UNIVERSITY OF CALGARY

An Agile Framework for Variability Management in Software Product Line Engineering

by

Yaser Ghanam

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE

DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

JULY, 2012

UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the faculty of graduate studies for acceptance, a thesis entitled "An Agile Framework for Variability Management in Software Product Line Engineering" submitted by Yaser Ghanam in partial fulfilment of the requirements of the degree of Doctor of Philosophy.

Supervisor, Frank Maurer, Department of Computer Science

Robert Davies, Department of Electrical and Computer Engineering

Jonathan Sillito, Department of Computer Science

Internal/External Examiner, Richard Levy, Faculty of Environmental Design

External Examiner, Robert Biddle, Carleton University

Date

Abstract

During the past few years, research in agile product line engineering has been gaining more popularity, driven by the much needed ability to combine the flexibility and high responsiveness of agile methods with the economic advantages of reuse and mass customization offered by software product lines. This dissertation presents a novel framework to manage variability in software product lines in an agile context.

By leveraging agile practices such as iterative and incremental development, test-driven development, and refactoring, this dissertation shows that a reactive approach to variability management is indeed feasible. The findings of this research demonstrate that acceptance tests can play an important role in variability elicitation; but they may not be sufficient to deduce implicit constraints from requirements. This issue is addressed by using executable acceptance tests alongside feature models in order to uncover implicit constraints and hidden dependencies. The dissertation also discusses the role of executable acceptance tests in supporting the evolution of variability by providing instantaneous feedback on the impact of adding or removing features or variants. For requirements that cannot be adequately described using acceptance tests such as usability and portability requirements, the dissertation demonstrates how such requirements can be treated using a lightweight and reactive approach.

At the implementation level, the results of this research show that realizing variability can occur in a reactive manner provided that proper refactoring and testing practices are followed. The results also illustrate how the process can be made more systematic by using tests as a common starting point to inject variability on-demand. The efficiency of the process can be improved by providing automated tool support. Once variability has been realized in the system, the dissertation discusses how individual products can be built using the derivation technique or the instantiation technique.

Finally, the dissertation presents important findings on the issues and challenges likely to arise when adopting a new software product line framework in an industrial context. The findings reveal a number of technical challenges, but also bring to surface non-technical issues related to the business needs, the organizational context, and a raft of human factors.

"In the name of Allah, the most gracious, the most merciful"

Acknowledgements

All praise and gratitude are due to Allah, God Almighty, for giving me the strength to rise out of the most difficult circumstances to finish the journey I have started; and for teaching me how to nurture patience with persistence, how to be strong but compassionate, how to be confident yet humble, and how to be a good leader but also a good follower.

A special thank you to Dr. Frank Maurer for being a mentor, a friend and a source of inspiration; for giving me advice, support and guidance without pushing too hard; and for being so patient and understanding all the way. Without your guidance and compassion, this journey would have been much harder.

Many thanks go to Alberta Innovates Technology Futures for funding my doctoral studies. I also would like to thank people in the Smart Home Program at TRLabs-Calgary, especially: Jennifer van Zelm, Dr. Rainer Iraschko, and Dr. Robert Davies for offering valuable collaboration opportunities.

To my family, friends and colleagues, thank you all for being there when I needed you and for believing in me at every step of this journey. To Armin Eberlein, thank you for providing me with the opportunity to come to Canada and realize a dream of a lifetime.

Dedication

To my dear parents, Ammar and Fadwa:

For all your devotion and hard work throughout my life, and for pushing me to go above and beyond expectations. All that I am now or ever hope to be, I owe to both of you.

To my kind and loving wife, Maha:

For always believing in me, and for taking the best care of our son when I am too distracted with work. I love you, not only for what you are, but for what I am when I am around you.

To my son, Ibrahim:

For bringing a lot of happiness and joy to my life, and for welcoming me at home with the cutest little smile after a long day of work.

To my brothers and sisters in Syria, Egypt, Tunisia, Yemen, and Libya:

For sacrificing their lives in the fight for freedom and democracy so that we and future generations may live with dignity. May Allah bless your souls and grant you Paradise.

Table of Contents

List of Tables

List of Figures and Illustrations

List of Listings

CHAPTER ONE: INTRODUCTION

Agile product line engineering is a new area of research that has been gaining momentum throughout the past few years. The general goal of the research in this field is to combine two notions, namely: agility and reuse. Agility as manifested in software paradigms such as Agile Software Development (ASD) provides the nimbleness needed to cope with changing requirements; while reuse as practiced in paradigms like software product line (SPL) engineering provides economic advantages through a "build-once-use-many-times" strategy.

1.1 Roots

In its early days, software development was approached using models that were analogues to those in other engineering disciplines such as civil engineering. As a result, the sequential waterfall model [Royce1970] was the defacto standard in software development until it was found to be unrealistic and incapable of coping with the fast pace and special needs of software projects [Kruchten2004].

During the past two decades, great advancements have been made in software development pertaining to iterative and incremental development models, which changed the typical lifecycle of software products. In 2001, the Agile Manifesto was declared as an umbrella under which many of the iterative approaches came together to cherish and promote certain values and principles [Agile2010]. The four major principles behind ASD were defined as: individual and interactions are valued over processes and tools; working software is valued over comprehensive documentation; customer collaboration is valued over contract negotiation; and responding to change is valued over following a plan. The agile community has brought forward a multitude of practices and methods to support such principles. Agile methods generally focus on short iterations driven by a strong relationship with the customer and a continuous feedback throughout the lifecycle of the project. ASD handles uncertainty differently from traditional approaches (i.e. waterfall) by investing the bare minimum in the early stages of requirement analysis and system design. Instead, ASD emphasizes the delivery of working software on a regular basis to solicit early feedback and mitigate any risks [Highsmith2001]. Iterative models in general proved to be more realistic and effective in achieving fast delivery to obtain customer feedback and incorporate this feedback in the following cycles of development [Bittner2006].

Another dimension of advancement in software development has been software reuse [Jacobson1997]. Instead of building software products from scratch, assets that were produced in previous software projects should be enhanced and reused. Reuse offered a great potential in terms of improving productivity and quality [Mili1995]. Software reuse opened the door for many research directions such as design patterns [Gamma1995], component-based software engineering [Lau2004], and SPLs [Clements2001]. An SPL is a family of software-intensive systems that share a common set of features while allowing for a margin of variability to satisfy different customer needs [Clements2001]. Effectively, SPLs achieve reuse at the product level rather than the component level. That is, groups of components are reused together in a prescribed way to build whole products. SPLs deal with similar systems as a family of products sharing a library of core assets. But since customer requirements are rarely exactly the same, shared assets have to accommodate a certain degree of variability (aka. customizability). Commonality between systems is what makes SPLs economically viable; whereas variability is what makes mass customization possible. Companies consistently report that SPLs yield significant improvements [Sharp2000, Linden2007, Bergey2004, Brownsword1996]. Some reported reductions in the number of defects in their products and cuts in costs and time-to-market by a factor of 10 or more [Schmid2002]. Having said that, it is important to realize that there is an adoption barrier to the SPL practice in its traditional form [Kruger2002]. The amount of up-front investment needed to build the reusable assets and get the SPL to a profitable stage is tremendous – which goes strictly against the core principles of ASD as will be explained next.

1.2 Iterative Reuse & Variability

Having realized the advantages of ASD as an iterative paradigm and SPL engineering as a reuse paradigm, it is interesting to determine whether the two can be combined to achieve reuse and variability while maintaining the main themes of agility such as iterative development and minimum up-front design. As rewarding as this sounds, combining ASD and SPL practices in a single work environment seems to be a complicated predicament, and even counter-intuitive. On the one hand, traditional SPL engineering approaches put a strong emphasis on up-front domain analysis and architecture design. On the other hand, ASD does not applaud such practices; and instead, proponents of agile methods preach a 'just-enough' philosophy in which actual coding activities start as soon as enough requirements are gathered to fill a short iteration.

The research question I address in this dissertation is whether it is possible to achieve reuse and variability through SPL engineering in an environment where ASD practices are common. If we can manage to wisely reconcile the conflicts between the philosophies of the two paradigms, we will be able to amplify production capabilities without compromising agility. The next section explains the different dimensions of the problem at hand in more detail.

