability techniques from the three books *Feature-Oriented Software Product Lines Concepts and Implementation* (Apel et al., 2013), *Software Product Lines in Action* (van der Linden et al., 2007) and *Software product line engineering: foundations, principles and techniques* (Pohl, Böckle, & van Der Linden, 2005) been described briefly. The names of the techniques differ to some extent in the books, but the thought behind the techniques are the same.

Some techniques mentioned in these books could be considered as good software development practices such as parameters, version control, design patterns and components. However, when these can be used in a specific fashion to create multiple products from shared software assets, we consider then as SPL variability approaches. We have not found any literature to define the difference between good software development practices and SPL variability approaches that can be used to reuse code to a large extent.

2.2.1 Parameters - Configuration

The parameter approach can be used to alter the flow of a program with the use of parameters which also is called configuration (Apel et al., 2013). This control of the program flow could then be handled using conditional statements (Apel et al., 2013). With this approach the implementation has different variations internally. The outcome of the component depends on the input to its interface (van der Linden et al., 2007).

A strong point of this technique is that it is easy to use and that it is very flexible (Apel et al., 2013). These two properties result in that the technique is often used in an ad hoc manner and that the configuration becomes scattered over the codebase and that the separation of concerns is low (Apel et al., 2013).

2.2.2 Design Patterns

One downside of the parameter based approach was that the variability could become scattered if not used in a good way. Design patterns are a way of gathering

functionality in a certain place (Apel et al., 2013). There are many design patterns but Apel et al. (2013) mention four specifically: *Observer pattern*, *Template-Method pattern*, *Strategy pattern* and *Decorator pattern*. Design patterns are also often combined or altered to fit the current needs (Apel et al., 2013). Two good aspects of design patterns are that they are very well established and that they yield a good separation of concerns. A downside could be the boilerplate code and the architectural overhead (Apel et al., 2013).

2.2.3 Framework

Another alternative for implementing variability is a framework. A framework consists of an architecture which has well-defined interfaces which can be extended with plugins (van der Linden et al., 2007). The framework could then adopt plugins to construct different products (Apel et al., 2013). The plugins could be either for all products or specially made for a certain product (van der Linden et al., 2007).

van der Linden et al. (2007) mention two types of frameworks: *White-Box* and *Black-Box* frameworks. White-Box frameworks have concrete abstract classes which is visible for developers looking to construct variability through child-classes. Black-Box on the other hand separate framework code and extensions through interfaces. In a Black-Box framework objects and callback functions are connected through hotspots. The naming behind Black-Box framework is that developers should not have to understand the underlying implementation of the framework but merely their interface to the opposing White-Box were a class is extended and implementation is visible to the developer.

2.2.4 Components

Apel et al. (2013) describe that components can be used for variability since they can be used by several other parts of a system or even external separate systems. They further describe that components have all their implementation encapsulated and that it is accessed by clearly defined interfaces. van der Linden et al. (2007) describe that components can achieve variability since they can be replaced by other components with the same default implementation. Further, they mention that product derivation cannot be performed automatically but instead require gluecode. Another issue mentioned is that if a component require replacement they need to have the exact same interface to be compatible for change.

2.2.5 Version Control

Version control in its general form used to aid developers in tracking changes done to source code and to support collaboration during development (Apel et al., 2013). *Branching* is something that most version control systems support and allows a developer to have multiple copies of the same file independently according to Apel

et al. (2013). The authors present two ways to construct product lines using version control: customer-specific variation and merging per-feature branches.

Customer-specific variation is a technique where there exists a baseline product that a customer want. This product exists in its own branch for this specific customer which gives the developers the possibility to tailor and make changes to the software that the customer requires. This leads to multiple variants of the software where the number of variants corresponds to the amount of customers.

The other alternative *merging per-feature branches* construct multiple products by merging features branches into some baseline. An example described by the authors is to have a base product on one branch. In addition to the base branch there are two feature branches: *colored* and *weighted*. Depending on what a customer want a developer could merge the baseline with a feature branch to construct a desired product. For example the baseline product with the feature of colored. But there is also the possibility to have the baseline together with the feature weighted or to have both features together with the baseline.

Like many others of the variability methods, there exist several different alternatives for using version control such as: git[1], svn[2] and Mercurial[3].

2.2.6 Build Systems

A build system is a tool responsible for all build-related actions such as running generators, compiling source code and running tests. Build systems can be simple shell script or something more advanced such as make, ant or maven. Since build systems have control over the compilation of a program, they can be used to manage the variability through a configuration file. A build system could, for example, be used to select between two components which have the same interface but different implementation using configuration (Apel et al., 2013).

2.2.7 Preprocessor

"A preprocessor is a tool that manipulates source code before compilation." (Apel et al., 2013). A commonly used preprocessor to achieve variability is *cpp* which was developed to enhance the C-language (Liebig et al., 2010). Even though it was developed for the C-language, the cpp preprocessor can also be used for other languages such as Java, C# or even artifacts that are text-based since cpp is line based (Liebig et al., 2010). Cpp supports several ways to achieve variability such as using the following keywords: *#if*, *#else*, *#ifdef* and *#ifndef* and *#endif* (Kernighan, 1988, p. 91-92).

Apel et al. (2013) presents an example of preprocessor usage to achieve variability

[1]https://git-scm.com/
[2]https://subversion.apache.org/
[3]https://www.mercurial-scm.org/

in the form of a graph. The graph can be extended with color and weighted functionality if those variables are defined using *#ifdef*. These variables can then be defined through a macro to construct a specific product that includes both colored and weighted functionality of the graph.

2.2.8 Feature-Oriented Programming

Feature-oriented programming is a composition-based approach to achieve variability by using features for separation. To be able to express which parts of a program support a certain feature and to enable encapsulation of composable and modular units, new language constructs are needed (Apel et al., 2013).

Apel et al. (2013) describe an example of implementing multiple versions of a graph using feature-oriented programming. The first version of a graph consists of three classes: graph, edge, and node. More variants of a graph implementation can be extended by adding roles to all of these, for example, "weighted". The weighted role could then be added to the graph, edge, and node individually. Furthermore, another role could be added such as "colored". An adapted figure from Apel et al. (2013) can be seen below in Fig 2.1.

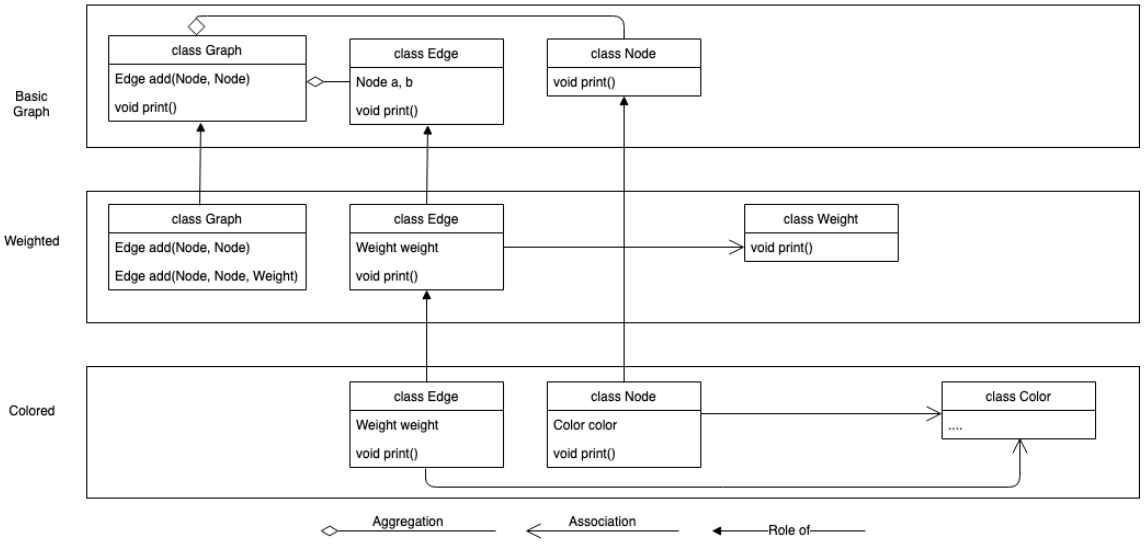

Figure 2.1: Example usage of feature-oriented programming for variability, adapted from Apel et al. (2013)

By doing so four potential variants can be constructed:

1. A graph default implementation
2. A graph with the additional functionality of weighted
3. A graph with the additional functionality of colored
4. A graph with both the functionality of colored and weighted.

2.2.9 Aspect-Oriented Programming

Aspect-oriented programming was first described as a concept by Kiczales et al. (1997). They found many problems which can not be sufficiently solved using procedural or object-oriented programming. With previous approaches, the result would be "tangled" code that is hard to develop and maintain. These problems arise with functionality that "cross-cut" the systems basic functionality. The properties of the design decisions to solve this problem is called "aspects".

Apel et al. (2013) summarizes the power of aspects neatly by saying: "It enables code that is associated with one crosscutting concern to be localized into one code unit, thereby eliminating code scattering and tangling. Moreover, aspects can affect multiple other concerns with one piece of code, thereby avoiding code replication.".

Hunt (2006) presents an example of where aspect-oriented programming can come to use. Assume that a large system would like to log information. One way to achieve this is to implement a logging subsystem. To make use of this logging you as a developer would have to go through the rest of the system and add log statements within methods where you find it appropriate. The result is that potentially pure business logic are now bloated by logging statements which might make it hard to read or even worse, hard to understand. Logging is an example of a crosscutting concern, an aspect that cuts across many other modules. Aspect-oriented programming can create a self-contained module for logging that is dynamically linked with the business logic in a way that the business logic can remain unchanged.

In the context of using it for software product lines Apel et al. (2013) describe that the most straightforward alternative is to implement one aspect per feature. To include the features a customer wants a build system could be used to include features at joint points. A joint point the authors defines as: "A join point is an event in the execution of a program at which aspects can be woven into the program. The source code locations that give rise to a join point are called its join-point shadows."

2.3 Certification for Medical Device Software

The standard IEC 62304 is one of the standards that 1928 Diagnostics follow to comply with the medical device regulation on their cloud platform. IEC 62304 sets the standard on how companies or organizations should work with medical device software. The standard is a framework with life cycle processes together with connected activities. These activities are there to secure a safe design and maintenance of the medical device software.

In IEC 62304 there are three *software safety classifications*: A, B, and C. Depending on which classification the software have are there connected activities to be done. Classification A is the one with the smallest amount of activities and C is the classification with the most amount of activities.

- Class A: "no injury or damage to health is possible"

- Class B: “non serious injury is possible”
- Class C: “death or serious injury is possible”

The standard also describes that for systems with class B or C, the manufacturer needs to subdivide the software into so-called *software items*. For those with classification C, the design documentation and interface documentation for these software items should be with enough detail so that a developer can implement it correctly. There are many activities in this standard but examples of activities are risk management and testing.

As previously mentioned, 1928 Diagnostics has a cloud platform where some parts which are defined as medical device software. Since only some parts of the codebase are affected by the IEC 62304 processes, activities and tasks 1928 Diagnostics need to differentiate the code which needs certification. By differentiating the developers and compliance managers know which processes need to be followed.

2.4 Platform Pathogens

1928 Diagnostics is a company which delivers two services: infection tracing, and diagnostics to whether a pathogen is resistant to variants of antibiotics. The infection tracing service currently supports five pathogens including *staphylococcus aureus*. With infection tracing a hospital can, for example, see how an outbreak has evolved from patient to patient or between hospital employees. 1928 Diagnostics also have another service which is used for diagnosing whether a pathogen is resistant to certain antibiotics. This service is a medical device since it can be used in clinical decisions regarding treatment of individual patients. The service currently has one pathogen which is *staphylococcus aureus*. In the codebase, each pathogen has its own so-called *pipeline* which is used for analyzing the pathogen. These pathogen pipelines share some analysis functionalities between them but also include variability in what analysis is appropriate. These features are currently in separate files which are mainly divided by functionality.

3

Related Work

From the reviewed literature we looked at, there were no papers or studies that focus on how SPL variability in a safety-critical and agile context impacts the BAPO. However, there is literature describing some combinations of the areas which are applicable to this thesis.

3.1 Agile Development in Safety-Critical Systems

Since this thesis is conducted in the safety-critical domain and in an agile software development process the relationship between these two has been reviewed. The use of the agile process in safety-critical systems has been limited in the industry. This is mainly because of the need to document and analyze the safety-critical requirements conflict with the agile ways of working with documentation (Kasauli et al., 2018).

In the research area, *agile with safety requirements* the industry especially sees a need to investigate the infrastructure around these two sometimes conflicting ways of working (Kasauli et al., 2018). This need is a good indicator that the thesis is not only relevant for 1928 Diagnostics, but also that other industries could benefit from the research findings. The primary reason why agile development and development of safety-critical systems can be considered problematic is that of the tension between the Agile Manifesto (Beck et al., 2001) which states the following:

1. "Individuals and interactions over processes and tools"
2. "Working software over comprehensive documentation"
3. "Customer collaboration over contract negotiation"
4. "Responding to change over following a plan"

Compared to safety-critical systems where documentation and following a plan is necessary to assure safety. However, as the definition states, it is "x over y" not that they can't be combined. The insight that there might be a tension between the agile methodology and the extra work needed for the safety-critical software has been taken into consideration when reviewing the effects on the BAPO at 1928 Diagnostics.

Abdelaziz, El-Tahir, and Osman (2015) have identified a need for a method where one can see if the use of agile methods results in a software system which is safe to use. The outcome of the paper is an approach which manages both the safety process for the software and the process regarding the requirements. This paper focuses on the effects on the process of working with a system. The paper does not mention how the results could affect the business, organizational or architectural view which is researched in this thesis.

Cawley, Wang, and Richardson (2010) mention that the medical regulations do not impose a specific software development methodology, the important thing is instead that the mentioned activities are correctly done. Their initial idea was to do a Systematic Literature Review (SLR) about agile development in the medical device industry, however they did found that there was not enough material there. Thus they broadened the SLR to be more general and then instead did the SLR about regulated safety-critical embedded software development. The lack of literature in the area is a strong indicator that there is a need to investigate how one can work with agile together with safety-critical software in the medical device domain which this partly focus on.

3.2 Agile Development in Software Product Lines

There exists research in the area of agile development within software product lines. Some call this research *APLE* which means *agile product line engineering* while others call it ASPL to refer to the same research (Díaz, Pérez, Alarcón, & Garbajosa, 2011). Díaz et al. (2011) have performed a systematic literature review of the experiences and practices of APLE. Their review has found challenges with integrating agile software development in software product line engineering, but also that there are sufficient reasons to move toward a combination of software product line engineering and agile software development. One identified issue from their review shows that there are identified issues with traceability when it comes to domain engineering when applying APLE (Díaz et al., 2011). Traceability is an important factor when developing safety-critical systems and is something that will need to be taken into account when working with the process and architecture in this thesis.

3.3 Software Product Lines for Safety-Critical Systems

Vaccare Braga et al. (2012) has conducted research in adopting SPL for Unmanned Aerial Vehicles (UAV) which similarly to 1928 Diagnostics requires different certification levels. They mention six issues with regards certification and SPL in the UAV domain:

- **Certification level of complex products:** A created product could be

quite complex; a component which has a certain certification level might not only contain sub-components with the same certification level. There is the case of when a sub-component is not in the critical path and therefore can not generate a case where someone can get hurt, then that sub-component does not need to be certified.

- **Usage context and the certification level:** Depending on who the user is, the certification level of the product can differ. The part of 1928 Diagnostics' system which can diagnose a pathogen's antibiotic resistance can be used by doctors who treat patients, but it can also be used by researchers who use it for research purposes only. For these two cases, it is only the user who treats patients who need a certified product even though it is the same product.
- **Additional features and certification levels:** Some features can be specially developed for a certain certification level. The reason for this, is because it is an extra feature requirement on a higher certification level. This special features, can, of course, be used in lower certification levels but the point is that it is needed for certain higher certification levels. This has not been identified as an issue when talking to 1928 Diagnostics. However, can this be good to know for the future.
- **Features can contribute (positive/negatively) to obtain a particular certification level:** Depending on the functionality of a feature it could provide assistance on hinder or retard certification. As an example, Vaccare Braga et al. (2012) mention a weather resistance feature that could contribute towards particular certification levels while a feature which is more cost effective might do the opposite. At 1928 Diagnostics there are some features which are needed for certification, for example quality control. Thus will this quality control feature contribute to comply with the standard.
- **Features associated with design decisions can impact certification:** A system could potentially be implemented with different designs, each design focusing on something special. Some designs could have more safety features and could, therefore, be considered helpful for certification, while other designs which are more cost-effective could be considered less safe. The example Vaccare Braga et al. (2012) mentions is that "a system architecture can have two alternative designs, with or without redundancy to improve reliability and this decision impacts certification".
- **The development process should be adapted according to the certification level:** In the issues above, it is the features which affect the certification level. However, the development process also affects the certification level one can achieve. For certain standards, there needs to be a specific set of tasks done within the development process. Because of this, different parts of the system can require different tasks to be done. One development process can for example include more rigorous testing activities and requirements analysis.

Vaccare Braga et al. (2012) has proposed an infrastructure for using a SPL when certification is required for some of the products. The paper focused on mapping

the architecture to the activities and artefacts that a standard requires. The mapping was done through a meta-model, there a product was connected to a certain certification level and each product had several features. Vaccare Braga et al. (2012) mentions that SPL products can be divided into smaller parts such as components and that SPL products are derived through SPL assets which are associated to features. One can because of this attach a certification level to each feature (component).

4

Methodology

This chapter presents the methodology that was used for answering the research questions for this thesis and threats to validity. First, the general research methodology is introduced followed by the steps done in this thesis. Lastly, the threats to validity are presented.

Action research is the general research methodology followed in the thesis; Stringer (2013) describes action research as *"[...] a systematic approach to investigation that enables people to find effective solutions to problems they confront in the everyday lives."* Stringer continues by making a distinction between action research and other methods such as experimental or quantitative research that look for generalizable explanations. Instead, action research looks at problems in specific situations and localized settings.

Action research is closely related to case study research where the difference is that case study research is purely observational while action research involves change (Runeson & Höst, 2009). This thesis looked at one case, 1928 Diagnostics in a specific situation to answer the research questions. Since one case is studied and observed it might seem apparent to use a case study research methodology. However, as earlier mentioned a case study is purely observational which would have made it hard to answer a research question involving evaluating a change (research question 3). Therefore, since this study required change and looked at problems in a specific situation and localized setting, action research was more suitable.

This thesis has been conducted in an iterative manner and five iterations have been finalized. The input of each iteration is the output of the previous iteration except for the first iteration where the literature was the main input. In each iteration, a topic is investigated and answered. All iterations can be seen in Figure 4.1.

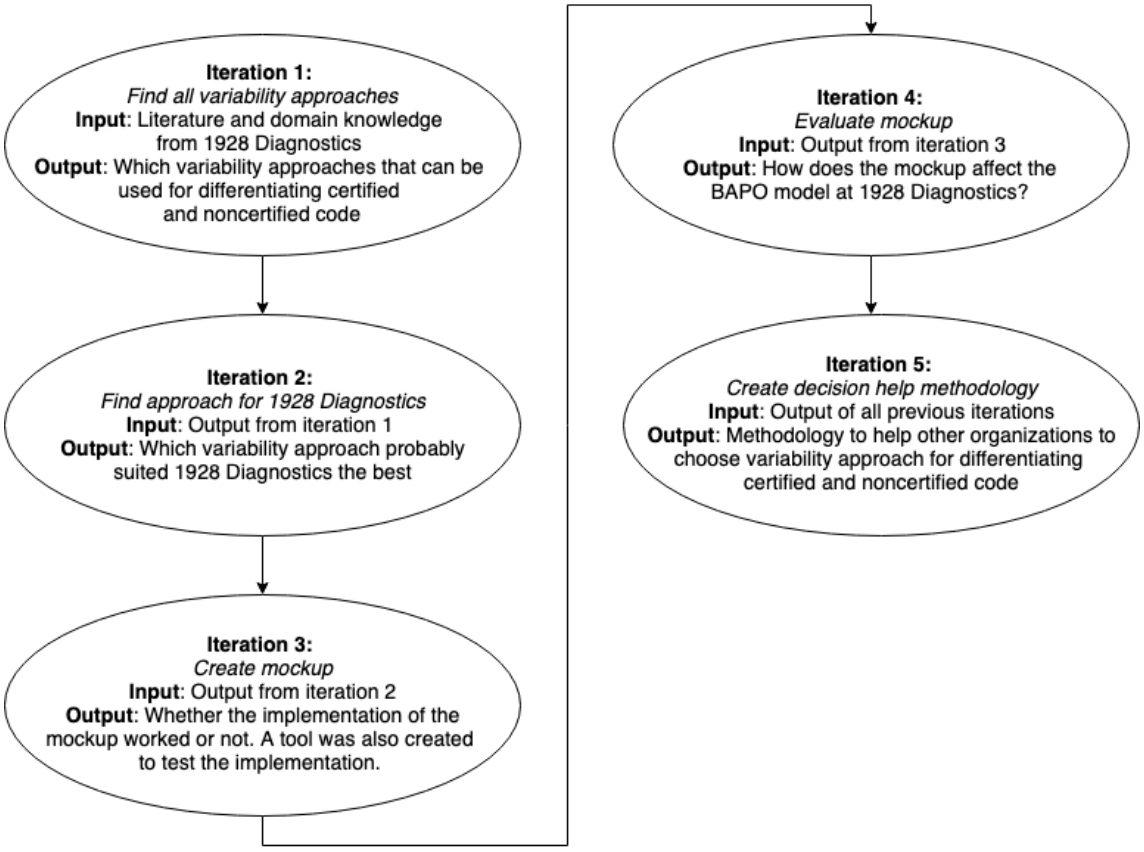

Figure 4.1: The iterations done during the thesis

4.1 Iteration 1: Finding variability approaches which can be used for differentiation

The focus for this iteration was understanding the thesis domain and answering the first research question "*Which variability approaches for implementing an SPL can be used for differentiating between certified and noncertified code?*".

4.1.1 Literature Review

To be able to answer RQ 1 the researchers wanted to find the available variability approaches. Followed by later narrow down to the variability approaches which could support differentiation between certified and noncertified code. To find which SPLE variability approaches that exist, both papers and books on the subject of variability within an SPL were reviewed. To find relevant literature the following keywords were used: *Software Product Line Engineering*, *Software Product Line Engineering Design*, *Software Product Line Engineering implementation*, *Software Product Line Engineering variability*, *Software Product Line*, *Software Product Line Design*, *Software Product Line implementation* and *Software Product Line variability*.

To better understand the literature on the connection between the agile methodology, safety-critical software and SPLE, the following keywords were combined: *agile*, *safety-critical*, *safety-critical software*, *certification*, *SPLE*, *variability approaches* and *variability management*. When interesting papers were found, their references were looked up and reviewed.

The three sources for finding relevant literature were Scopus[1], Google Scholar[2] and IEEE xplore [3]. In addition to these searches the supervisor of the thesis assisted in pointing towards related literature.

To understand how each variability approach could affect safety-critical systems in general was the standard *IEC 62304* reviewed. The standard was chosen as a data source since it gives an idea of what could be of interest for a company that wants to construct an SPL. It clearly targets what needs to be done to certify a product in accordance with the development process.

4.1.2 Unstructured Interviews to Investigate Gaps in Knowledge

According to Runeson and Höst (2009) it is important to use several data sources to limit the effects of interpretation from a single data source. In line with these guidelines, this thesis used several data collection methods to improve the reliability of the thesis. Unstructured interviews have been a frequent used method for data gathering in this thesis.

Robson and McCartan (2016) state that a commonly used typology for differentiating between interview types is: *structured*, *semi-structured* and *unstructured* interviews. The authors present a survey as an extreme example of a structured interview. The questions in a survey are fixed with a pre-defined order and standardized wording. Further the authors describe semi-structured interviews and unstructured interviews as more flexible alternatives which allow the subject interviewed being more free to say what they like on the broad topic of the interview.

Runeson and Höst (2009) present unstructured interviews as interviews where the researcher formulates questions as general concerns and interests from the researcher. Therefore, the conversation will develop based on the interest of the subject and the researcher.

Unstructured interviews with the CTO and the QA director of 1928 Diagnostic were held to find out if there was any other aspect of the variability approaches that could be interesting to investigate. The reason for conducting unstructured interviews was to explore potential gaps in the knowledge of the researchers by interviewing two persons that have experience in certifying a medical device product.

[1]https://www.scopus.com
[2]https://scholar.google.se/
[3]https://ieeexplore.ieee.org/

4.1.3 Summarize and Analyze the Information from the Literature and Unstructured Interviews

From the literature several variability approaches were found. The variability approaches that had a lacking description in literature were excluded. To strengthen the reliability the researchers tried to find several sources that stated the same information about the variability approaches. This was put together into a table where the data could be reviewed. From the unstructured interviews with the CTO and QA director the table was filled with new information. The new information that came from the interview was investigated further through researching if there was any literature that supported those claims.

4.1.4 Construct Solutions for Differentiating Certified and Noncertified Code

When the necessary information about each variability approach was available, the researchers analyzed whether the approaches could be used for differentiating certified and noncertified code. A discussion session was held between the researchers. The goal was to construct ideas on how the variability approaches could be used to achieve differentiation between certified and noncertified code. The reason for only involving the researchers at this stage was since they had the most knowledge of the variability approaches and due to time restrictions. The outcome of this iteration was which variability approaches could be used for differentiating certified and noncertified code, not specific to 1928 Diagnostics.

4.2 Iteration 2: Finding the most suitable variability approach for 1928 Diagnostics

This iteration focused on the second research question "*In the context of 1928 Diagnostics, what is the most suitable way to derive multiple products from a shared codebase with different certification levels?*". The input for this iteration was which variability approaches could be used for differentiating code with different certification levels which was the outcome of iteration 1.

4.2.1 Understand 1928 Diagnostics setting

To be able to find the most suitable variability approach for 1928 Diagnostics, the company and their setting needed to be understood by the researchers. To do so, a set of activities were performed: *Review of Artifacts*, *Presentations and Introductions by Developers*, *Code walk-through*, *Observing the Development Process* and *Unstructured Interview*.

- **Review of Artifacts**: To understand 1928 Diagnostics' system architecture and to understand what artifacts that was necessary to comply with IEC 62304, documents from a previous certification process were reviewed. This included the system's architectural design for everything related to the certified product.
- **Presentations and Introductions by Developers**: To further understand the system, specifically the parts that did not involve certification, two presentations were held by two separate developers. The outcome of these presentations was a rough mapping of how the system was designed and where certified parts were located.
- **Code walk-through**: To get a more detailed view of how the code was structured and potential limitations to which variability approach that could be suitable, a manual code walk-through was conducted. Here the researchers first looked at the code from a broad perspective and investigated how the system was setup from a technical perspective. Later the researchers looked at different parts of the code in more detail. The researchers did not need to understand the code exactly but a general overview was needed.
- **Observing the Development Process**: To understand the agile process for development, observations were used through attending daily stand up meetings and other planning related to the workflow. This was done to find potential requirements on how 1928 Diagnostics work with evolving their architecture and how this could affect which variability approach that could be the most suitable.
- **Unstructured Interview**: An unstructured interview was held with the QA Director of 1928 Diagnostics to find if there was any specific aspects of the variability approaches that was of interest to the company. There was a large focus on the requirements from the standard IEC 62304.

4.2.2 Focus group to choose the most suitable variability approach

When knowing the specific requirements for the company, a decision had to be made to choose which variability approach was the most suitable for 1928 Diagnostics. This was achieved through a focus group with the CTO and the QA Director of the company. The CTO was chosen to be one of the participants due to his technical experience and knowledge about 1928 Diagnostics' system, and the QA director because of his previous experience working with medical device products and standards.