1.3 Problem Statement

So far, the focus of ASD methodologies has been to develop software systems that satisfy a specific customer base, without worrying about best practices to handle variations of requirements in the system. Recently, the agile community has been investigating ways to scale agile up to the enterprise level rather than the team level as in [Leffingwell2007] and [Shalloway2009]. This will eventually require that agile organizations adopt a paradigm that supports organization-wide reuse and enables the efficient handling of variability in the reusable assets.

Nonetheless, adopting traditional SPL reuse and variability management techniques within an ASD process is a challenge. For one, ASD fosters a culture of minimalism in up-front investment and process overhead including documentation. This is in direct conflict with traditional approaches to SPL engineering where a whole phase, namely domain engineering [Pohl2005], is dedicated up-front for domain analysis and variability management. This phase is often demanding and entails heavyweight processes and considerable overhead as will be explained later (in the Background section). Secondly, organizations that adopt ASD depend heavily on fast delivery as a mechanism for quick customer satisfaction and feedback, which is too difficult to achieve when a domain engineering phase is to occur before delivering any products. Thirdly, the flexibility to accommodate changes in requirements and new customer requests is an important characteristic of ASD. Strictly adhering to a domain engineering phase where the requirements are set for the next phases of development hinders such flexibility.

To summarize, reuse and variability in traditional SPL engineering are treated proactively by conducting an extensive domain analysis to understand the sources of variability and design for this variability. This proactive treatment does not stand well with the ASD philosophy of minimal investment in requirement and design up-front. Therefore, there is a need to investigate the feasibility of a reactive approach to variability management, which is the focus of this dissertation.

1.4 Dissertation Scope & Goal

Within the scope of this dissertation, reuse and variability management are very closely related given that variability occurs whenever a decision is made to reuse an asset in a context that is not identical to the original context of the reusable asset. Hence, from this point forward, I use the term variability management with the assumption that reuse is necessarily implied. If the old and new contexts are identical, variability becomes irrelevant.

The goal of this dissertation is to investigate whether it is feasible to treat variability management in a reactive as opposed to proactive manner in order to lower the adoption barrier to SPLs in agile environments.

Through a number of research enquiries, I address the different aspects of variability management, and I show how these aspects can be achieved by leveraging existing agile principles and practices. Although SPL engineering has areas other than variability management such as scoping (i.e. defining the production capabilities of the product line), this dissertation only focuses on variability management as the main area of interest.

1.5 Significance & Contribution

The significance of being able to establish and manage an agile product line is threefold. For one, it will allow organizations to adopt SPL practices in an incremental manner instead of a big-bang transition. Also, for organizations that cannot afford to put large investments into proactive domain engineering, a reactive agile product line framework can be a viable alternative to improve productivity and quality of software systems delivered to the customers. Thirdly, for circumstances when speculation becomes too risky such as emerging technology domains or volatile market conditions, adopting a SPL framework that is highly flexible to changing conditions is very rewarding.

This dissertation offers a number of contributions to academics and practitioners in the agile community as well as the SPL community. The contributions can be summarized as follows:

- A comprehensive literature survey of significant work in the area of agile product line engineering.
- An approach to elicit variability in business logic requirements and evolve feature models using test artefacts that are by-products of ASD.
- An approach to elicit variability in presentation and portability requirements and evolve variability profiles using lightweight analysis.
- A variability modeling technique that leverages executable test artefacts to provide better traceability and communicability.
- A test-driven approach to reactively and systematically realize variability at the code level.

- An approach to support the product derivation process using the extended feature modelling technique.
- An empirical investigation of the technical and non-technical challenges that agile organizations are likely to face when making the transition to a SPL strategy.
- A toolset that supports the contributions above by automating tedious and error-prone aspects of the framework.

1.6 Dissertation Outline

After the introduction chapter, I use chapter 2 to lay foundational background knowledge on topics relevant to this dissertation. In chapter 3, I present a literature review of related work. In chapter 4, I list my research questions, discuss my research methodology and provide a bird's-eye view of the major components of this research. In chapters 5 through 10, I present the studies I conducted to answer the research questions (listed in Chapter 4). And finally, I summarize the results and findings of my research and draw some conclusions in Chapter 11.

1.7 Terminology

In this section, I provide definitions for the terms frequently used throughout this dissertation.

Agile Software Development (ASD): a group of iterative software development methodologies that emphasize continuous customer involvement, autonomous teams, flexibility to change, and rapid and frequent delivery of working software [Agile2010]. (The terms "*Agility*" and "*being agile*" are used throughout the dissertation to reflect this definition).

Agile Organization/Environment: a software development organization/environment that exhibits the major characteristics of ASD as in the definition above.

Software Reuse: building software products using artefacts that were used in building other software products [Frakes1995].

Software Product Line (SPL): a family of software-intensive systems that share a common set of features while allowing a specific margin for differentiation to satisfy diverse customer needs [Clements2001].

Variability: the notion that the components constituting the software architecture may vary due to a range of factors including diverse customer needs, technical constraints, and business strategies.

Variation Point: an aspect in a certain requirement that can have multiple states of existence in the system. Typically, it reflects why a requirement may vary from one product to another. For example, a security requirement may vary due to the need for different "levels of security".

Variant: a state of existence of a certain requirement in the system. Typically, it reflects how a requirement may vary due to a certain variation point. For example, in a security requirement, the "level of security" can be one of three: moderate, high and very high.

Domain Engineering (DE): the first phase in traditional SPL engineering which entails the identification of the common features and the variability of a SPL, the derivation of a reference architecture, and the realisation of reusable components and their quality assurance [Pohl2005].

Application Engineering (AE): the second phase in traditional SPL engineering which entails the realisation of customised products by utilising the SPL variability and binding the variation points with the predefined variants or with variants developed especially for the customer.

Mass Customization: producing software that meets the individual needs of customers with near mass production efficiency [Tseng2001].

Unit: a small testable part of a system, typically realized as a function, procedure or a method.

Unit Test: a test that verifies the correctness of the behaviour of an individual unit or the interaction between units in a software system.

Test-Driven Development: a software development technique wherein the writing of tests occurs before the writing of production code.

Acceptance Test: a test conducted to determine whether or not a software system has satisfied a subset of its acceptance criteria [Melnik2006].

Executable Acceptance Test: a test that is automated to run (execute) against the system in order to test an acceptance criterion. English-like executable specifications are a specific instantiation of executable acceptance tests.

Refactoring: the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure [Fowler2004].

Customer: an entity that specifies the needed features in a given system, and decides on the acceptance of such features based on a set of acceptance criteria. This can typically be a single person, a group of people, or a market segment represented by an account manager.

Feature: a chunk of functionality that delivers value to the customer.

Feature Model: a representation of the requirements in a given system abstracted at the feature level. If represented using a tree structure, it can be referred to as a feature tree [Kang1990].

Requirement Traceability: the ability to describe and follow the life of a requirement, in both forwards and backwards direction (i.e. from the specification of abstract features to the realization in code and vice versa) [Gotel1994].

Software Module: a part of the system that encapsulates data and behaviour of a given entity, typically realized at the code level by a class.

Software Component: an independent and cohesive part of the system that is composed of a number of modules which effectively constitute a sub-system.

Smart Home System: a software system designed to enable the monitoring and controlling of a closed environment (e.g. home, office) by interacting with hardware devices connected to the system via a number of technologies (e.g. wireless protocols, central server).

CHAPTER TWO: BACKGROUND

This chapter provides the background necessary before proceeding to the subsequent chapters of this dissertation. The chapter covers a range of concepts including agile software development, software reuse, and software product line engineering. The depth of coverage of each topic is commensurate with the significance of the topic within the scope of the dissertation.

2.1 Agile Software Development (ASD)

ASD includes a collection of iterative software development methodologies that, according to the Agile Manifesto [Agile2010], give customer involvement and satisfaction the highest priority. Agile practitioners preach an iterative development approach that encourages values and practices such as stakeholder communication, early feedback from customer, test-driven development, short iterations, just-in-time design and continuous integration.