The focus group was presented with how each variability approach should work in both the general case and specific to their system. Based on this presentation each variability approach was discussed with pros and cons. To ensure that each participant understood the approach and had the possibility to say his or her opinion did the moderators after each discussion ask if there was any unclarities or other

thoughts. The outcome of this iteration was a specific variability approach that suited 1928 Diagnostics. The focus group was voice recorded so that no information should be lost afterwards and the length of the focus group was approximately two hours.

Shull, Singer, and Sjøberg (2007) describe the empirical research approach focus group to be similar to brainstorming but that focus groups instead focus on a special question. Focus groups are not only used to generate ideas, but focus groups are also used when the researcher wants thoughts and feedback from the participants. Shull et al. (2007) continue describing focus groups as well planned discussions between the participants on a special subject. Normally the group is a setup of 3 to 12 participants and they should be selected through their own characteristics with regards to the research question.

4.3 Iteration 3: Mockup construction of the variability approach

The input to this iteration was which variability approach suited 1928 Diagnostics the best. In this iteration the chosen variability approach was implemented for a part of the codebase. This implementation is further on referred to as a *mockup* and consist of a branch with code of 1928 Diagnostics that is refactored to the chosen variability approach.

4.3.1 Choose part of codebase to implement the variability approach

To find the most suitable place in the codebase to construct the mockup a discussion was held with the CTO and QA Director of 1928 Diagnostics. The focus of the discussion was to find a part of the codebase where functionality was used from a certified and noncertified context. By finding a portion of the codebase with both a certified and noncertified context the evaluation would be more realistic.

4.3.2 Deep review of the selected codebase

The codebase contained no previous marking of what belonged to a certified product or not. Therefore a more detailed review was required. It was therefore crucial to understand what sub-parts needed to be certified since they would be handled differently with the variability approach. It was also important to understand which code was shared between certified parts and noncertified parts for the same reason. This was done by first creating a call-graph for the selected part of the system to get knowledge of all function calls for the selected part. Using the methods gathered from the call-graph, the codebase was searched for each of these methods recursively upwards to know if the method was used somewhere else and if that part was certified

or not. The information gathered from this step was which code that belonged to a certified product and which code was shared between the certified product and noncertified product. This information was later needed when the mockup was constructed.

4.3.3 Brainstorming and discussions about the implementation technique

When the researchers had a better knowledge of what code belonged to a certified product or noncertified product, the researchers had a brainstorming session to discuss architectural alternatives on how the mockup could be constructed.

Following the brainstorming session an unstructured interview was held with two experienced developers from 1928 Diagnostics who been continuously informed about the researchers work. The topic of this interview was to understand how they would structure the architecture using the selected variability approach. There was continuous discussions about the different pros and cons with each architecture that came up during the interview. This was done to lower the risk of missing potential alternatives in implementation.

4.3.4 Focus group to choose implementation technique

Following this unstructured interview a focus group was held with the same two developers and the CTO where an architecture decision was chosen. The questions that were used during the focus group can be found in section 7.1.2, the questions are not described here due to they require a bit of context from the result of iteration 3. After each question the moderators asked why the participants thought as they did, and for the pros and cons of each alternative that was discussed. The focus group was approximately one hour and it was voice recorded.

4.3.5 Development of the variability approach

When the implementation technique had been chosen could the researchers develop the mockup. To construct the mockup the researchers created a separate branch using git. With the help of the deep review of the selected codebase and assistance from developers of 1928 Diagnostics the selected code was refactored. The construction of the mockup took approximately a week of development. Following the construction of the mockup the researchers wanted to test if tooling could be added to assist the development process. A tool was constructed on a separate git repository and tested on the constructed mockup. The tool was a proof of concept in how the mockup could be used for developers in their daily work through continuous integration.

4.4 Iteration 4: Effects on BAPO

The input to this iteration was the constructed mockup of the chosen variability approach. In this iteration, the mockup's potential effects were evaluated with regards to the BAPO model for 1928 Diagnostics. It is directly connected to the third and last research question *How does the chosen variability approach affect the organization's BAPO?*.

4.4.1 Focus group to evaluate the effects on the BAPO

To answer RQ 3 a focus group was held with the QA Director, CTO and COO. These persons were chosen because of their individual knowledge areas. The focus group was divided into four blocks based on the BAPO model. Each block had multiple questions associated with it which had been created in advance. The questions can be found in Appendix A.1. Most of the questions are based on aspects mentioned in literature, especially Apel et al. (2013). However, some questions have been created from insights from the knowledge from the previous iterations. Most questions were targeted to a specific person, this since the researchers believed that the person would have the most knowledge about the question. However, the moderator always asked the other persons if they wanted to add something or if something was unclear.

Like the two other focus groups this was also voice recorded. One of the researchers took notes during the focus group while the other one managed the focus group. The focus group was approximately two hours. After the focus group both researchers went through the voice recording afterwards to finalize the result. Finally, the participants were given the opportunity to review the result.

4.5 Iteration 5: Decision support for selecting variability approach

The last iteration answer the final research question: *Can a methodology be constructed to derive the most suitable vari-ability approach for differentiation between certification levels? If so, how?*. This iteration focused on creating a methodology for choosing a variability approach for differentiating certified and noncertified code. This methodology was constructed by synthesising the work done during iteration 1-4 from a methodology perspective by the researchers and keep what worked well and improve the steps where the methodology fell short. It were the core results from the previous iterations that was used to create this methodology.

4.6 Threats to Validity

This section presents the different threats to validity for this study. The section is divided into the classifications: construct validity, internal validity, external validity and reliability which are used by both Runeson and Höst (2009) and Yin (2003).

4.6.1 Construct Validity

Runeson and Höst (2009) describe construct validity as: "This aspect of validity reflect to what extent the operational measures that are studied really represent what the researcher have in mind and what is investigated according to the research questions."

This study uses focus groups which have weaknesses like all other qualitative research methodologies (Shull et al., 2007). Two weaknesses focus groups have are that they can become biased by group dynamics and that the group sizes can be too small. That the group size is too small could be a threat to validity for this thesis. However, since 1928 Diagnostics is a small company it is difficult to involve more people than done through this study. That group dynamics could be biased has been mitigated through discussion control by the moderators. Questions which have been more relevant for a certain subject have been directed to that person but all subjects have been asked if they want to add something.

One threat to construct validity that exists is the risk of misinterpretation of questions and concepts during focus groups and the interviews. Concepts or questions with regards to for example what a *component* is and how it can be used for separation of certified and noncertified code can have very different meaning depending on your background. To cope with this issue, the concepts were explained thoroughly before proceeding to any discussions or questions. Furthermore, the people involved in the study were continuously informed about the researchers work and progress to lower the risk of misunderstandings and misinterpretation of concepts and questions. The subjects were also given opportunities to review the data collected so called member checking to make sure that no misinterpretation of their phrasing or intentions occured (Shenton, 2004).

4.6.2 Internal Validity

Internal validity concerns the ability to draw causal conclusions. For example, if a researcher tries to determine whether event y leads to x there might be a third event which is the real reason for x to occur which the researcher is not aware of (Runeson & Höst, 2009).

This thesis does not aim to draw causal conclusions and therefore this risk should be low. However, there is always a risk of confirmation bias from the researchers. Nickerson (1998) describes the concept neatly by saying: "People tend to seek in-

formation that they consider supportive of favoured hypotheses or existing beliefs and to interpret information in ways that are partial to those hypotheses or beliefs". The risk of confirmation bias for this study is no exception, but since we are two researchers rather than one we consider the risk lowered. Additional triangulation was done by having a supervisor in the form of a university researcher where each step has been discussed in advance and iterated on. The study has also been peer reviewed at seminars conducted at the university for further triangulation and by member checking of the participants of the study.

4.6.3 External Validity

External validity is concerned with what extent a study's findings can be generalized and to be of interest to people outside of the studied case (Runeson & Höst, 2009). Runeson and Höst (2009) state for case studies that: "There is no population from which a statistically representative sample has been drawn." However, they continue to describe that the intention with a case study is to extend the generalizability of the results to cases with common characteristics where the findings are relevant.

This thesis is conducted at one company, therefore no statistical conclusion can be drawn by this study. However, like Runeson and Höst (2009) argue, for studies with similar characteristics the result could still be relevant. Flyvbjerg (2006) argues similarly that one case study cannot be used to statistically generalize results unless more case studies are performed but that it would be incorrect to conclude that one cannot generalize from a single case. Flyvbjerg continues by saying that it rather depends on how well the case is chosen. We consider the company to be well chosen. They are interested in moving towards an SPL and clearly relate to the research questions. A positive and negative aspect of using this company for the research is the size. For a larger company it would require to involve a lot more personnel which increase the risk of miss interpretations since everyone taking part of the study will not be as informed. The people involved at 1928 Diagnostics were heavily involved in the research decreasing the risk of miss interpretations. However, the downside is that aspects which might only appear in larger companies are lost. This downside is mitigated somewhat by the QA Director of 1928 Diagnostics since he has experience from larger companies within the MedTech industry.

4.6.4 Reliability

Threats to reliability consider the risk of the data and analysis being dependant on the specific researcher. If another researcher would conduct the same study the result should be the same (Runeson & Höst, 2009). This thesis is conducted by two researchers which mitigates the risk compared to only being one.

5

Iteration 1: Finding variability approaches which can be used for differentiation

In iteration 1 the focus was on finding what SPL variability approaches exist and which of these could be used for differentiating certified and noncertified code. This was done to answer RQ 1 *"Which variability approaches for implementing an SPL can be used for differentiating between certified and noncertified code?"*. The variability approaches' different attributes were also investigated since system requirements then could be mapped to certain variability approaches that could be a potential fit.

Note that RQ 1 is not in direct connection to 1928 Diagnostics. RQ 1 is about finding all variability approaches that are available for systems in general. RQ 1 was created from a need from 1928 Diagnostics, which is to answer RQ 2 *In the context of 1928 Diagnostics, what is the most suitable way to derive multiple products from a shared codebase with different certification levels?*. To answer RQ 2 there is a need to know which variability approaches exist. The results of RQ 1 could be beneficial not only for 1928 Diagnostics but also for other companies or organizations with a similar need.

5.1 Results

From the literature several variability approaches were found. To achieve variability Apel et al. (2013) describe both language-based and tool based approaches. These approaches are *parameter-based*, *design patterns*, *frameworks*, *component and services*, *version control*, *build system*, *preprocessor*, *feature-oriented programming* and *aspect-oriented programming*. The approaches are described more detail in Chapter 2.

A decision was made not to include build systems as a variability approach since it was considered as an add-on together with other variability mechanisms. Code generation was another variability approach that was mentioned by Pohl et al. (2005) and van der Linden et al. (2007), but due to the limited descriptions in the SPL literature it was left out. Worth noting is that *components and services* will further

on be referred to as only components. Since components provide a service and could also function as a web-service if deployed.

5.1.1 Variability Approaches' Quality Attribute

The eight variability approaches that were found from the literature have different attributes and they can fit in different contexts depending on which attributes the variability approach has. Apel et al. (2013) introduces six quality criteria for the variability approaches and define them as:

- **Pre-planning effort** - Pre-planning is the effort made to ease the workload later when changes or new features are needed for a system.
- **Feature traceability** - Feature traceability is the concept of being able to connect a problem to its solution, for example, where a user story is implemented in the code.
- **Separation of concerns** - The principle *separation of concerns* recommends that related code should be implemented together and the opposite, that unrelated code should be separated.
- **Information hiding** - A system can be divided into modules. The modules which communicate with external parts then include only the information that is required by the external part. Thus the internal modules can hide information that the external modules do not need to know, or should not know. This is a good methodology to follow since a developer can work and think about a module without knowing the internal implementations of other modules.
- **Granularity** - Features induce changes upon a program, depending on how large the impact is the granularity is different. Features implemented at lower levels, which induce small changes, are called fine-grained. Those implemented at higher levels, which induce large changes, are called coarse-grained.
- **Uniformity** - When a codebase is uniform, the features are written and set up similarly to each other if possible. Thus, when features are similarly implemented, the developer does not need to re-familiarize themselves with each feature too much.

In addition to the six quality attributes which have been described in Apel et al. (2013), other quality attributes have been found. Some of these attributes have been mentioned in the literature briefly and other attributes have been found through discussions and unstructured interviews between the researchers, CTO and the QA director who also has experience from other companies with regards to certification of products. The other seven quality attributes found:

- **Feasible to differentiate code with different certification levels** - This is whether the variability approach can be used for differentiating code with different certification levels. The need for this has emerged through discussions

with the QA director regarding the certification process.

- **Transition in steps possible** - This is whether the variability approach can be implemented in steps or if the approach needs to be implemented all at once. This is important to know if the system is big. If so, it could then be troublesome to do it all at once. This attribute also comes from discussions with the QA director.
- **Ease of adding new features similar to existing ones** - Code which is shared between other features does not need modification. The literature mentions the ease of evolution for the variability approaches. Based on this there was a discussion between the researchers about how the evolution could be done and it was split into two attributes: this one *Ease of adding new features similar to existing ones* and the opposite one *Ease of adding new features not similar to existing ones*.
- **Ease of adding new features not similar to existing ones** - A new feature needs the existing shared code to be extended or modified. This could be interesting from a business perspective since it shows how easy it could be to add a new part to a system.
- **Support removal of dead code** - Dead code is code which is never used and thus can be removed without affecting the outcome of the program (Debray, Evans, Muth, & De Sutter, 2000).

  Depending on how you interpret the regulatory requirements that come with certifying a product you could interpret the removal of dead code to be beneficial. Another reason for removing dead code could be because of limited resources, this is often the case in embedded systems where memory resources are limited. Only few variability approaches support the removal of dead code. The possibility to remove code is an attribute which is mentioned in literature and it was also is mentioned in discussions about the certification with the QA director.
- **Reusability** - This is to which degree the variability approach reuses the code which is shared between features. Reusability is the main goal of using SPLE and it is therefore mentioned in literature for some of the variability approaches.
- **Artifact support** - Artifact support means that the approach can handle artifacts other than code. This includes documents such as a requirements specification or a risk analysis. The literature does mention this aspect to some extent. When discussing the certification process with the QA director it was mentioned that this is helpful for the certification process if one, for example, could connect code with a requirements specification.

The variability approaches' quality attributes values have been summarized in Table 5.1. The values which can be backed by Apel et al. (2013) have been marked with a star (*). Important to note that the values that are backed by literature is still an interpretation from us as researchers on how the value should be scaled in the table.

The quality attributes which have been created through discussion and unstructured interviews cannot all be backed by literature and they can therefore only be seen as hypotheses.

Apel et al. (2013) also introduce which dimensions of variability the approaches have; what binding time they have, if they are a tool- or language-based and if they are annotation- or composition-based. These dimensions are important when one wants to know how the variability approach will work technically and how it can be used. What dimensions a variability approach has can also be seen in Table 5.1.

This table can be used for anyone who has an interest in understanding how they could set up a variability approach for differentiating code with different certification levels.

	Parameter	Design Patterns	Components	Version Control	Pre-processors	Frameworks	Feature Oriented P.	Aspect Oriented P.
Business related								
Feasible to differentiate code with different certification levels	Yes	Yes	Yes	Yes	Yes	No	No	No
Transition in steps possible	Yes	Yes	Yes	Yes	Yes	No	No	No
Ease of adding new features similar to existing ones	High	High*	High	High	High	High	High*	Medium
Ease of adding new features not similar to existing ones	Medium	Medium	Medium*	High	Medium	Low*	Low	Low
Support removal of dead code	No	No	No	No	Yes	No	No	No
Architecture related								
Uniformity	Low*	High*	Medium*	Low*	Low*	High*	Medium*	Low*
Reusability	Low	High*	High	Low	Low	Medium	Medium	High
Separation of Concerns	Medium*	High*	High*	Low*	Low*	High*	High*	High*
Information Hiding	Medium*	High*	High*	No*	No*	High*	No*	Low*
Granularity	Line*	Line	File*	Line*	Line, Method, File*	Method, File*	Method*	Line *
Process related								
Preplanning effort	Low*	Medium*	Medium*	Low*	Low*	High*	Low*	Low*
Feature Traceability	Low*	High*	High*	Low*	Medium*	High*	High*	High*
Artifact support	No*	No	No	Yes*	Yes*	No*	Yes*	Experimental only*
Organization related								
-								
Dimensions of variability								
Binding time	Run, load	Compile, Load, Run	Compile	Compile	Compile	Load, Run	Compile, Load	Compile, Load
Language based	Yes	Yes	Yes	No	No	Yes	Yes	Yes
Tool based	No	No	No	Yes	Yes	No	No	No
Annotation based	Yes	No	No	No	Yes	No	No	No
Composition based	No	Yes	Yes	No	No	Yes	Yes	Yes

Table 5.1: Relations between the variability approaches and their attributes, general for all projects

5.1.2 Variability Approaches Conformity to Certification

The first attribute in Table 5.1 is whether the variability approach can be used for differentiating certified and noncertified code. Whether a variability approach could be used for this purpose or not is based on discussion and brainstorming between the researchers.

Through continuous discussions with the QA and CTO at 1928 Diagnostics two use cases were found for the differentiation. These use cases should be general for all systems. This has however not been validated since 1928 Diagnostics is the only company researched.

- **Use Case 1: Altering control flows:** The code's control flow differs depending on whether the code which calls it certified or not.
- **Use Case 2: Usage of the same code**: A certified product and noncertified product run the same code (the control flow does not change). The code then needs to be marked whether it is associated with a certified product or not.

The first case is when a piece of code's control flow differs depending on whether the code which calls it is certified or noncertified. Then the alternating flows need to be marked whether they should be certified or not. The other case is when certified and the noncertified code uses the same functionality. The functionality which is shared between both certified and noncertified code then needs to be certified since it is used by the certified code. A project can, of course, need for both of these cases and can then combine the possible solutions which are presented in this chapter.

A summary of the variability approaches' individual possibilities to be used for the cases can be found in Table 5.2.

Variability Approach	**Case 1**	**Case 2**	**Motivation**
Parameter	Yes	No	A language-based approach that can support different control flows in a natural way, for example, with if-statements.
Design Patterns	Yes	Yes	A language-based approach. Solutions were found for use case 1 and 2 using inheritance.
Framework	No	No	For a framework to work the framework itself would have to be certified, this since it constructs certified products. The framework could then adopt plugins to construct different products. This would not fit a system where a large portion of the code does not need certification and it would then not assist the separation.
Component	No	Yes	A system can be divided into components where each component has a certification level.
Version Control	Yes	Yes	There are two viable solutions using version control. Solution 1: Branches can be created for different features and each feature can have a certification level. Then a product can be composed of the merging of different branches. Solution 2: One branch for each certification level (product), this solution usually have a lot of duplicated code and is hard to maintain.
Preprocessor	Yes	No	A preprocessor can be used for use case 1. It can be used to remove unused code in the control flow which is not used.
Feature-oriented programming	No	No	No solutions were found without additional strategies added on top.
Aspect-oriented programming	No	No	No solutions were found, primarily since it targets only the crosscutting concerns and not the codebase as a whole.

Table 5.2: The possibilities of using the variability approaches for differentiating code for the two use cases

5.1.2.1 Solutions for Differentiating Certified and Noncertified Code

To demonstrate the viable solutions that were found, an example scenario has been created which can be seen in Figure 5.1. This example has been inspired by 1928 Diagnostics' codebase. Each box represents a piece of code. The dark blue boxes are parts which need certification and the white boxes are those which do not need certification. The light blue box, D, is the code which is shared between both the certified and noncertified parts, A and B. The boxes C and E is there to understand that A and B often use code other than the shared part D.

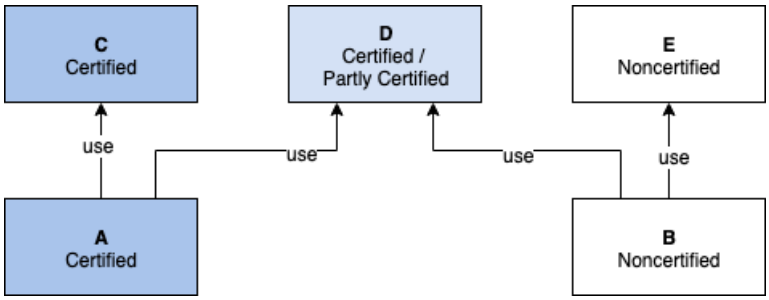

Figure 5.1: Exemplification of problem

Parameter-Based
Parameter-based variability could support variability between the certified and non-certified code for use case 1, altering control flows. The level of granularity can be chosen by the developer, from very high level down to specific lines. An example of how to achieve variability with parameters can be found in Figure 5.2. Depending on where the method D is defined in the system can it be defined as fine-grained or coarse-grained. If the effects are large because of the control-flow is it coarse-grained, and if the effects are small is it fine-grained.

```
def D:
    # Some shared functionality
    # between C and Noncertified code
    # Some more...
    if (C):
        # Do something certified related
    else:
        # Do something noncertified related
    # Some shared functionality
    return result
```

Figure 5.2: Separation of certified and noncertified code using parameter-based variability on a line granularity

Design Pattern
There are many available design patterns, however, an example solution has been made using inheritance to demonstrate the idea. This solution for use case 1, see

Figure 5.3, is built on that part A and B need different implementations or methods from D. Because of this the code which is shared for both A and B can be placed in the class D-shared. Then each implementation of D can inherit from that class.

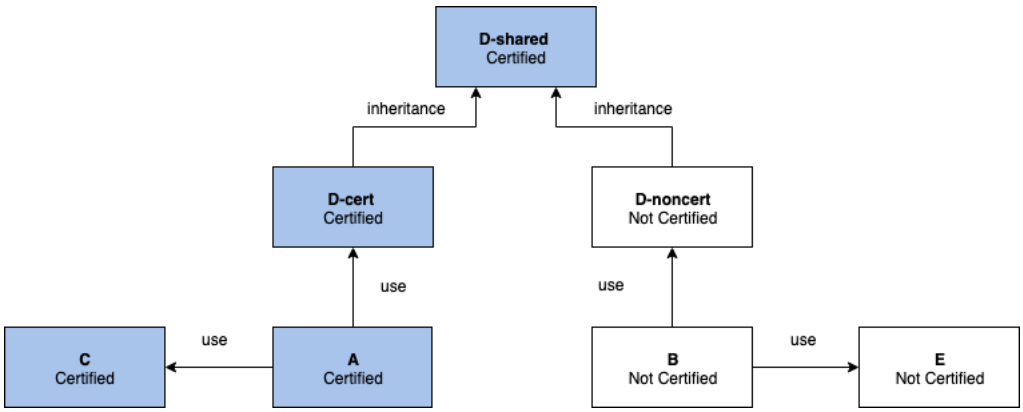

Figure 5.3: The shared code for D is within the class D-shared, individual implementations are in either D-cert or D-noncert

The solution for use case 2 is to mark if a certain class is certified or noncertified. This can be achieved if all classes with features inherit to a super-certified-class or extend a super-certified-interface, where it says that the class is certified or not. These classes or interfaces are not intended to contain code, they are rather to be used as a marker if the class is certified or not. In Figure 5.4.

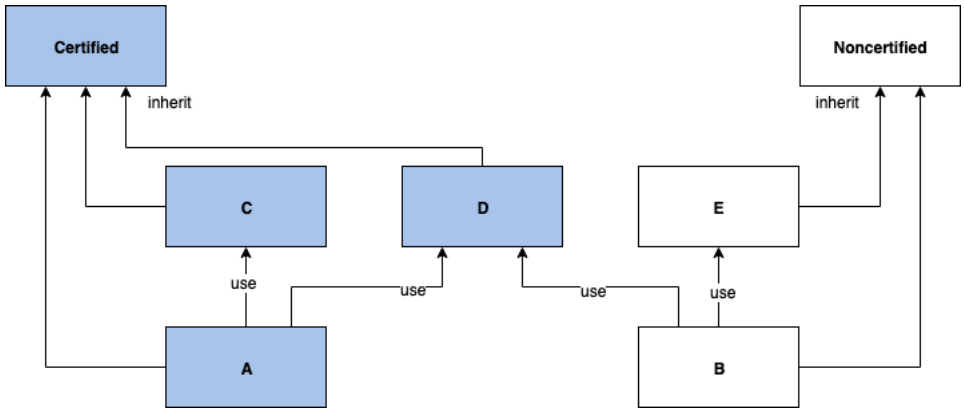

Figure 5.4: Parts of the code inherit to a 'super-class' Certified or Noncertified

Components

Components can support the separation between the certified and noncertified code for use case 2. They can also function in various setups due to the flexibility in size. In the problem example in Figure 5.1 can one think that each box is a component and that each component has a certification level. Then each component also needs some type of annotation, so a developer knows if a component is certified or not.

Version Control

There are two viable solutions using version control for use case 2. In the first solution version control could be used to differentiate certified and noncertified code.

The solution found builds on using branches for different features and each feature is then certified or noncertified. The solution is presented in figure 5.5. Branch D has some shared functionality that both A and B want to use. A and B, then can pull the branch down to their branch when they need it. A and B can decide which version of each branch they want to use.

One should not mix up this solution up with using version control as it usually is used. Version control can, of course, be used together with any other solution in this chapter but not as a solution for differentiating code with different certification levels.

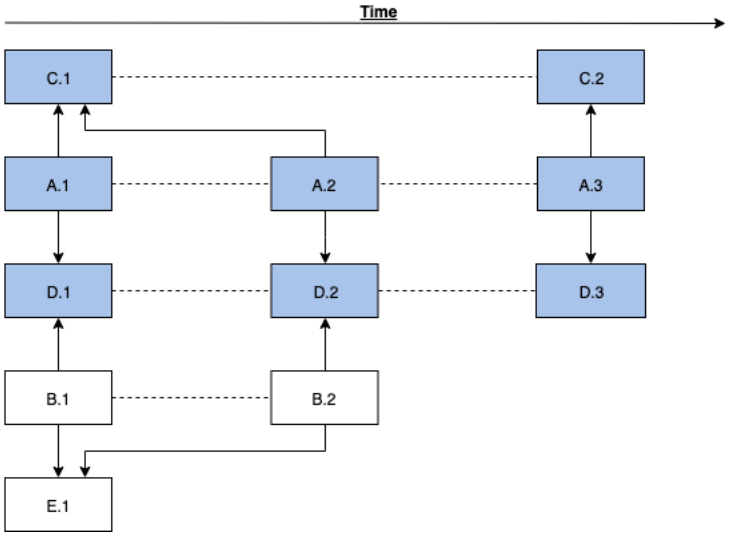

Figure 5.5: Each feature has a specific certification level. Products can be composed by merging the branches together.

The other solutions are using a clone-and-own methodology. Here each product has its branch with a specific certification level. This type of solution usually has a lot of duplicated code. It is a fast way of producing different products but it can become hard to maintain the code since much code is duplicated.

Preprocessor
The solution for use case 1 that is based on a preprocessor is similar to the parameter-based approach, but rather than controlling the control flow at run-time, a preprocessor supports this during compile time by removing code. When compiling the code in Figure 5.6 it could result in the code in Figure 5.7, i.e. removing function B which in this case is noncertified.

5.2 Discussion

This section will discuss the result presented above by discussing different variability mechanisms individually. However, before doing so, it must be noted that there could be more ways to use the variability approaches to differentiating code between

```
def D:
    # Some shared functionality
    # between C and Noncertified code
    # ifdef(A)
    A()
    # endif

    # ifdef(B)
    B()
    # endif
    return result
```

Figure 5.6: Certified and noncertified code together before running a preprocessor.

```
def D:
    # Some shared functionality
    # between C and Noncertified code

    A()
    return result
```

Figure 5.7: Result of running a preprocessor from the previously shared certified and noncertified code

different certification levels. The ones presented are the ones for which we found a solution. There might be ways to use the variability approaches that we have missed.