The field of software engineering has matured enough to realize that getting the customer requirements right is key to the success of any software project. This is why traditional software engineering approaches invest so much time at the beginning of the project life cycle to elicit these requirements, clarify any vagueness around them, document them and produce designs that attempt to satisfy them. However, given the high level of uncertainty of customer requirements at the beginning of the project and the frequent changes of the requirements throughout the lifetime of a software system, agile methods discourage large investments in up-front analysis and design. Big-design-up-front (BDUF) is seen by many agile practitioners as the antithesis of agility. Agile methods tackle requirements in a different manner through a number of fundamental practices. The following subsections cover some of these practices.

2.1.1 Iterative & Incremental Development

Before kicking off the development activities in a project, agile teams are naturally inclined to undergo an initialization phase in order to share a project vision, define a rough scope, and discuss high-level aspects of planning and estimation. However, agile teams do not spend but a few weeks in this phase before iterative development actually starts. Detailed requirements are only determined during development iterations and only for features that are part of the current increment. Requirements are elicited from the customer in the form of user stories [Cohn2004] and made more concrete by defining their acceptance tests (ATs) [Reppert2004]. These user stories are prioritized and assigned to releases, each of which includes a number of short iterations (typically two to four weeks). A number of user stories are implemented during the iteration; and at the end of the iteration, a working version (aka. increment) of the system is delivered to the customer. The customer gets the final say on how well those requirements were satisfied and what needs to be done in the next iteration in terms of new features, bug fixes, usability issues and other tasks as depicted in Figure 1. If ATs were defined, those tests are used as evaluation criteria.

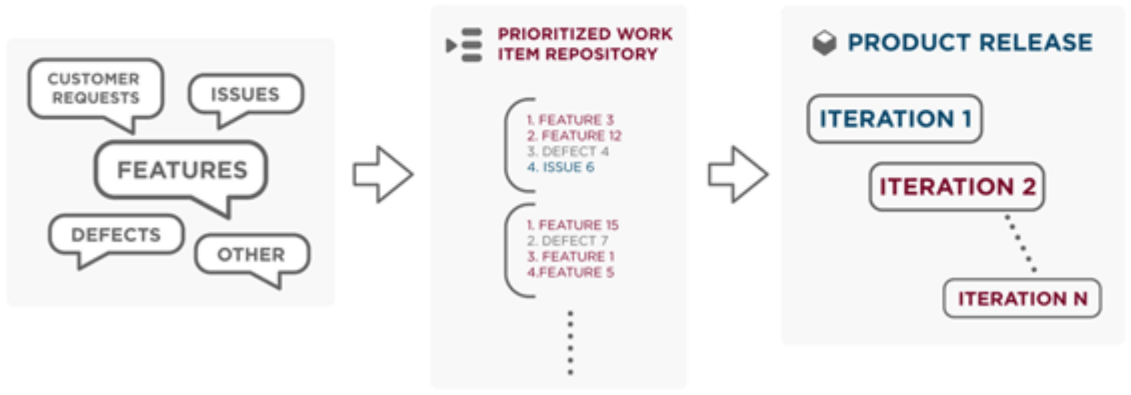


Figure 1 - Releases and iterations (adapted from [Agile2011])

The architecture of the system evolves gradually in a bottom-up fashion as the project needs become clearer. Design decisions are agreed upon by the members of the development team who talk to each other in their daily stand-ups.

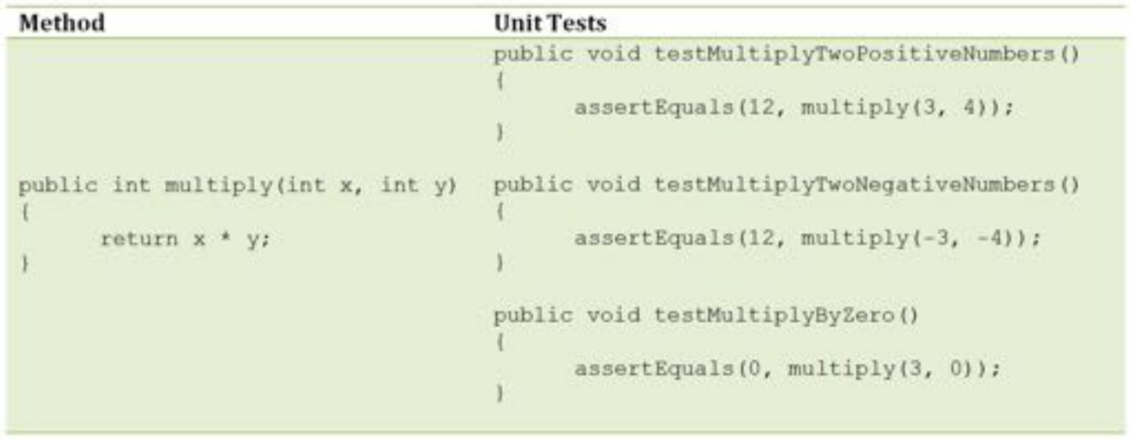
While scientific data on agile methods is not yet conclusive, they seem to work well according to a growing number of case studies, experience reports and controlled experiments investigating individual agile practices (e.g. business organizations are reporting success in adopting agile practices like Test Driven Development [Melnik2007b]).

2.1.2 Continuous Testing

Continuous testing is one of the main characteristics of iterative development which aims for early and less expensive defect detection [Highsmith2001]. Agile methods adopt this concept wholeheartedly by making tested and running software the primary measure of progress in an agile project. Ideally, a feature is not considered "done" until it has automated tests associated to it. In my work, I am specifically interested in two types of tests: unit tests and acceptance tests.

2.1.2.1 Unit tests

A unit is defined as a small testable part of a system. A unit test verifies the correctness of the behaviour of an individual unit, or the interaction between units. An example unit (method) along with its tests is shown in Figure 2. Agile methods encourage developers to test the code they write incrementally in a frequent and rapid code-and-test cycle. The lesser the time gap between coding and testing, the better. Extreme Programming (XP [Beck2004]) took this concept to “extreme” by promoting the idea of test-first development where a test-then-code cycle is repeated to ensure high test coverage and improve modularity [Beck2003]. Tests are automated to be executed frequently and to help in the refactoring of the code base.



Method

```
public int multiply(int x, int y)
{
      return x * y;
}
```

Unit Tests

```
public void testMultiplyTwoPositiveNumbers()
{
      assertEquals(12, multiply(3, 4));
}

public void testMultiplyTwoNegativeNumbers()
{
      assertEquals(12, multiply(-3, -4));
}

public void testMultiplyByZero()
{
      assertEquals(0, multiply(3, 0));
}
```

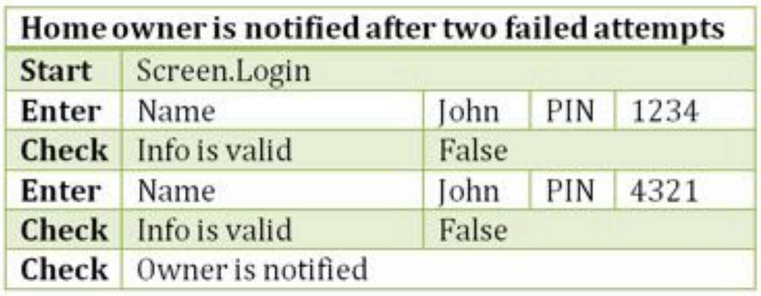
Figure 2 - An example of a unit (method) and its tests

2.1.2.2 Acceptance tests (ATs)

Requirement specifications – in their traditional format – exist in a number of documents and are written in a natural language. The correctness of the behaviour of a system is determined against these specifications using test cases or scenarios. On the other hand, executable specifications are written in a semi-formal language that aims to reduce ambiguities and inconsistencies. Executable specifications take various formats ranging from very formal [Fuchs1992] to English-like [FIT2010]. The English-like ones are often called scenario tests [Kaner2003], story tests [Kerievsky2010], or ATs [Perry2000]. These names highlight the role of these artefacts as:

1. Cohesive documentation of the specifications of a given feature.
2. Accurate, high-level validity tests: by being executable, these specifications can be run (executed) against the system directly in order to test the correctness of its behavior from the customer's perspective.

Throughout this dissertation, I will use the general term executable AT (EAT) to refer to the English-like specifications that can play the two roles above. EATs are usually produced in collaboration with domain experts (e.g. customer representatives) as acceptance criteria and feature specifications. Figure 3 shows an example of an EAT. If the behaviour of the system matches the expected one as specified in the EAT, the test passes. Otherwise, the test fails indicating either a technical problem in the code, or a business problem in understanding the specifications of the system. To link the EAT to actual production code, a thin layer of test code – called fixture – is used. EATs are usually executed using tools like FIT [FIT2010] and GreenPepper [GreenPepper2010].



Home owner is notified after two failed attempts				
Start	Screen.Login			
Enter	Name	John	PIN	1234
Check	Info is valid	False		
Enter	Name	John	PIN	4321
Check	Info is valid	False		
Check	Owner is notified			

Figure 3 - Executable AT

2.1.3 Refactoring

As defined by Fowler et al. [Fowler2004], refactoring is the "process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure." The main goal of refactoring is to improve the design of the code that has already been written so that it may become more readable, maintainable, testable, and reusable. Typical examples of refactoring include: extracting a method out of a code segment, moving fields from one class to another, renaming variables and methods, abstracting certain methods or classes, and restructuring code into a particular design pattern.

Refactoring ideally should not change the external behaviour of the code. This can be verified by running the tests that cover the code segment of interest before and after the refactoring process. If the tests failed after refactoring, then the behaviour has changed and the developers may need to retract the changes and investigate the cause of failure.

2.1.4 Agile organizations

Defining what "agility" is and what it entails has always been a controversial subject in the software community, both in academic and industrial contexts. Therefore, defining what constitutes an agile organization is rather challenging. In the context of this dissertation, I label organizations as agile if they exhibit an iterative and incremental development process, adhere to rigorous testing and refactoring practices, and lay emphasis on minimum investment in specifications and design up-front.

2.2 Software Reuse

Simply put, software reuse is the notion of building software products using artefacts that were used in building other software products [Frakes1995]. This definition has grown in complexity as the research area expanded. The definition now encompasses other aspects such as designing for reuse [Bieman1995], implementing reuse [Prieto-Díaz1996], managing reusable assets [Henninger1997], searching for and retrieving reusable assets [Frakes1994] and others. What is to be reused has also changed over time. Initially, code reuse was the main objective. Nowadays, reuse includes other artefacts such as design documents, use cases, test cases as well as processes and procedures [Mohagheghi2004].

The notion of reuse came later in the evolution of the software engineering field - because in the early days, with only a few software products available, it made sense to build new software from scratch. Nowadays, however, as millions of code bases are available world-wide, it is hard to claim that all parts of a new software system are novel. Some estimates suggest that 60% of the design of all business applications is reusable [Tracz1987], and only 15% of software code is unique in a given domain or organization [Joyce1988]. In practice, software systems in a given domain (e.g. learning management systems) solve similar problems; and therefore, there is a high potential for reuse. Even for software systems across different domains, similarities exist. Take the example of architectural layers concerned with operations at the operating system level such as file management.

2.2.1 Advantages of Software Reuse

At a first glance, the case for software reuse seems to be rather straightforward. If an organization can reuse existing artefacts to satisfy the requirements of new products, then the organization will potentially be able to gain a number of benefits [Sommerville1985, Jacobson1997, Joyce1988, Gaffney1989, Pohl2005] such as:

1. Deliver new products faster: because less time is spent on problem-solving and writing new code and test cases, development and testing cycles are both shortened.
2. Reduce development and maintenance costs: this is correlated with the previous point, because less time usually means less resource consumption and hence less cost. Also, if a given artefact is reused in multiple products, maintaining that artefact is cheaper than maintaining three individual ones.
3. Improve the quality of existing artefacts: the initial use of an artefact reveals bugs and flaws that can be resolved in future products. Also, if the reuse of existing artefacts is accompanied with quality reviews, it is likely that the number of defects will be reduced in that artefact as more products are built.
4. Reduce risks: reusing artefacts that proved to work as expected in other systems is associated with less risks compared to deploying newly developed ones.
5. Provide better project estimates: reusing existing artefacts to build parts of a new system reduces the uncertainty surrounding initial estimates of time and cost.

Jacobson et al. [Jacobson1997] provides real experiences of the benefits of software reuse. For example, Motorola reported increases in productivity by a factor of 10. Some other organizations reported reductions of up to 10 times in defect density and maintenance costs. Time to market was also reduced by a factor of 5. Cuts in overall development costs were anywhere between 15% and 75%.

2.2.2 Reuse Granularities

Code reuse can be achieved at different levels of granularities. The lowest level of reuse happens at the code fragment level when a developer copies a fragment of a method or a class to reuse it in their own context. Object-oriented programming provided a new level of reuse that has become widely common amongst practitioners [Mili1995]. Classes as a whole can be reused across different applications. Object-oriented programming has also enabled the concept of pattern reuse. Gamma et al. [Gamma1995] provided a handful of design patterns that enable the reuse of object-oriented solutions to solve similar problems. At a higher granularity, reuse may happen at the component level. Component-based software engineering [Heineman2001] is a technique for packaging software in a way that allows flexible reuse and composition of software. By mixing and matching appropriate components (or sub-systems), the users may be able to assemble a complete system [Roschelle1996]. Commercials-Off-The-Shelf (COTS) systems are a good example of this level of granularity [Morisio2002].

Reuse can occur at a very high level of granularity when a whole system is reused to solve a similar problem in a given domain. Reuse of systems is special, because it is an organizational (or team) decision rather than an individual decision. When the organization needs to develop a system that is similar to a previously developed one, this may be an opportunity to achieve a high level of reuse by considering the architecture of the existing system as a basis for the new one. This type of reuse is implemented differently in different organizations. Namely, it can be ad-hoc or systematic. If an ad-hoc approach is followed, the organization pursues this reuse opportunity by producing a copy of the code base of the existing system, tweaking it to meet the slightly different needs of the new system, and finally releasing it as a new product. This is often referred to as clone-and-own [Eriksson2005]. Depending on how much the systems share in common, this technique may actually provide huge savings compared to starting development from scratch.

However, as the demand increases for new similar systems, problems of ad-hoc reuse start to arise. With every new system, the organization adds another code base that will need to be maintained and updated separately. This is disadvantageous because maintaining configuration management branches (e.g. moving features between branches, finding redundancies) usually requires significant efforts [Sillito2007]. Moreover, if any update or bug fix takes place in the common part of these systems, the bug will need to be fixed in all the repositories. Also, as the systems evolve, their architectures start to deviate more from each other creating many separate development cycles – until they eventually are seen as completely independent products. In organizations where hundreds of customer-specific requirements are to be satisfied (e.g. Ericsson AXE [Jacobson1997]), ad-hoc reuse simply does not work.

Realizing the need for systematic reuse of systems, there has been a strong push towards a more planned approach. The idea was first formulated in 1976 by Parnas [Parnas1976] who coined the term program families. Later on, the term software product lines became more popular [Clements2001]. I dedicate the next section to elaborate on this concept.

2.3 Software Product Line Engineering

A software product line (SPL) is a family of software-intensive systems that share a common set of features while allowing a specific margin for differentiation to satisfy diverse customer needs [Clements2001]. The high level of reuse between systems in the product line makes SPLs economically effective; while the flexibility to accommodate certain variations within the reusable assets makes mass customization possible.

Experience reports indicate that the SPL practice can render significant improvements in software quality, cost, and time-to-market [Schmid2002]. To illustrate the idea, I will use an example of a smart home system.

2.3.1 Example: The Smart Home System

Choosing the smart home example stems from the fact that smart homes were the application domain through which I tested and validated the work in this dissertation*. Smart home systems serve as a prime example of the advantages of using a SPL practice combined with agile principles and practices.

A smart home system encompasses a number of features such as: Energy Manager, Security Manager, Media Controller, Automation Engine (for automatic control of lighting and other devices), Weather Watch and many others. Customers of smart home systems should be able to choose a subset of such features that fulfills their specific needs and wants. For instance, tech-savvy customers consider the Automation Engine to be essential; whereas novice customers do not trust automation and thus prefer not to have it. Furthermore, even within a single feature, it should be possible for customers to tailor certain aspects of the feature to satisfy their specific requirements. The Security Manager, for example, offers different techniques to secure access control such as PIN-protected locks, access by magnet cards and fingerprint authentication. When choosing to have the Security Engine, customers may select one or more of these options. As a result, different smart home systems may have different combinations of features as well as a number of possible variations within these features – a notion that is referred to as *variability*.