5.2.1 Parameter

Parameter-based variability could be used for differentiation between certified and noncertified code. The usage of parameters to achieve separation between certification levels could potentially be good for cases where there are not a lot of products that need to be derived. It is also an approach which is widely supported in programming languages meaning that for most companies this could be applicable. Because of the wide support we believe it could be used for both frontend and backend. The pre-planning effort is also low (Apel et al., 2013) which means it could be a favourable alternative for an organization that works in an agile fashion. However, there are problematic areas when using parameter-based variability. The more a developer adds specific control flow based on parameters the harder it will be to follow a specific control flow for different products, even more so if it is done on a fine-grained granularity. From a certification point of view, this could be problematic since at least for standards such as IEC 62304 you need to be able to follow the control flow for the product that you intend to certify and keeping this information

up to date, could end up being problematic.

5.2.2 Design Patterns

Using design patterns to support the separation of different certification levels could be suitable for projects written in an object-oriented programming (OOP) language which supports principles such as inheritance. Design patterns is something most developers are familiar with and we believe works both well for agile or waterfall working methodologies. Design patterns also support an extension of functionality which makes it more suitable than for example parameters for a larger pool of products. The tradeoff for design patterns is that they require more pre-planning to support extension. Depending on how much pre-planning required it could be problematic for working agile. Another potential problem is the use case of frontend where OOP is not as common meaning it might not be a viable alternative.

5.2.3 Component

Components are a variability mechanism that works well for both waterfall and agile processes. As a developer, you can both design all components you want upfront but it also works well to construct one at a time along the way due to new needs. The idea of having a clear interface and functionality that is reusable could work well for both frontend and backend. The possibility to construct "boxes" which have a requirements specification containing input and output is something that works well for standards such as IEC 62304. A downside with components is the necessary glue code that is required to construct new products. If your goal is to construct a lot of products using a set of components there is a risk that the glue code work will lead to a too high cost to pay.

Components function on a higher granularity than many other variability approaches which could give them the potential to be combined with other variability approaches. Potentially, one could combine the component variability together with something such as parameters which functions on a lower granularity. However, due to time restrictions this is not something this thesis looks into.

5.2.4 Version Control

Version control could be a suitable alternative for separation between certification levels if you have a small number of features and want to keep them completely separate. Tools such as git are also something that most developers are familiar with which makes it easy to adapt. Furthermore, version control works for both agile and waterfall development.

Version control creates a clear distinction between what certification level a feature belongs to. The major downside of using version control is the lack of natural reuse and that it could become problematic when one wants to merge in another branch.

Assuming that the company or organization consist of several teams working on different certification levels it could be hard to keep track of the relevant version and which dependencies there are. Furthermore, it might be problematic to scale up the number of features using version control.

5.2.5 Preprocessor

Using a preprocessor for separation of different certification levels could be a good choice if the products to construct have hard restrictions on not containing any dead code. Not only does a preprocessor support removing code, but it also does so at a fine-grained granularity. A preprocessor is also suitable for both small and large scale projects where multiple products are to be constructed. Working with a preprocessor should neither hinder agile or waterfall software development. A potential downside of using a preprocessor is that we have not found any examples where it is used for frontend applications while, on the other hand, it is commonly used in the embedded systems industry. Another issue that might be one of the reasons why we have not found any examples for it in frontend could be that most preprocessors are language-specific.

6

Iteration 2: Finding the most suitable variability approach for 1928 Diagnostics

Two use cases were presented in the previous chapter for differentiating certified and noncertified code:

- **Use Case 1: Altering control flows:** The code's control flow differs depending on whether the code which calls it certified or not.
- **Use Case 2: Usage of the same code**: A certified product and noncertified product run the same code (the control flow does not change). The code then needs to be marked whether it is associated with a certified product or not.

Through discussions with 1928 Diagnostics it was discovered that they only had the need for the second use case. As mentioned in 2.4 1928 Diagnostics have different pipelines for the pathogens. Some analysis (features) are performed for all pathogens, others are run for almost all while some are only performed to a specific pathogen. The variation in what analysis to run are based on biological differences. There are also differences in which databases are used based on pathogens and their threshold for their analysis but this is not the variability focus of the thesis. In iteration 1 five variability approaches could be used for either use case one or two in total. However, only three of these could be used for the second use case. These variability approaches are:

- Design Patterns
- Component and Services
- Version Control

To answer RQ 2 *In the context of 1928 Diagnostics, what is the most suitable way to derive multiple products from a shared codebase with different certification levels?* the result from Chapter 5.1 was used and built upon by adding attributes specific for 1928 Diagnostics to find the most suitable alternative for the company. This chapter first introduces these 1928 Diagnostics specific quality attributes gathered through an interview. Following, a focus group and their opinion of different variability approaches are presented. Finally, the variability approach that was chosen by 1928 Diagnostics is presented.

6.1 Results

Through an unstructured interviews with the QA director additional quality attributes were elicited. These are:

- **Compatibility with backend** - If the variability approach potentially worked for the backend.
- **Compatibility with frontend** - If the variability approach potentially worked for the frontend.
- **Ease of implementation** - How easy it would be to work with the future implementation of the variability approach.
- **Transition time** - How long it would potentially take to implement the variability approach in their system.

One factor that was particularly interesting for the company was whether or not the separation of different certification levels could be achieved for *backend or frontend* specifically. The architecture and the details of how the frontend and the backend are constructed are different. Also, the user interface must provide not only the functionality but also high usability which is much more difficult with automated techniques (Pleuss, Hauptmann, Dhungana, & Botterweck, 2012).

Another factor that 1928 Diagnostics found valuable was the *transition time* for moving to a specific variability approach. The longer it would take to transition the longer they would be restricted on developing new functionality which is a risky move for a small company. The last factor of importance for the company was the *ease of implementation (agility)*. The company could not stress enough that they wanted to continue to work agile and therefore this was something that was included.

The 1928 Diagnostics specific attributes and their valuation can be found in Table 6.1 where they are categorized by the categories of BAPO. The specific quality attributes are presented in the next section. The attributes are as stated before elicited from 1928 Diagnostics, they could however be interesting for other organizations. The attributes and their estimation for each variability approach can be found in Table 6.1.

	Design Patterns	Components	Version Control
Business related			
-			
Architecture related			
Compatibility with Backend	Yes	Yes	Yes
Compatibility with Frontend	Yes	Yes	Yes
Process related			
Ease of developing with (agility)	High	High	Low
Organization related			
Transition time (implementation time)	Medium	Medium	High

Table 6.1: Relating quality attributes to variability approaches specific for 1928 Diagnostics

6.1.1 Suitability of Variability Approaches

To narrow down the variability approaches to those that could be suitable for 1928 Diagnostics' needs, a focus group was held. Since the variability approaches had many attributes that the focus group could discuss around, some of 1928 Diagnostics' main requirements was selected as main discussion points. These main requirements were:

- Easy to work with in an agile way
- Easy to develop new products through a flexible codebase
- Improved feature traceability
- Not to long transition time

In the following sections, the variability approaches' differentiation between certified and noncertified code are presented with opinions and thoughts of the focus group.

6.1.1.1 Design Patterns

The focus group saw several problems with using design patterns for their setting. One of the core issues was the possibility to both apply this strategy to the frontend as well as the backend. Furthermore, the developers thought that this change would be a somewhat large change to their existing codebase even for only the backend which would require less change than the frontend.

6.1.1.2 Version Control

Version control was one of the separation alternatives that the focus group wanted to discard quickly. The reason they were highly skeptical was due to previous developer experience to maintain two versions of something in parallel and keeping them in

sync. The focus group also thought that this approach would be too ad-hoc and hard to maintain a structured process as required by IEC 62304. They further believe it would be unfeasible to keep the necessary artifacts up to date the larger the codebase and company grows.

6.1.1.3 Components

The focus group found using components a good way to differentiate certified and noncertified code. Also, by moving to use components which requires a clear interface to input and output, they would be forced to remove the usage of passing around a large context object which they currently do. Another factor that the focus group found useful with components is the feature traceability that components provide.

6.2 Discussion

The focus group decided to try to evaluate components as a way to differentiate certified and noncertified code. The reasoning for doing so is that 1928 Diagnostics does not need to customize functionality based on what pathogen pipeline is calling the function which from the previous chapter is defined as use case 1. Rather, the functionality stays the same but might be used in another step of the process than in another pipeline. By using components they can simply include or exclude functionality in accordance to what a certain pathogen needs. The control flow of the functionality does not change (use case 2). Further, design patterns were considered too big of a change and be difficult to implement. Version control was perceived as too ad-hoc which potentially could lead to bad code in the long run.

From a certification standpoint and being inline with IEC 62304 which is the standard the company intends to follow feature traceability is an essential factor. Apel et al. (2013) describe components as something that provides good feature traceability so inline with the existing theory we believe components are a good option from that aspect.

7

Iteration 3: Mockup construction of the variability approach

In the previous chapter, a description of how the focus group selected components was presented. This chapter presents the implementation of the variability approach, which from now is called *mockup*, and how it was constructed. This mockup is then used in the next chapter to answer the final research question: *How does the chosen variability approach affect the organization's BAPO?*.

7.1 Result

The first that needed to be decided in this iteration was which part of 1928 Diagnostics' codebase that was to be tested with the variability approach. The chosen part had both a certified and noncertified context which shared some features. The chosen part of the codebase partly belongs to a pipeline that is considered safety-critical. To find potential safety-critical dependencies, usage from non-safety-critical context and get a better understanding of the codebase a call graph was constructed. Each method that was called by this selected part was then searched for in the whole codebase to find other callers. By searching for usages other than from the chosen pipeline, methods which might have another certification level could be found. This is important since they will be affected by the changes which are to be done in later steps.

7.1.1 Possible Component Setups

After getting a deeper understanding of the dependencies and architecture, the thesis workers found through their knowledge about the system and discussions that components could be set up in different ways. Through a brainstorming session two ways were created which could be used to differentiate between certified and noncertified code. One alternative is *functionality first* which means that the components are structured based on functionality first and then marked with certification or not. The other alternative is *certification first* where the components are structured based on certification and then on functionality. These two alternatives are more thoroughly presented in the upcoming sections.

To alleviate the risk of not seeing any other setups for components an unstructured interview was held with two developers from 1928 Diagnostics. The developers were asked how they would structure the components. The developers found the same two alternatives as the thesis workers and did not present any other alternatives than those two.

- **Functionality Followed by Certification Categorization:** One of the ways of setting up components could be to think about functionality first and then after the components have been set up by functionality, certification can be added afterwards. This approach needs some annotation of the certified parts to be in place. Examples of this could be file or method naming which or adding special comments annotating the specific certification level.
- **Certification Followed by Functionality Categorization:** The other way to setup components would be first to create components based on whether they are certified or not and then have sub-components which are divided by functionality. To be able to use a certified code part the developer needs to go through the certified-component interface.

7.1.2 Focus Group to Select Component Setup

To evaluate which setup of components that were most suitable for 1928 Diagnostics a focus group was held. There were three things to be adressed which all were connected. The focus group discussed the three questions in the same order as the list below.

- **Does 1928 Diagnostics want to divide the components firstly on functionality or certification?** As earlier mentioned in section 7.1.1 there are two alternatives for how to structure categorize the components where one must be chosen before implementation.
- **At what level of granularity does 1928 Diagnostics want to annotate the components?** There are multiple levels for how to annotate something as certified such as file, method, line or a whole component.
- **If the component model which takes functionality in consideration first is chosen, how does 1928 Diagnostics want to implement the annotation if a piece of code is certified or not?** Related to the previous question about granularity an implementation must be achieved somehow. How that is to be done is the purpose of this question.

The focus group chose to move on with *functionality followed by certification* and the reasoning for doing so can be found below.

7.1.2.1 Functionality Followed by Certification Categorization

According to the focus group this model was the natural way of setting up components for a developer since the related code is near each other. The closeness between

similar code makes it easier for a developer to find the functionality they are looking for. However, it is not clear how the developer or any other tool would know if a piece of code is certified or not. This would need some extra annotation to solve that problem. Since the focus group chose this model to be implemented in the mockup was the two other questions, how to use granularity and annotation, solved in the context of this model. There were three options for the granularity; line, method, and file/module. The focus group directly thought that the line-granularity was too much work and not appropriate for 1928 Diagnostics' codebase. The choice was therefore between method and file level. The focus group chose to move forward with module granularity since it gives a clearer view of what code is certified or not. If a component would have method-granularity methods could become mixed up and used in the wrong way more easily. This model does not naturally solve how to understand what is certified code or not. Because of this, that specific solution requires need some type of annotation for that problem. The options which were discussed in the focus group were: file naming, a modules init file and comments. The focus group thought that naming the file/module could be a problem if they wanted to change the name in the future. Comments could be unclear if not used in a precise way. A member in the focus group mentioned that since 1928 Diagnostics uses Python[1] they could use the regular package functionality and create an init parameter to "certified = true". This was the choice which the focus group liked the most and decided to go with.

7.1.2.2 Certification Followed by Functionality Categorization

With the model "certification followed by functionality" the focus group understood that it probably would be the better option to know what code is certified clearly. However, the focus group saw that the negative effects of this approach outweigh the positive. One negative effect is that this is not how they think about the order of prioritization when designing their architecture. Separating functionality has precedence over certification levels, not the other way around. A second issue mentioned is that they believe it could be problematic to combine functionality that is related but with different certification levels if the functionality is further apart. An example of the second issue is when the noncertified code would like to use certified functionality. It would then be more problematic if the functionality is located in a completely different part of the codebase compared to having it nearby and marked as certified. The third negative effect mentioned was that it could be difficult to find functionality without knowing the certification level. Instead of looking for related functionality in one part of the codebase you would have to look at both the certified and noncertified part and their related functionality to find what you are looking for. The group said that it is already hard to find what you are looking for and they do not want to make it worse. A fourth negative aspect of this model was that it would be hard to change the certification level of a part of the system. Using this model the part of changing would need to be moved to another place. The fifth and final thing the group mentioned was that it potentially could be a problem with

[1] https://www.python.org/

this kind of separation if one uses some web framework, as 1928 Diagnostics do. Web frameworks often require a certain file structure to function which might not be compatible with this model.