* This work was in collaboration with TRLabs – Smart Home project.

Variability can be business-driven when it is due to variations in the actual business logic or behaviour in the system like the examples mentioned above. Variability can also be presentation-driven. For example, monitoring and controlling a smart home can occur via different technologies such as a normal PC, a home media-center with a touch screen, or a mobile device. These different presentation media have different characteristics such as screen dimensions and touch capability – which will require developing different interfaces to suit the capabilities of each device.

One way to handle these variations in requirements is to build a separate system for each customer, and reuse what can be reused in an ad-hoc manner. However, using this approach increases the number of systems the organization will need to support and maintain. That is, if there were twenty customers with varying requirements, the organization would have to support and maintain twenty different systems (also twenty code repositories with, likely, a substantial overlap amongst them). The economic disadvantages of this approach are inescapable taking into consideration that all these systems are very similar to each other and that they only vary in the presentation layer or in a particular aspect of the business layer. Another approach is to support only one system and add configurations as needed without duplicating the code repository. This technique overcomes the redundancy problem of clone-and-own, but as the system evolves and more ad-hoc configurations are added, the complexity of the configurations and their different combinations start to grow beyond what can be managed and maintained efficiently. Without a system that defines how variations should be handled, these variations are likely to disappear into the code without an explicit representation that enables traceability and communication.

On the other hand, SPL engineering provides an interesting alternative to solve this problem. SPL engineering looks at all these similar systems as one family of applications that share a common base and are allowed to vary in a prescribed way. Variations and configurations are handled systematically and explicitly through variability management practices to control the complexity of the variations and communicate these variations clearly across the organization. In traditional SPL approaches, variability management is part of the up-front domain engineering phase which precedes all development of the actual applications. For the example above, this means that before starting to build and deliver any of the smart home applications to the customers, the organization should conduct a domain analysis to understand what is common between smart home applications, and what variations the smart home product line should support (e.g. variations in access control, variations in the UI). Moreover, the organization should design, build and test those assets that are anticipated to be reused in the applications such as the Security Manager.

After the domain engineering phase, the application engineering phase starts as engineers begin to assemble and tailor the reusable assets to deliver individual smart home applications. For instance, to deliver a particular application, application engineers need to bring together the Weather Watch module, the Media Controller, the Security Manager, and the Automation Engine. The Energy Manger was excluded because it was of no interest to the current customer. Based on the customer's preferences, within the Security Manger, application engineers will need to set the configurations so that only access by PIN is supported; and within the Automation Engine, the engineers will choose a lower level of automation. The systematic treatment of variability in the requirements is what sets SPL engineering apart from ad-hoc approaches. In the sections to follow, I will discuss systematic variability, domain engineering, and application engineering in more details.

2.3.2 Variability Definition

Variability in software systems refers to the notion that the components constituting the software architecture may vary due to a range of factors including diverse customer needs, technical constraints, and business strategies. According to the Orthogonal Variability Model (OVM) by Pohl et al. [Pohl2005], variability in a product line is described by a number of variation points, a set of variants for each variation point, and possibly some constraints. A *variation point* is an aspect in a certain requirement that can have multiple states of existence in the system. Each state of existence is called a *variant*. The selection criteria of variants might be governed by some *constraints*. For example, let's say that the Weather Watch feature consists of three aspects, namely: Weather Model, Weather Trend Analyzer, and Weather UI Panel. Both the Weather Model and the Weather UI Panel represent *mandatory* aspects without which the Weather Watch model would not be useful. On the other hand, considering a number of factors such as cost and computation capability, the Weather Watch may or may not have the Weather Trend Analyzer aspect which makes this aspect *optional*. Optionality can be expressed as a variation point for which two variants are defined:

Variation point **VP1 – Inclusion of the Weather Trend Analyzer**:

Variant **V1 - Trend Analyzer is included.**

Variant **V2 - Trend Analyzer is not included.**

Governed by the constraint:

Constraint **C1 - V1 and V2 are mutually exclusive**.

Besides optionality, when multiple alternatives could be selected for a given feature, this variability can also be expressed as a variation point for which there might be two or more variants. In the following example, depending on the type of hardware platform running the Weather Watch feature, two variants are defined - only one of which can be selected for a given system:

Variation point **VP2 - Panel type**:

Variant **V1 - Handheld panel.**

Variant **V2 - PC panel.**

Governed by the constraint:

Constraint **C1 - V1 and V2 are mutually exclusive**.

If more than one variant can be selected for a given feature, the constraint is defined using a minimum-maximum format. For example, in the Security Manager feature, the customer can select one, two or all three of the supported access control technologies - namely, V1: PIN-protected locks, V2: access by magnet cards, and V3: fingerprint authentication. Therefore, the constraint would be:

Constraint **C1 – [1..3] multiplicity imposed on V1, V2, V3**

Mutual exclusivity can also be described using a [1..1] multiplicity constraint.

Handling variability in the software family in a prescribed way is called *Variability Management*. This includes introducing new variation points and variants, updating existing ones, modeling variation points and variants, ensuring traceability and consistency between the model and all other artefacts in the system, implementing variations at the code level, communicating knowledge about customization possibilities to the different stakeholders, and deriving instances of the system based on the required configuration [Chen2009].

2.3.3 Variability Sources

Variability in a system can occur due to variations in the business logic requirements (aka. functional requirements) or due to variations in non-functional requirements. Business logic requirements describe *what* a system ought to do in terms of scenarios or workflows by a given user. On the other hand, non-functional requirements are attributes of the system that describe *how* a system ought to perform its functions in terms of usability, portability, security, and many other '-ilities' [Chung1999]. In other words, non-functional requirements capture the properties and constraints under which a system should operate [Antón1997]. To develop a quality software system, both functional and non-functional requirements are to be taken into account [Chung2009]. Researchers reported a number of issues that make non-functional requirements more difficult to elicit and manage than functional requirements. These issues include definition problems (i.e. what constitutes a non-functional requirement), and representation problems (i.e. where to document non-functional requirements) [Glinz2007]. This is mainly due to the fact that non-functional requirements are generally crosscutting concerns that do not necessarily belong to a single feature or another, but they are usually associated with the system as a whole. In the scope of this thesis, I only discuss two non-functional aspects, namely: presentation and portability.

2.3.4 Variability Modeling

Feature modeling has become an essential aspect of software engineering in general and SPL engineering in particular. A feature model is a representation of the requirements in a given system abstracted at the feature level [Riebisch2003]. A feature can be broadly defined as a chunk of functionality that delivers value to the end user. In SPLs, feature models represent a hierarchy of features and sub-features in a product line and include information about variability in the product line and constraints of feature selection. Throughout this dissertation, I use a common modeling technique called Feature-Oriented Domain Analysis (FODA) [Kang1990]. FODA was one of the earliest modeling techniques on which many other techniques were based (e.g. [Kang1998] and [Fey2002]). FODA models features in a given system in what is called a feature tree and uses a specific notation to describe variability as explained in the following example. Consider the simple feature tree in Figure 4a that represents the Weather Watch feature. Both the Weather Model and the Weather UI Panel are mandatory features, which is denoted with a solid line. On the other hand, as discussed previously, the Weather Trend Analyzer is optional, which is denoted with a dashed line. The Weather UI Panel can have one of two different formats depending on whether the application is to run on a handheld device or a normal PC. This is denoted in the tree as an arch. Unless otherwise stated, an arch represents a mutually exclusive constraint. By looking at this tree, one can deduce the different instances that can be produced from this generic system such as the ones shown in Figure 4b.

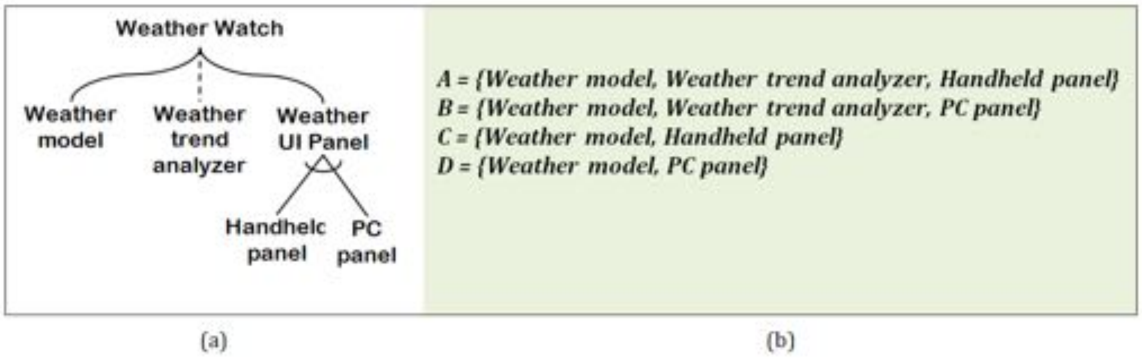

Weather Watch
Weather model
Weather trend analyzer
Weather UI Panel
Handhelc panel
PC panel
A = {Weather model, Weather trend analyzer, Handheld panel}
B = {Weather model, Weather trend analyzer, PC panel}
C = {Weather model, Handheld panel}
D = {Weather model, PC panel}
(a)
(b)

Figure 4 - Modeling variability

2.3.5 Domain Engineering & Application Engineering

The two major phases in SPL engineering are domain engineering and application engineering [Pohl2005]. During domain engineering, a comprehensive analysis is conducted to specify the scope, commonalities, and variations in the prospective SPL. Scoping is concerned with defining the limits of the family of products so that a clear cut is made between products that belong to the family and those outside the family. Commonality and variability analysis is concerned with determining the requirements of the members of the software family, and defining how these requirements may vary. This includes determining all sources of variation (i.e. variation points) as well as the allowed values (i.e. variants). Decisions have to be made on which artefacts are expected to be reused across different products, and how they should be designed in order to be reusable. Domain engineering also includes activities to build a reference architecture that has to be flexible enough to accommodate the predefined variations. This architecture is typically documented – along with all the requirement specifications – to disseminate to application engineers the required knowledge of how to use the architecture as a reference in the instantiation process of individual products. Moreover, the realization of reusable artefacts – as determined by the commonality and variability analysis – happens in the domain engineering phase alongside with their quality assurance measures. As depicted in Figure 5, the four stages in the domain engineering phase are requirement engineering, design, realization, and testing. The outcome of this phase is domain artefacts including the variability model.

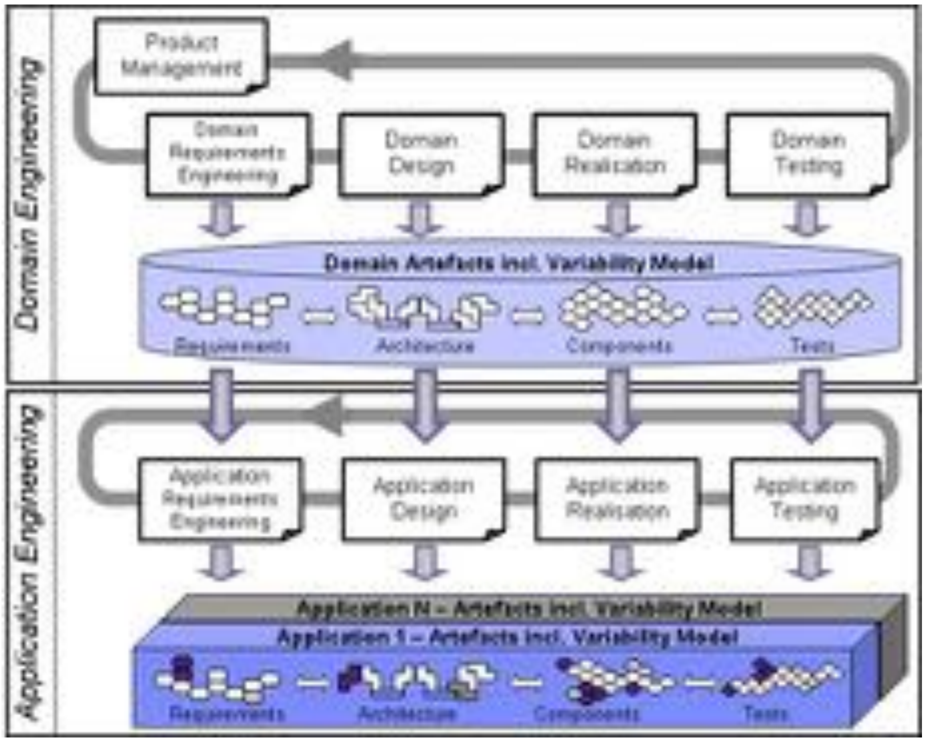


Figure 5 - Domain Engineering & Application Engineering (adapted from [Pohl2005])

After the domain engineering phase comes the application engineering phase. As a starting point, application engineers use the reference architecture, the reusable artefacts, and the variability profile – that were all defined in the domain engineering phase. Based on the specific requirements of a certain product, application engineers make decisions on what artefacts need to be included and what variants should be selected for each variation point. The outcome of this phase is an instance of the system that represents a specific product. If the reference architecture cannot accommodate the requirements of a certain product, or if the product cannot be derived from the currently available domain artefacts, application engineers ideally should provide feedback to domain engineers pertaining to the new requirements. As shown in Figure 5, the same four stages in domain engineering also exist in application engineering, but they produce individual product (aka. *application*) artefacts.

2.4 Product Derivation

Product derivation is the process of building individual products using a base set of assets. This process usually happens during application engineering as discussed above. Deriving products can be done using assembly approaches or configuration approaches as explained by Deelstra et al. [Deelstra2005]. Assembly approaches entail putting together a subset of the artefacts available in the product line to build unique products. This category of approaches is also described as extractive because the assets needed to build unique products are to be extracted from an existing asset base. On the other hand, configuration approaches involve providing a set of values to configuration parameters in the architecture to instantiate a unique product. These approaches are also called instantiation approaches [Bosch2000]. These approaches stem from basic concepts in configuration management; however, configuration management systems are generally not sufficient to support product derivation in a product line [Thao2008]. Configuration management systems do not natively provide mechanisms to maintain derivation relations between the core assets and derived products [Gurp2006]. This is mainly due to the concept of branching wherein a copy of an artefact is created and subsequently changed independently from its original – which causes unnecessary overhead when used solely to manage variability and derive products [Beuche]. Therefore, in this thesis, I use certain aspects of configuration management but in combination with common derivation practices that are based on variability management concepts.

2.5 Chapter Summary

In this dissertation, I use a number of software engineering concepts that were explained in this chapter. Reuse and variability can be handled in an ad-hoc way or a systematic way. Ad-hoc approaches pose a raft of challenges when the number of products starts to increase. SPL engineering resembles a systematic approach to manage variability across a number of systems in a given domain. Traditionally, a SPL process entails two sequential phases. The first is domain engineering in which requirements of the domain are collected and analyzed, variability is defined, and reusable artefacts are built and tested. The second is application engineering in which actual end-products are delivered to the customers after combining and customizing the reusable artefacts. SPLs are advantageous because they provide a systematic framework to handle variability, but at the same time they require large investments up-front and they are not sufficiently flexible in accommodating changing requirements. On the other hand, ASD practices such as iterative development, continuous testing, and refactoring are effective in supporting the evolution of software products in a fashion that allows high flexibility to accommodate unanticipated changes.

CHAPTER THREE: LITERATURE REVIEW

This chapter provides a comprehensive literature review that covers published research on areas relevant to the work presented in this dissertation. I cover the literature on related work in three steps. First, I survey existing research on combining ASD and SPL engineering in general. Second, I narrow down the focus of the review to discuss key research on variability management. Third, I cover key research on feature modeling and traceability; because a major component of the framework I developed is concerned with that aspect.

3.1 Agile Product Line Engineering

Research on combining ASD and SPL engineering is not abundant. This may be attributed to the fact that both paradigms in their current manifestations are relatively young. Although iterative and incremental development can be traced back to the 1970s or even earlier [Larman2003], it was only a decade ago when the Agile Manifesto [Agile2001] was declared. Since then, ASD has been increasingly gaining popularity and acceptance. Regarding SPLs, the concept itself goes back to the 1970s when Parnas [Parnas1976] proposed the notion of product families to manage non-functional variability [Linden2007]. However, the term SPL and its current practices were not introduced until the early 1990s [Linden2007]. A second factor contributing to the shortage of studies in this area is the seemingly contradicting philosophies and perceptions of the two paradigms in their respective communities.