7.1.3 Constructing a mockup

After deciding on how to construct the mockup using functionality first and annotating with the help of python regular packages in the focus group, the next step was to construct it. Due to confidentiality reasons, the whole code cannot be disclosed but an example can be seen in Figure 7.1 where the file *___init___.py* contains "certified = True". Python's official documentation describes python regular packages as: "A regular package is typically implemented as a directory containing an ___init___.py file. When a regular package is imported, this ___init___.py file is implicitly executed, and the objects it defines are bound to names in the package's namespace. The ___init___.py file can contain the same Python code that any other module can contain, and Python will add some additional attributes to the module when it is imported." (Python Software Foundation, 2019)

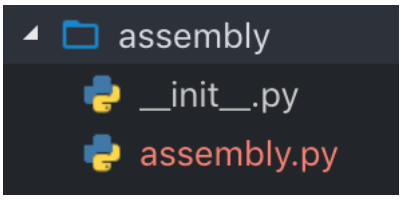

Figure 7.1: Regular package annotated as certified through the ___init___.py file

This step involved refactoring existing code and putting it in different components based on functionality and annotate them with the help of the___init___.py file. Also, 1928 Diagnostics previously used a *context object* which was passed around to a majority of the functions. This context was removed as input and replaced by the required input to make each component have clear interfaces. Previously functions had input from the context that was not necessary for them to achieve their task.

Furthermore, a tool was created as a proof of concept that the refactoring to components could be used to assist developers to check whether or not a python package can be considered certified. The results from running the tool on the mockup can be seen in Figure 7.2 and 7.3. The code for the tool is public and the latest version can be found at `https://github.com/Oscmage/certified_scanner`. The version used at the time of writing this thesis can be found in Appendix A.2

7.2 Discussion

The choice of constructing components based on functionality first rather than certification comes with benefits such as being similar to the developers' current way

```
The following files are certified and ok:
[ '/Users/OscarEvertsson1/Programming/alex/worker/executor/tasks/gene_calling/__init__.py',
  '/Users/OscarEvertsson1/Programming/alex/worker/executor/tasks/gene_calling/gene_calling.py',
  '/Users/OscarEvertsson1/Programming/alex/worker/executor/workpath/__init__.py',
  '/Users/OscarEvertsson1/Programming/alex/worker/executor/workpath/workpath.py',
  '/Users/OscarEvertsson1/Programming/alex/worker/executor/shell/__init__.py',
  '/Users/OscarEvertsson1/Programming/alex/worker/executor/shell/shell.py',
  '/Users/OscarEvertsson1/Programming/alex/worker/executor/tools/blast/__init__.py',
  '/Users/OscarEvertsson1/Programming/alex/worker/executor/tools/blast/blast.py',
  '/Users/OscarEvertsson1/Programming/alex/worker/executor/tasks/assembly/__init__.py',
  '/Users/OscarEvertsson1/Programming/alex/worker/executor/tasks/assembly/assembly.py',
  '/Users/OscarEvertsson1/Programming/alex/worker/executor/tasks/mccortex/__init__.py',
  '/Users/OscarEvertsson1/Programming/alex/worker/executor/tasks/mccortex/mccortex.py',
  '/Users/OscarEvertsson1/Programming/alex/worker/executor/tasks/_common/sample_files/__init__.py',
  '/Users/OscarEvertsson1/Programming/alex/worker/executor/tasks/_common/sample_files/sample_files.py',
  '/Users/OscarEvertsson1/Programming/alex/worker/executor/tasks/variant_calling/__init__.py',
  '/Users/OscarEvertsson1/Programming/alex/worker/executor/tasks/variant_calling/variant_calling.py']
```

Figure 7.2: Output of the tool where the modules imported are marked as certified

```
The following modules are NOT certified but imported by something certified
[ '/Users/OscarEvertsson1/Programming/alex/worker/executor/tasks/resistotype/resistotype.py',
  '/Users/OscarEvertsson1/Programming/alex/worker/executor/__init__.py',
  '/Users/OscarEvertsson1/.pyenv/versions/3.7.2/lib/python3.7/logging/__init__.py',
  '/Users/OscarEvertsson1/.pyenv/versions/3.7.2/lib/python3.7/os.py',
  '/Users/OscarEvertsson1/.pyenv/versions/3.7.2/lib/python3.7/abc.py',
  '/Users/OscarEvertsson1/.pyenv/versions/3.7.2/lib/python3.7/_py_abc.py',
  '/Users/OscarEvertsson1/.pyenv/versions/3.7.2/lib/python3.7/_weakrefset.py',
  '/Users/OscarEvertsson1/.pyenv/versions/3.7.2/lib/python3.7/stat.py',
  '/Users/OscarEvertsson1/.pyenv/versions/3.7.2/lib/python3.7/posixpath.py',
  '/Users/OscarEvertsson1/.pyenv/versions/3.7.2/lib/python3.7/genericpath.py',
  '/Users/OscarEvertsson1/.pyenv/versions/3.7.2/lib/python3.7/re.py',
  '/Users/OscarEvertsson1/.pyenv/versions/3.7.2/lib/python3.7/enum.py',
  '/Users/OscarEvertsson1/.pyenv/versions/3.7.2/lib/python3.7/types.py',
  '/Users/OscarEvertsson1/.pyenv/versions/3.7.2/lib/python3.7/functools.py',
  '/Users/OscarEvertsson1/.pyenv/versions/3.7.2/lib/python3.7/collections/__init__.py',
  '/Users/OscarEvertsson1/.pyenv/versions/3.7.2/lib/python3.7/_collections_abc.py',
  '/Users/OscarEvertsson1/.pyenv/versions/3.7.2/lib/python3.7/operator.py',
  '/Users/OscarEvertsson1/.pyenv/versions/3.7.2/lib/python3.7/keyword.py',
  '/Users/OscarEvertsson1/.pyenv/versions/3.7.2/lib/python3.7/heapq.py',
  '/Users/OscarEvertsson1/.pyenv/versions/3.7.2/lib/python3.7/lib-dynload/_heapq.cpython-37m-darwin.so',
```

Figure 7.3: Part of the output of the tool where the modules imported are not marked as certified

of working. Concepts such as OOP prescribe thinking about what functionality is similar and structure the code based on that. The other alternative would be problematic if you, for example, wanted to move functionality from certified to non-certified. If the certified and noncertified code is completely separate meaning they might not even be in the same repository, it could become problematic of moving functionality between them. It could also lead to different teams recreating functionality for the certified and noncertified code in the long run. If you instead keep the functionality that is related close in the codebase and then categorize it based on certification level the risk of recreating functionality is most likely lower. This would probably also make refactoring of something that previously belonged to certification to now belong to a noncertified component easier since they are similar in functionality.

One of the weak points of components is that they are unsuited for fine-grained and crosscutting features (Apel et al., 2013). This was something that during development became clear especially for functionality that was previously defined as

"*common*" where a lot of unrelated but commonly used functionality was located. By refactoring this functionality it ended up in two separate components where one is certified and the other one not. Compared to other methods of separation this might not be the most optimal way to solve this.

Another problem with using components like this is that it can be hard for developers to distinguish between certified and noncertified code when they are developing and importing functionality. Since the imported component is not different in any way except the *___init___.py* file the developer will not notice if the component is certified unless they manually check for it. However, there are solutions such as the tool made as a proof of concept that could be used before merging new changes to a master branch. For example, it is not uncommon that developers use pull requests[2] in their daily workflow. If you for every merge of a pull request add the tool as part of your continuous integration you could get a warning saying that some components that are imported are not marked as certified. This would then aid the developer in making sure that no noncertified code is imported from a certified context.

Finally, the usage of components and marking them in the *___init___.py* file as certified and noncertified could be extended to not only working for true or false. The file could instead contain something like level: A, B and C. These level would then be precisely the same classification as some standard such as IEC 62304 which use A, B and C. This is currently not a need for 1928 Diagnostics but is something that they have mentioned could be necessary for the future.

[2]https://www.atlassian.com/git/tutorials/making-a-pull-request

8

Iteration 4: Effects on BAPO

This chapter presents the evaluation of the constructed mockup of the chosen variability approach with regards to BAPO. A description of the mockup can be found in Chapter 7. This chapter answers research question 3: *How does the chosen variability approach affect the organization's BAPO?*.

8.1 Result

The evaluation of the mockup has been done through a focus group with three persons from 1928 Diagnostics and two moderators. The persons who attended the focus group from 1928 Diagnostics were the head of QA, the CTO and the COO. More information about the focus group can be read in Chapter 4.

8.1.1 Business

The focus group found it hard to imagine whether the changes if done to the whole codebase would affect the development pace in the general case. If there is any difference they believed that it would be a minor positive effect. However, they believed that the mockup resulted in a more flexible codebase in the sense that functionality could be reused to a larger extent than previously. Due to this flexibility, they also thought that it would give them the possibility to develop and release individual pipelines easier which is a competitive advantage.

The focus group thought that it would be easier to create a certified product because of how these architectural changes done with the mockup could aid risk management, constructing artifacts, tests and verification. Risk management would be easier since now it is more clear what code is included in a certified product than previously. The same reasoning was the motivation for constructing artifacts, tests and verification, since everything related to one component is located inside it, it is easy to keep track of. Except that it could aid in constructing a certified product, they did not see any difference in if it would change the development pace for either certified or noncertified products.

The focus group did not think that the mockup would change their business plan. Mainly since one of the reasons for 1928 Diagnostics to take part in this research

was to be able to gain a competitive advantage when constructing certified and noncertified products. Therefore, even if the mockup resulted in positive effects it does not change the business plan.

The focus group did not think that they would be able to attract more customers by tailoring the product to a specific market. Instead they saw a potential need when it came to tailoring the product to certain customers. Customization possibilities could, in this case, be a hospital with their specific patient medical journals. This type of tailoring could lead to more customers. 1928 Diagnostics is however still a quite small company, and because of the company size it could be difficult at this stage to have that customization.

A potential positive effect on the development cost is something the focus group believed would be achieved in the long run, but not instantaneously. The main benefit would be that developers will not have to try to determine themselves if code belongs to a certified product or not. It would also aid the process since the developers know which process to follow and that would probably decrease the cost for the certification.

8.1.2 Architecture

The main thing the focus group mentioned with regards to code quality improvements was that the mockup achieves clear segregation between certified and noncertified code. They also mentioned that the codebase is now more decoupled which gives them more flexibility for reusing functionality.

The focus group did not think it was easier to find code compared to previously but also noted that they have a relatively small codebase of python code with around 15 000 lines of code. They believed that it most likely would be easier to locate code with the changes made if the codebase was bigger.

With the mockup, the focus group found it easier to see how things are connected, which is something they anticipated due to the nature of separation into components. They also found that the changes achieved the intention of more easily being able to differentiate between which code belongs to certification and not than before.

8.1.3 Process

To construct a certified product there are a set of tasks (such as risk management) which are required to be performed during development. The focus group thought that the mockup could be of great assistance for these tasks. As shown in Chapter 7, the mockup can be extended with tooling which could help developers during development. The focus group also thought that it would be easier for developers to work in the codebase since they now can differentiate whether they are working with certified or noncertified code.

Because of the segregation of the certified and noncertified parts, the focus group

thought that the development could be easier when working with the certified parts than previously. This is due to the fact that a developer clearly knows which process to follow and the possibility of using tools which can be created because of the mockup. One of the largest benefits of the mockup is the aid that it can give to the process through tooling.

The focus group said that they now easily could extend the components with related artifacts. These artifacts could then be used to auto-generate the *technical documentation* belonging to a certain component which is used for meeting regulatory requirements (Force, 2011). Being able to auto-generate technical documentation would be a major benefit for 1928 Diagnostics. If a component contains the artifacts necessary for certification of that component, it could be used in different products and save a lot of time of certifying those products. This is something the QA director mentioned he had never seen before and that he thought could be extremely useful.

The focus group did not think that solely the architecture changes could help to distribute the workload for certification but that the architecture could be a good basis to allow it for the process later on.

The focus group did not think that the mockup would affect the communication between employees either positively or negatively. They believe that it depends more on the individuals rather than the architecture, but this might be different for larger organizations. However, they did believe that the mockup could trigger more questions or dialogues between the roles related to QA and developers if they are working on the certified code.

8.1.4 Organization

The focus group did not think that there will be any new roles introduced as a result of the mockup. Neither did they see that the changes would be related to the possibility to scale up the organization right now, or that the team structure would change as a result. Neither did they believe that the mockup generally would affect the organizational part of the company. However, they mentioned that it might be different for a large company with several teams and a larger codebase.

Even though the focus group believed that the mockup would have a low impact on the organizational aspect they thought that developers could feel more responsible for working with certified code over noncertified. In addition to this, and the previous discussion, the focus group mentioned that components could be marked with other information using the same method as for safety. The focus group mentioned security which could result in developers being extra careful in making changes. Also, if changes are done, adding more verification to make sure that the certified code fulfills the requirements could be achieved by tooling.

8.2 Discussion

It is interesting to see that 1928 Diagnostics believes that the mockup will affect many parts of the company. It is clear that 1928 Diagnostics believe that some parts had a larger impact than others and it is clear that the mockup had less impact on the organizational part compared to the business, architecture and process parts. However, since the mockup is purely a test and not something that has been used by 1928 Diagnostics and studied longitudinally, the effects are speculations with the exception of the architecture were the changes can be observed in the mockup.

1928 Diagnostics mentioned throughout the focus group that the effects on the organizational and business aspect might have been different if the case company would have been larger. At the moment the development team at 1928 Diagnostics is a very flexible and cross-functional team and everyone talks to each other. If a company would have had several development teams the component architecture might have affected the team coordination more and also made it potential impacts on the business more obvious.

One of the largest potential impacts the mockup has brought is the possible aid it can bring to the development- and certification process. If a component could be reused not only in the sense of its functionality but also support reuse for a set of artifacts which is necessary to comply with a standard to achieve certification, the impact could be extremely positive. This would mean that 1928 Diagnostics would not only get a competitive advantage by reuse of functionality that a regular SPL would give (see Chapter 2). The company would also get a competitive advantage from the reuse of artifacts which can be very time consuming according to the QA Director of 1928 Diagnostics.

Another important factor mentioned by the focus group is that developers can see if the code they are working on is certified. If this would be combined with tooling which checks if certified code is changed such as the proof of concept presented in Figure 7.3, this could lead to their software becoming safer since the developers know that they are working on certified software. Since the developers are aware that they are working on certified code they would know to follow a certain process which is required to achieve certification (a safer product). Even if developers would forget to follow a certain process, tooling could then check if artifacts associated with safety certification have been changed or notify the developers that a certain process needs to be followed such as during pull requests.

That the focus group believes that the development pace for creating new pipelines would increase is something that was expected. This is due to the fact that it was one of the goals of using a software product line technique. As mentioned in the background chapter, when implementing software product lines the development pace which relates to the development cost goes up since the code can be reused to a larger extent.

9

Iteration 5: Decision Support for Selecting Variability Approach

This chapter introduces a methodology which describes how a company or organization can use the results of the previous result chapters to decide which variability approach would fit their software system the best. Since all software systems and organizations have different requirements and systems, it will differ which variability approach that fits them the best. This chapter answer the final research question: *Can a methodology be constructed to derive the most suitable variability approach for differentiation between certification levels? If so, how?.*

9.1 Results

To find the most suitable variability approach, six steps have been created. An overview of the steps needed can be found in Figure 9.1.

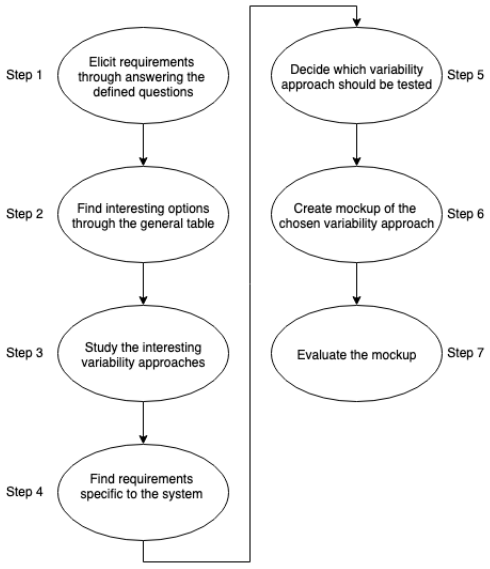

Figure 9.1: An overview of the steps in the methodology

9.1.1 Step 1: Elicit the Requirements

To be able to select a suitable variability approach the requirements need to be known. To gather the requirements from the stakeholders' elicitation is done through a set of questions which can be found below. We recommend that at least three types of roles are involved in answering these questions; a technical leader, a QA person and a person who has general responsibility and future vision for the system.

It is fine not to know the answers to a question. However, the more questions that can be answered the more favorable the result will be in narrowing down the most suitable variability approach. If a particular question cannot be answered it is important to understand why; it could be that another person with a different background needs to be included in the elicitation or that some angle needs to be investigated further.

- (1) Do you prefer migrating into using the variability approach in smaller steps rather than doing it all at the same time? (Yes, No)
- (2) To which degree do you need many new features which are similar to existing ones (if there is any)? (Low, Medium, High)
- (3) To which degree do you need adding new features which are not similar to existing ones (if there is any)? (Low, Medium, High)
- (4) Do you need to remove dead code because of optimization or customer requirements? (Yes, No)
- (5) To which degree is it important that the codebase is uniform? (Low, Medium, High)
- (6) To which degree is it important that the code is reused to a high degree? (Low, Medium, High)
- (7) To which degree is it important to separate code depending on the concern? (Low, Medium, High)
- (8) To which degree is it important to have code abstraction? (Low, Medium, High)
- (9) Do you need a specific granularity? (Line, Method, File)
- (10) To which degree do you want to work iteratively? (Low, Medium, High)
- (11) To which degree is it important to be able to see where a feature is implemented? (Low, Medium, High)
- (12) Do you need to label artifacts other than code? (Yes, No)
- (13) Do you need a specific binding time? (Compile-time, Load-time, Run-time)
- (14, 15) Do you want the approach to be natively supported in a programming language or can it be an extra tool? (Language-based, Tool based)

- (16, 17) Do you want the possibility of having the code for a specific feature in multiple locations or all code in one place? (Multiple places, One place)
- (18) Do you need that the code's control flow differs depending on whether the code which calls it certified or not (use case 1)? (Yes, No)
- (19) Do you have certified and noncertified code that ran the same code (the control flow does not change), and thus the need to mark the code whether it is associated to a certified product or not (use case 2)? (Yes, No)

When these questions have been answered the first requirements should be known.

9.1.2 Step 2: Narrow Down the Variability Approaches

Using the requirements gathered in step 1 and knowledge about the variability approaches the goal with this step is to narrow down the number of suitable variability approaches. Table 5.1, from Chapter 5, describes the attributes of each variability approach. Using this table as a starting point, Table 9.1 has been created. In this table only the variability approaches that can be used for differentiating certified and noncertified code are presented together with their attributes. In this table additional information about whether a variability approach can work for the two different use cases have been added at the end. To mention the use cases again they are:

- **Use Case 1: Altering control flows:** The code's control flow differs depending on whether the code which calls it certified or not.
- **Use Case 2: Usage of the same code**: A certified product and noncertified product run the same code (the control flow does not change). The code then needs to be marked whether it is associated with a certified product or not.

You should now think about which use case your company or organization need for, if not both.

	Parameter	Design Patterns	Components	Version Control	Pre-processors
Business related					
(1) Transition in steps possible	Yes	Yes	Yes	Yes	Yes
(2) Ease of adding new features similar to existing ones	High	High*	High	High	High
(3) Ease of adding new features not similar to existing ones	Medium	Medium	Medium*	High	Medium
(4) Support removal of dead code	No	No	No	No	Yes
Architecture related					
(5) Uniformity	Low*	High*	Medium*	Low*	Low*
(6) Reusability	Low	High*	High	Low	Low
(7) Separation of Concerns	Medium*	High*	High*	Low*	Low*
(8) Information Hiding	Medium*	High*	Medium*	No*	No*
(9) Granularity	Line*	Line	File*	Line*	Line, Method, File*
Process related					
(10) Preplanning effort	Low*	Medium*	Medium*	Low*	Low*
(11) Feature Traceability	Low*	High*	High*	Low*	Medium*
(12) Artifact support	No*	No	No	Yes*	Yes*
Organization related					
-					
Dimensions of variability					
(13) Binding time	Run, load	Compile, Load, Run	Compile	Compile	Compile
(14) Language based	Yes	Yes	Yes	No	No
(15) Tool based	No	No	No	Yes	Yes
(16) Annotation based	Yes	No	No	No	Yes
(17) Composition based	No	Yes	Yes	No	No
Use Cases					
(18) Can be used for use case 1	Yes	Yes	No	No	Yes
(19) Can be used for use case 2	No	Yes	Yes	Yes	Yes

Table 9.1: Relating the quality attributes to the variability approaches that can be used to differentiate certified code and noncertified code.

To find the approaches that could be suitable one should map the answers from step 1 to the data in Table 9.1. To be able to map the questions and the rows in the table there are numbers associated with each question and quality attribute. This should ideally be done by the same persons as in step 1.

The narrowing down can be exemplified: for a certain system it became clear from step 1 that there is a need to migrate to using the variability approach in smaller steps and that the binding time needs to be run-time. When looking in Table 9.1, this concludes that there only are two approaches available: parameter-based and design-patterns.

9.1.3 Step 3: Study the variability approaches which are left

From the last step some variability approaches should have been removed since they do not fit the requirements. The variability approaches left should now be studied in order to ensure that the decision makers have the necessary foundation to make an informed choice. If other persons who do not study the variability approaches should be involved someone should hold a brief presentation for them on the subject. One can study the material in this thesis background chapter or look at referenced books in this thesis.

9.1.4 Step 4: Other Important Quality Attributes

Table 9.1, shows information about each variability approach which is general to all software systems. However, for some quality attributes it might differ how well a particular variability approach works for a certain system. In Table 6.1 some other quality attributes are described, but in contrast to Table 5.1 the data in the table is specific to 1928 Diagnostics. For example in Table 6.1 there is an attribute to how well a particular variability approach fits the backend. How well it fits will of course differ depending on how a system's backend is set up.

This step focuses on answering how well the variability approaches conform to these attributes of your system and also if there are any other attributes which might be vital for your system. Questions have been created which should be answered. As in step 1 and 2 we recommend that the same roles are involved in filling in the new table. These were a technical leader, a QA specialist and a person who has general responsibility for the system and future vision of the system.

- To which degree is the variability approach compatible with the backend? (Low, Medium, High)
- To which degree is the variability approach compatible with the frontend? (Low, Medium, High)
- To which degree is the variability approach easy to work with? (Low, Medium, High)

- How long will the transition time be for the variability approach? (Low, Medium, High)
- Are there any other attributes that are important for a variability approach specific to your system?

When the questions have been answered, the answers should be compiled into a table similar to Table 6.1. Use this table to narrow down the variability approaches that could be suitable for your system.

9.1.5 Step 5: Decide Variability Approach

The previous steps focused on finding out which variability approaches that potentially could fit the system by using different quality attributes in the tables used.

The outcome of this step is to choose one variability approach that should be used in the company's or organization's system. To be able to choose one variability approach there needs to be a discussion, and we recommend that the same persons attends this discussion as in the previous iterations. The discussion could preferably be done through a focus group. The group should then decide which approach is the most suitable for the system. Here the pros and cons of each approach should be discussed. When discussing the pros and cons it could be helpful for the participants to see Table 9.1 and the table created from step 4 and have these as a base point.

In the case of uncertainty of the effects of the variability approach the following two steps (6 and 7) can be taken to test and evaluate the variability approach on a smaller piece of the codebase.

9.1.6 Step 6: Mockup (optional)

If there is uncertainty about how well the selected variability approach will fit in practice, a mockup can be created to test the selected approach from step 5. Since it is a test, a part of the system's codebase needs to be selected, in which the implementation will be tested on. The selected part should be a good representation of the whole system and the problem that needs to be solved. The selected part then needs to be understood in depth and whether the code is certified or not needs to be known. This can be done by analyzing function calls and existing documentation. With the needed background information, the implementation of the mockup can be created.

9.1.7 Step 7: Evaluate (optional)

After implementing the variability approach, it could be useful to evaluate the potential effect. A focus group could be assembled with at least the same people as in the previous steps. To evaluate the mockup with regards to a company's or organization's BAPO the questions in appendix A.1 can be used. If some other aspect,

other than the BAPO model, could be evaluated, then create questions regarding that area.

9.2 Discussion

The methodology introduced in this chapter can be used by anyone who wishes to differentiate code with different certification levels. This methodology is, therefore, a general contribution to the research area of software engineering. This methodology aids the user since he or she does not need to look into the literature and since some attributes and data are new conclusions that have raised by discussion.

However, there are of course possible risks of using the methodology. There is a risk that the data in table 5.1 which is not marked with a star (*) could be wrong since it has not been found in the literature. Another possible risk is that some variability approaches have been left out of the table or that new techniques have been created.

Something that is mentioned in step 4 is that the user needs to find if there are any other attributes of the variability approach which are not in the tables. Finding other attributes could be hard and there is a risk of missing essential aspects for the specific company or organization. It is essential to mention that the evaluation in step 7 does not need to focus on BAPO. The user can also create other types of questions to evaluate the mockup with another focus, but then the questions in the appendix probably are not of any use.

10

Conclusion

The purpose of this study was to look at how an SPL could, in the long run, be adopted in an agile development process and derive multiple products where some products require certification and others do not. The study was conducted using an action research methodology at the company 1928 Diagnostics to answer the following four research questions:

> **RQ 1:** Which variability approaches for implementing an SPL can be used for differentiating between certified and noncertified code?
>
> **RQ 2:** In the context of 1928 Diagnostics, what is the most suitable way to derive multiple products from a shared codebase with different certification levels?
>
> **RQ 3:** How does the chosen variability approach affect the organization's BAPO?
>
> **RQ 4:** Can a methodology be constructed to derive the most suitable variability approach for differentiation between certification levels? If so, how?

To answer **RQ 1** we went through SPL literature and looked closely at eight SPL variability approaches. Out of these eight, we concluded that in theory five of these could support differentiating between certified and noncertified code. Those are: parameter, design patterns, version control, and preprocessor and components. This finding could be useful for companies similar to 1928 Diagnostics but also to companies who want to derive certified and noncertified products. The finding is also relevant to research where an initial step is taken in how different variability approaches could be used to differentiate certified and noncertified code. The researchers had not found any literature that had done this review in a systematic way before.

RQ 2 was answered by taking the five variability approaches we concluded could work in theory and narrowing them down using a focus group at 1928 Diagnostics. The result of this focus group was that components were the most suitable variability approach for 1928 Diagnostics. As mentioned in 3, Vaccare Braga et al. (2012) created a metamodel where they connect components (features) to a specific certification level. Even though their metamodel contain a lot of information that this thesis have not focused on, this thesis have now shown that it is possible to connect components with a specific certification level and use these to create different

products.

Research question **RQ 3** was resolved by implementing a mockup using components of a minor part of the 1928 Diagnostics code base and then evaluating the mockup by conducting a focus group with the QA director, CTO, and COO of 1928 Diagnostics. The thesis concludes that potential effects on 1928 Diagnostics would primarily be related to the architecture and the assistance the mockup could provide to the process later. Based on how the components were implemented future tooling could be positive to minimize the extra time spent to work with a certified product. The conclusion is also to some extent that the business will be affected by a potential minor increase in development pace and an easier possibility to tailor end products to individual customers. For both **RQ 2** and **RQ 3** the result is specific for 1928 Diagnostics, however should it be applicable to companies with a similar organizational size and system.