One of the earliest attempts to bring together researchers and practitioners from both areas was the Agile Product Line Engineering Workshop in 2006 [Cooper2006]. The workshop aimed to discuss commonalities and points of variation between the two practices. The theme of the discussions in the workshop was around how feasible it was to integrate the two approaches. Other workshops on the same topic followed in 2008 [McGregor2008], 2009 [Ghanam2009b], and 2010 [Ghanam2010d].

In the following subsections, I discuss published work on combining ASD and SPL engineering under three main categories: utilizing both paradigms for different roles in a single organization, utilizing certain ASD practices in an already established SPL, and lowering the adoption barrier to SPL for companies that wish to adopt a SPL strategy.

3.1.1 Work on utilizing both paradigms for different roles in a single organization

Carbon et al. [Carbon2006] proposed an integration approach in which a typical domain-engineering-then-application-engineering process is followed. ASD plays a role only during the application engineering phase to tailor products for specific customer needs. In a later work, Carbon et al. [Carbon2008] looked at the issue of communicating feedback to domain engineers by application engineers. The authors suggested that the planning game be used where the application engineers play the role of the customer, and the domain engineers play the role of the developers to offer their feedback on the available core-assets. Hanssen et al. [Hanssen2008] presented a case study of a company where SPL practices and Agile practices were both utilized. The company used a SPL approach to manage their long-term strategic plans. For medium-term tactical concerns such as project development, the company used an ASD approach. The two paradigms were both present but not necessarily integrated. Navarrete et al. [Navarrete2006] conducted a preliminary analysis on how the Capability Maturity Model (CMM) process improvement guidelines could play a role in enhancing the maturity of a SPL, whereas ASD principles could play a role in enhancing the agility of a SPL.

While work in this category is interesting, it is different from the work I present in this dissertation. The main difference is that in my work I put more emphasis on an agile process that is tightly integrated with product line practices as opposed to giving each of the two paradigms a separate role.

3.1.2 Work on enhancing the agility of an already established SPL

McGregor [McGregor2008] presented an interesting theoretical attempt to reconstruct a hybrid method. In his article, he concluded that competing philosophies of ASD and SPL engineering make their integration difficult. But he asserts that the two paradigms can be tailored under the condition that both should retain their basic characteristics. McGregor identifies two ways to achieve the hybrid approach. The first is to use a skeletal SPL framework as a starting point, then add the quality of agility where appropriate and possible. The second is to start with an existing agile process and add product line qualities and concepts – especially for companies where an ASD is already in place. The majority of research efforts in the literature have so far focused on the first strategy. Therefore, this section surveys the literature on applying certain agile principles to enhance the agility of SPLs with the assumption that a SPL process already exists.

Kakarontzas et al. [Kakarontzas2008] discussed how Test-Driven Development could be used to enable the evolution of software components in a SPL. O’Leary et al. [O'Leary2007][O'Leary2010] proposed utilizing some agile practices such as agile planning games, early and continuous delivery, and automation to improve the product derivation process during application engineering. Noor et al. [Noor2008] proposed a process in which some agile principles such as stakeholder involvement, rapid feedback, and value-based prioritization can help in the planning phase of a product line. The authors proposed that a product map be structured at the feature level, domain level, and product level to facilitate the prioritization of features and the incremental development of the product. Taborda [Taborda2004] proposed the use of a release matrix in release planning to support the evolution and tracking of SPLs. Furthermore in the agile release planning area, Kurmann [Kurmann2006] identified release strategies to improve the agility of SPLs by synchronizing the platform release with the product release, and by releasing new features in a single product during the initial phase as opposed to integrating it fully with the product line to mitigate the risk of changing requirements. Trinidad et al. [Trinidad2008] suggested that there is a need for an automated support for feature model error analysis as a measure to achieve agility in a SPL. That is, for a SPL to become more agile, it needs to be open to change in the requirements, which according to the authors calls for an automated framework to detect errors resulting from such changes. Raatikainen et al. [Raatikainen2008] discussed the feasibility of employing the backlog management practice from ASD in SPL feature modeling. Feng et al. [Feng2007] were interested in the synergies of ASD and SPL engineering in the requirement engineering phase, and they developed a survey to collect expertise in agile requirements engineering for SPLs.

There are also some reported experiences of the use of ASD practices in established SPLs. For example, in a case study of a company called Salion, Clements et al. [Clements2002] reported the use of some agile practices such as daily builds and refactoring in a SPL-centered process. Mohan et al. [Mohan2010] contributed a second-hand analysis of the previous case study using complex adaptive systems (CAS) as an analysis framework. Moreover, Babar et al. [Babar2009] presented a case study of a company that used XP and Scrum methods within their SPL for several years. One of the issues that were found to be evident in the study was the need for up-front explorations. The authors assert that agile methods did not provide a mechanism for such explorations. While I agree that the need for up-front exploration is exacerbated in a product line context, I argue that agile methods in practice do accommodate this need by what is commonly called “a spike” or “iteration 0.”

The focus of the research efforts in this category is different from that of my research. The main difference is that I am not concerned with applying certain agile principles in a SPL process. On the contrary, I am interested in applying SPL principles in an ASD process.

3.1.3 Work on lowering the barrier to adopting a SPL process

Research efforts in this category assume that there is no SPL process in place, and aims to make adopting a SPL process easier for software organizations. In my research, I stress that for an approach to fit well with agile principles and practices, being incremental and reactive is key. By “incremental”, I exclude big-bang approaches that require most of the software assets and processes in the organization to be redefined and restructured in order to adopt SPL practices. And by “reactive”, I exclude proactive approaches in which a great amount of up-front speculation is required. The quest for an incremental and reactive approach to establishing and managing product lines is a relatively new phenomenon. For one, organizations did not want to throw away their investments in legacy systems and start all over again. Also, for many organizations the transition to systematically managed variability in their systems was too big a change if they were to follow the strict domain-then-application engineering model. Clegg et al.

[Clegg2002] proposed a method to incrementally build a SPL architecture in an object-orientated environment. The method provides useful insight into realizing variability in an incremental manner, but does not discuss other relevant issues such as how to communicate variability from the requirement engineering phase to the realization phase. Kruger [Kruger2002] proposed the idea that in order to lower the adoption barrier, domain engineering and application engineering should not be separate. As a result, he commercialized a tool that utilized the concept of separation of concerns to realize variability in software systems in a reactive manner. At the time of writing this dissertation, the tool was closed-source and not available for academic evaluation. Reactive approaches, with the support of tools like the one in [Kruger2002] has been reported to require orders of magnitude less effort compared to proactive approaches [Buhrdorf2003]. The aim of my work is somewhat similar to Kruger's. However, I differ in that I am not only concerned with realizing variability in a system. Rather, I am interested in a framework to manage the different aspects of variability as will be detailed later (e.g. understanding variability, modeling variability). The second major difference is that in my research I focus on contexts where ASD is common so that I can leverage existing agile practices to manage variability. This focus is somewhat similar to the work by Paige et al. [Paige2006] who proposed building SPLs using Feature Driven Development (FDD). They constructed an extension to FDD where two new phases were added: one for consideration of architecture and another for SPL component design. It is not clear, however, how much overhead is added by the proposed phases.

The spectrum shown in Figure 6 captures the position of my work within the existing body of literature. ASD lies at one extreme end of the spectrum, while SPL is positioned at the other extreme. This positioning is not intended to suggest that the two approaches are extremely contradictive, but rather to use the extreme ends as reference points for integration approaches. As seen in the figure, some approaches lean more towards the SPL end, because they effectively start form an existing SPL and then tailor certain processes to improve agility. In the middle are approaches that achieve a certain level of integration but not necessarily a complete assimilation. That is, both SPL and ASD practices are being used but at different levels or for different roles. My research (highlighted in red) leans more towards the ASD side because I start from an ASD process and then I introduce SPL notions and practices.