Finally, the significant outcome of this thesis is through answering **RQ 4**. The thesis shows that it is possible to construct a methodology to derive the most suitable variability approach and how. The methodology consists of five mandatory steps and two optional steps. Step one is to elicit the requirements for the specific organization/company. Step two continues by using the result from **RQ 1** to find suitable variability approaches. The methodology then advances by looking at more specific requirements for the company/organization. The necessary final step is then to select the most suitable variability approach. This methodology can be used for companies/organizations that need to differentiate between certified and noncertified code and a goal to construct an SPL. To the researchers knowledge there has not been any similar research done before where one selects the most suitable variability approach based on their properties. In most of the literature they only mention that they had selected a variability approach, but not why or how. The papers neither mention which other alternatives they had.

10.1 Future Work

This study is merely looking at one case on how different variability approaches could support a separation between different certification levels. This specific case is within the medical device industry, but seeing as several other fields involve certification the findings may be applicable in these as well. Investigating this research at these other industries that also share the issue of working with different certification levels would be very interesting, this to see whether there are any differences in the results or not. This is especially crucial for the methodology described in chapter 9 and the answer to research question 1.

Another aspect that could be interesting to dive further into is the frontend. The mockup constructed at 1928 Diagnostics only looks at code related to the backend where the most suitable variability approach might not be the same as for the frontend. A hypothesis that comes as a result of our discussions with developers at 1928 Diagnostics is that it could be harder to find a suitable variability approach that

works well with frontend for the general case. Pleuss et al. (2012) look at usability and software product lines without the aspect of certification. They present that there is a dilemma between automation and usability using software product lines. Adding on top of their research the factor of multiple certification levels could, therefore, be problematic for something that is already difficult.

The variability approaches that this thesis presents that could work for separation between certification levels are not necessarily all that exists. To not miss options for separation between certified and noncertified it could be valuable to look closer at the variability approaches. There might be variability approaches that support additional ways to separate. Additionally, there might even be variability approaches that this study looked at that do support separation but that this research did not manage to find.

A natural step forward from looking at how the separation of certification levels can be achieved is to look at how a development process could be constructed to enable working with both certified and noncertified code.

References

Abdelaziz, A. A., El-Tahir, Y., & Osman, R. (2015). Adaptive Software Development for developing safety critical software. In *2015 international conference on computing, control, networking, electronics and embedded systems engineering (iccneee)* (pp. 41–46). IEEE.

Alberts, B., Johnson, A., Lewis, J., Raff, M., Roberts, K., & Walter, P. (2002). *Molecular biology of the cell* (4th ed.). New York: Garland. Retrieved from `https://www.ncbi.nlm.nih.gov/books/NBK26917/`

Apel, S., Batory, D., Kästner, C., & Saake, G. (2013). *Feature-Oriented Software Product Lines: Concepts and Implementation.* Berlin, Heidelberg: Springer Berlin Heidelberg.

Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., ... Thomas, D. (2001). Manifesto for agile software development. Retrieved from `http://www.agilemanifesto.org` doi: 10.1177/0149206308326772

Cawley, O., Wang, X., & Richardson, I. (2010). Lean/agile software development methodologies in regulated environments–state of the art. In *International conference on lean enterprise software and systems* (pp. 31–36). Springer.

Debray, S. K., Evans, W., Muth, R., & De Sutter, B. (2000). Compiler techniques for code compaction. *ACM Transactions on Programming languages and Systems (TOPLAS)*, *22*(2), 378–415.

Díaz, J., Pérez, J., Alarcón, P. P., & Garbajosa, J. (2011). Agile product line engineering—a systematic literature review. *Software: Practice and experience*, *41*(2), 921–941. doi: 10.1002/spe

Flyvbjerg, B. (2006). Five misunderstandings about case-study research. *Qualitative inquiry*, *12*(2), 219–245.

Force, S. G. . o. t. G. H. T. (2011). *Summary Technical Documentation (STED) for Demonstrating Conformity to the Essential Principles of Safety and Performance of In Vitro Diagnostic Medical Devices.* Retrieved 2019-05-06,

from http://www.imdrf.org/docs/ghtf/archived/sg1/technical-docs/ghtf-sg1-n063-2011-summary-technical-documentation-ivd-safety-conformity-110317.pdf

Hunt, J. (2006, oct). *Aspect oriented programming with Java.* Retrieved from https://www.theregister.co.uk/2006/10/26/aspects{_}java{_}aop/

Kasauli, R., Knauss, E., Kanagwa, B., Balikuddembe, J. K., Nilsson, A., & Calikli, G. (2018). Safety-Critical Systems and Agile Development: A Mapping Study. , 470–477. Retrieved from http://arxiv.org/abs/1807.07800 doi: 10.1109/SEAA.2018.00082

Kernighan, B. W. (1988). *The C Programming Language* (2nd ed.; D. M. Ritchie, Ed.). Prentice Hall Professional Technical Reference.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., & Irwin, J. (1997). Aspect-oriented programming. In *European conference on object-oriented programming* (pp. 220–242). Springer. Retrieved from http://link.springer.com/10.1007/BFb0053381 doi: 10.1007/BFb0053381

Liebig, J., Apel, S., Lengauer, C., Kästner, C., & Schulze, M. (2010). An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the 32nd acm/ieee international conference on software engineering-volume 1* (pp. 105–114). ACM.

Nickerson, R. S. (1998). Confirmation bias: A ubiquitous phenomenon in many guises. *Review of general psychology*, *2*(2), 175–220.

Organization, W. H. (2007, May). *Medical Devices.* Retrieved from https://www.who.int/medical_devices/definitions/en/

Pleuss, A., Hauptmann, B., Dhungana, D., & Botterweck, G. (2012). User interface engineering for software product lines. In *Proceedings of the 4th acm sigchi symposium on engineering interactive computing systems* (p. 25). ACM. doi: 10.1145/2305484.2305491

Pohl, K., Böckle, G., & van Der Linden, F. (2005). Software product line engineering: foundations, principles and techniques. In *Systems and software variability management.* Berlin, Heidelberg: Springer Berlin Heidelberg.

Python Software Foundation. (2019). *The import system.* Retrieved 2019-05-22, from https://docs.python.org/3/reference/import.html{#}namespace-packages

Robson, C., & McCartan, K. (2016). *Real world research.* John Wiley & Sons.

Runeson, P., & Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, *14*(2), 131–164. doi: 10.1007/s10664-008-9102-8

Shenton, A. K. (2004). Strategies for ensuring trustworthiness in qualitative research projects. *Education for information*, *22*(2), 63–75.

Shull, F., Singer, J., & Sjøberg, D. I. (2007). *Guide to advanced empirical software engineering*. Springer.

Statista. (2019). *Internet of Things (IoT) connected devices installed base worldwide from 2015 to 2025 (in billions).* Retrieved 2019-02-04, from `https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/`

Stringer, E. T. (2013). *Action research*. Sage publications.

Vaccare Braga, R. T., Trindade Junior, O., Castelo Branco, K. R., De Oliviera Neris, L., & Lee, J. (2012). Adapting a Software Product Line Engineering Process for Certifying Safety Critical Embedded Systems. , 352–363.

van der Linden, F., Schmid, K., & Rommes, E. (2007). *Software Product Lines in Action* (Vol. 41) (No. 6193). Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved from `http://link.springer.com/10.1007/978-3-540-71437-8` doi: 10.1007/978-3-540-71437-8

Yin, R. K. (2003). Case study research: design and methods.

A

Appendix

A.1 Questions to evaluate BAPO

A.1.1 Business

- Compared to your old architecture setup, do you think this new setup will accelerate or decrease your development pace?
  - Is there any difference between certified and non-certified products?
  - How could this affect your business plan?
  - What changes would you make?
- Do you think you would get more or fewer customers if you could tailor the product to a specific market? (for example for different countries or only create a 100% c-marked product)?
- What effects do you think the new architecture will have on constructing new products?
  - Are there any differences between the effects on certified and non-certified products?
  - If so, which?
- Do you think the new architecture will affect the development cost?
  - If so, how and how much?
- Are there any other benefits regarding the potential benefits or disadvantages you would like to add regarding the business aspect?

A.1.2 Architecture

- Compared to the old architecture, how would you say that the code quality is affected?
  - Comparing the old and the new architecture, do you see any difference in how difficult it is to find the code that you are looking for?

  - Comparing the old and the new architecture, do you see any difference in how difficult it is to understand how the code is connected?
- With the changes done, is it easier or harder to know which code is certified and which is not?
- Are there parts of the code which you did not know was used by the certified parts?
- Are there any other benefits regarding the potential benefits or disadvantages you would like to add regarding the architecture aspect?

A.1.3 Process

- Do you think the new architecture will make it harder or easier to create new products?
  - If so, why?
- Do you think the new architecture will make the certification process during the development harder or easier?
  - If so, why?
- Do you think the new architecture will make the certification process before a release harder or easier?
  - If so, why?
- With the new architecture, do you think it is easier or harder to distribute the workload for tasks that are involved in the certification to more people?
- Do you think that communication between employees will become better or worse with the changes made?
- The new architecture makes it possible to connect tests with features in code, how would this affect your developing pace and process?
- Are there any other benefits regarding the potential benefits or disadvantages you would like to add regarding the process aspect?

A.1.4 Organization

- With the new architecture, is there a need for any new roles in the teams/organization?
  - If so, which?
- With regards to the organization would the changes made be an aid or problematic with regards to scaling up your organization?
- With the new architecture would you change the setup of the team structure?

  - If so, how and why?
- With the new architecture in place, would it be easier or harder to establish more teams?
- With the new architecture the developer easily can know if a file is certified or not. Do you think the general communication between QA and development will become better, worse or unaffected with this information?
  - Why, why not?
- If developers were aware that they are working with a certified product, do you think that they will feel more or less of a responsibility for the certified product?
  - Why, why not?
- Are there any other benefits regarding the potential benefits or disadvantages you would like to add regarding the organization aspect?

A.2 Proof of concept tool

Listing A.1: Proof of concept for checking imports to be certified

```
import sys
import os
import pprint

from modulefinder import ModuleFinder


def main(args):

    # Check input.
    ok, file_path, dir_path = input_ok(args)
    if not ok:
        return

    # Get all subdirectories and add them to
    sys.path to not get any missing modules.
    path = get_path(dir_path)

    # Create modulefinder and run it
    finder = ModuleFinder(path)
    finder.run_script(file_path)

    # Make a dict of packages that are certified
    cert_package_set = get_certified_packages(finder, dir_path)

    check_certified(finder, cert_package_set)


def check_certified(finder, cert_package_set):
```

```
    all_good = True
    pp = pprint.PrettyPrinter(indent=2)

    # Check if there were modules that modulefinder could not resolve.
    not_found_modules = finder.any_missing()

    if len(not_found_modules) != 0:
        all_good = False
        print(
            bcolors.get_red_color_string(
                "The following modules were
                    not found by modulefinder which means you can not trust this result"
            )
        )
        pp.pprint(not_found_modules)

    # No matter if we resolved modules above it might be interesting to
    see which ones were ok and which was not.
    ok_files, not_certified_files = _check_certified(finder, cert_package_set)
    if len(not_certified_files) != 0:
        all_good = False
        print(
            bcolors.get_red_color_string(
                "The following modules are NOT certified but imported by
                    something certified"
            )
        )
        pp.pprint(not_certified_files)

    if ok_files:
        print(bcolors.get_green_color_string("
        The following files are certified and ok:"))
        pp.pprint(ok_files)

    if all_good:
        print("All good!")


def _check_certified(finder, cert_package_set):
    ok_files = []
    not_certified_files = []
    for name, mod in finder.modules.items():
        file_path = mod.__file__

        # There might be modules that ModuleFinder could not resolve
        if file_path:
            file_dir = get_directory(file_path)
            if file_dir not in cert_package_set:
                # Not certified but imported, append to non_certified_list
                not_certified_files.append(os.path.realpath(file_path))
            else:
                # Append files that are ok files (they are certified)
                ok_files.append(os.path.realpath(file_path))
    return ok_files, not_certified_files
```

```
def get_certified_packages(finder, dir_path):
    cert_package_set = set()
    cert_package_set.add(dir_path)

    for name, mod in finder.modules.items():
        file_path = mod.__file__
        if file_path and '__init__.py' in file_path:
            certified = check_if_certified(file_path)
            if certified:
                package_dir = get_directory(file_path)
                cert_package_set.add(package_dir)

    return cert_package_set


def input_ok(args):
    if len(args) != 2:
        print("Expecting two argument, file path followed by directory")
        return False

    file_path = args[0]
    dir_path = args[1]

    if not os.path.isfile(file_path):
        print("Not a file, pass me something real.")
        return False

    if not file_path.endswith('.py'):
        print("Cmon, keep it real, give me a python file")
        return False

    if not os.path.isdir(dir_path):
        print("Invalid directory")
        return False

    file_path = os.path.realpath(file_path)
    dir_path = os.path.dirname(dir_path)

    return True, file_path, dir_path


def check_if_certified(file_path):
    with open(file_path) as f:
        for line in f.readlines():
            if 'certified=True' in line.replace(" ", ""):
                return True
    return False


def get_path(file_dir):
    path = sys.path

    for i, j, y in os.walk(file_dir):
        if '__pycache__' not in i:
            path.append(i)
```

```
    return path


def get_directory(file_path):
    last_slash_index = file_path.rfind('/')
    return file_path[0:last_slash_index]


class bcolors:
    HEADER = '\033[95m'
    OKBLUE = '\033[94m'
    OKGREEN = '\033[92m'
    WARNING = '\033[93m'
    FAIL = '\033[91m'
    ENDC = '\033[0m'
    BOLD = '\033[1m'
    UNDERLINE = '\033[4m'

    def get_red_color_string(text):
        return bcolors.FAIL + text + bcolors.ENDC

    def get_green_color_string(text):
        return bcolors.OKGREEN + text + bcolors.ENDC


if __name__ == '__main__':
    main(sys.argv[1:])
```