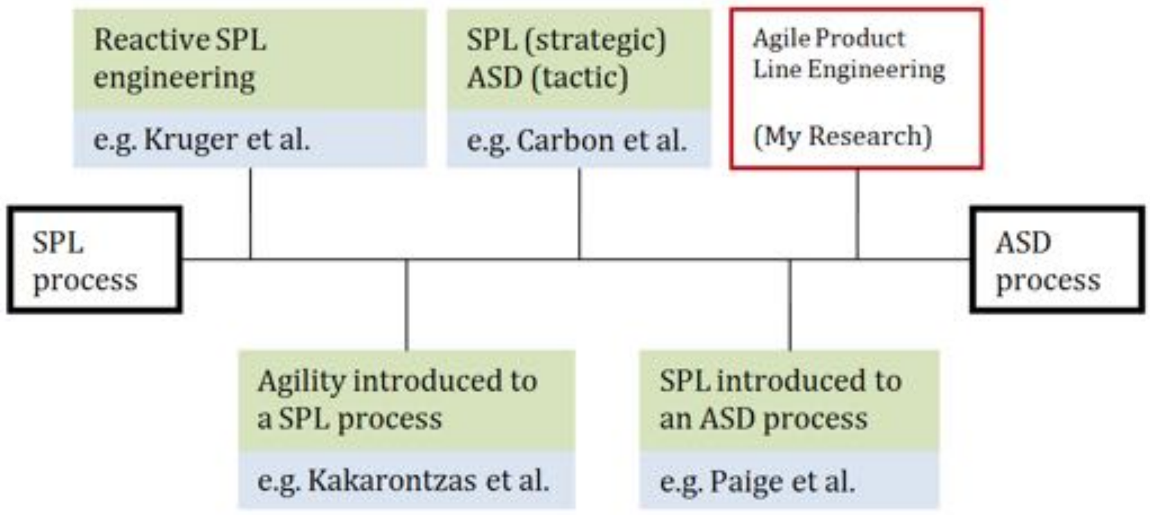


Figure 6 – Work in the literature

3.2 Variability Management

Variability management is a key concept in SPL engineering. There is a fairly large body of research investigating how to improve the analysis, modeling, and realization of commonality and variability in SPLs. The most common approach to variability management is called FODA, a short for Feature-Oriented Domain Analysis [Kang1990].

FODA aims at identifying features that define a given domain in terms of common, optional and alternatives aspects. FODA also provides feature modeling and definition techniques. The focus of FODA is the requirement engineering phase. Therefore, other approaches were later built to support the design and implementation phases such as FORM [Kang1998] and KobrA [Atkinson2000]. In my research, I use some aspects of FODA especially feature modeling using feature trees, and feature definition forms but I extend the idea by incorporating test artefacts to enhance traceability.

Buhne et al. [Bühne2004] proposed a variability approach (later called Orthogonal Variability Model or OVM [Pohl2005]) that was based on an explicit representation of variability through variation points and variants. To communicate variability to customers, Halmans et al. [Halmans2003] proposed extensions to use-case diagrams. In my work, I use OVM to describe variability profiles and I provide an alternative to use-case diagrams to communicate variability. Furthermore, Coplien et al. [Coplien1999] proposed an approach called FAST to identify, analyze and document scope, commonality and variability. In this approach, information is to be elicited and documented about the domain, the predicted commonalities, and the parameters of allowed variations. They also discuss concepts such as procedures, inheritance and class templates to realize variability in the code. Many other efforts were dedicated to the implementation of variability at the code level such as [Sharp2000b], [Anastasopoulos2009], and [Gacek2001]. I use some of these techniques in my approach for illustration and evaluation purposes.

Earlier variability management approaches – including FODA and KobrA – focused on specific phases of traditional software development such as requirement engineering and architecture design. Other techniques like FAST addressed variability at different phases but did not, however, consider the evolution of variability.

3.3 Feature Modeling & Traceability

There is a large body of research on feature modeling in software engineering in general, and SPL engineering in particular. As mentioned previously, FODA was one of the earliest techniques off which many other techniques were based. In my work, I use feature trees as described in traditional modeling techniques such as FODA.

Requirement traceability is the ability to describe and follow the life of a requirement, in both forwards and backwards direction [Gotel1994] (i.e. from the specification of abstract features to the realization in code and vice versa). Efforts to study traceability links between feature models and other development artefacts include the one by Filho et al. [Filho2002] in which they proposed the integration of feature models with the UML meta-model to facilitate the instantiation process. Another effort was the one by Ramesh et al. [Ramesh2001] in which use cases (representing requirements) were linked to design artefacts and from there to code artefacts. To group requirements at a more meaningful and comprehendible level of abstraction, Riebisch [Riebisch2004] suggested the use of feature models as an intermediate element between use cases and other artefacts. The main issue with this approach is that in real settings a massive effort is required to establish and maintain the traceability links due to the informal descriptions of the requirements – which made automation impossible [Pashov2004]. To solve the language informality issue, other techniques were proposed to establish traceability links. For example, Antoniol et al. [Antoniol2002] proposed an information retrieval method to link flat requirements to code artefacts. The caveat of the approach is that it is based on the hypothesis that programmers use names for program items (e.g. classes, methods, variables) that are also found in the text documents. There is also the issue of managing and maintaining the established traceability links. In a panel report, Huang [Huang2006] discusses the state-of-the-practice in traceability techniques. The report asserts that requirement trace matrices (RTMs) are often maintained either manually or using a management tool; and the amount of effort needed to keep these links up-to-date is enormous.

Commercial tools are available to support traceability. CaliberRM [CaliberRM2010], DOORS [DOORS2010] and other tools are used to manage and visualize traceability links. However, these links have to be established manually, and the tools do not address issues specific to feature models such as variability in requirement. Some SPL tools like pure::variants [Pure::Systems2010] provide add-ins to allow requirement models in traditional management tools to be remodelled as feature models. In my dissertation, I show how test artefacts can be used to extend feature models and enhance the traceability of variation points and variants in a SPL. To the best of my knowledge, this is a novel approach that has never been discussed in the literature before.

3.4 Chapter Summary

In this chapter, I discussed the work available in the literature in three different parts. The first part surveyed efforts to combine ASD and SPLs. The majority of the work done in this area focused on applying certain agile practices to an already established SPL. The purpose of my research, on the other hand, is to enable the adoption of SPL practices in agile contexts. The second part presented the literature on variability management as the key concept of interest in my dissertation. Finally, the third part discussed relevant literature on feature modeling and traceability. I listed traditional requirement traceability approaches, and I discussed attempts to achieve requirement tractability using feature models.

CHAPTER FOUR: RESEARCH APPROACH

In this chapter, I present the methodology I followed in my research. First, I explain the research goal. Then, I talk about research questions. And finally, I discuss my evaluation strategy. Details about specific research methods will be deferred to relevant chapters in the dissertation.

4.1 Research Goal

My research aims at investigating whether it is feasible to manage the different aspects of variability in a reactive as opposed to proactive manner in an environment where ASD is common. Therefore, the main goal of this research is to construct a *framework* for *agile organizations* to enable *systematic variability management* for similar software products. In my work, I try to leverage existing agile practices to build such a framework.

Throughout this dissertation, I use the following definitions for the terms used in the previous statement:

- **Framework**: a number of practices embraced within a defined process and supported by a set of tools.
- **Agile Organization:** a software development organization that exhibits the major characteristics of ASD.
- **Systematic:** provides for consistency of processes, practices, and conventions across the different projects and teams in the organization. Ad-hoc approaches are generally non-systematic.
- **Variability management:** handling variability in a software family in a prescribed way. This includes eliciting variability from requirements (i.e. variation points and variants) to build variability profiles, realizing variability at

the code level by introducing and evolving variation points and variants, modeling variability to communicate knowledge about customization possibilities to the different stakeholders, ensuring traceability and consistency between the model and other artefacts in the system, and deriving instances of the system based on the required customization [Chen2009].

4.2 Research Questions

Based on the goal of my research as stated in the previous section, and given the evaluation considerations as will be discussed in the following section, I divided the main problem into smaller problems that need to be tackled in order to build the proposed framework. The proposed framework, as shown in Figure 7, entails five main stages that repeat iteratively as more requirements come from existing or new customers. In the following sections, I give a general overview of each stage and the relevant research questions. I also shed a light on the research question pertaining to transferring this framework to an industrial context.

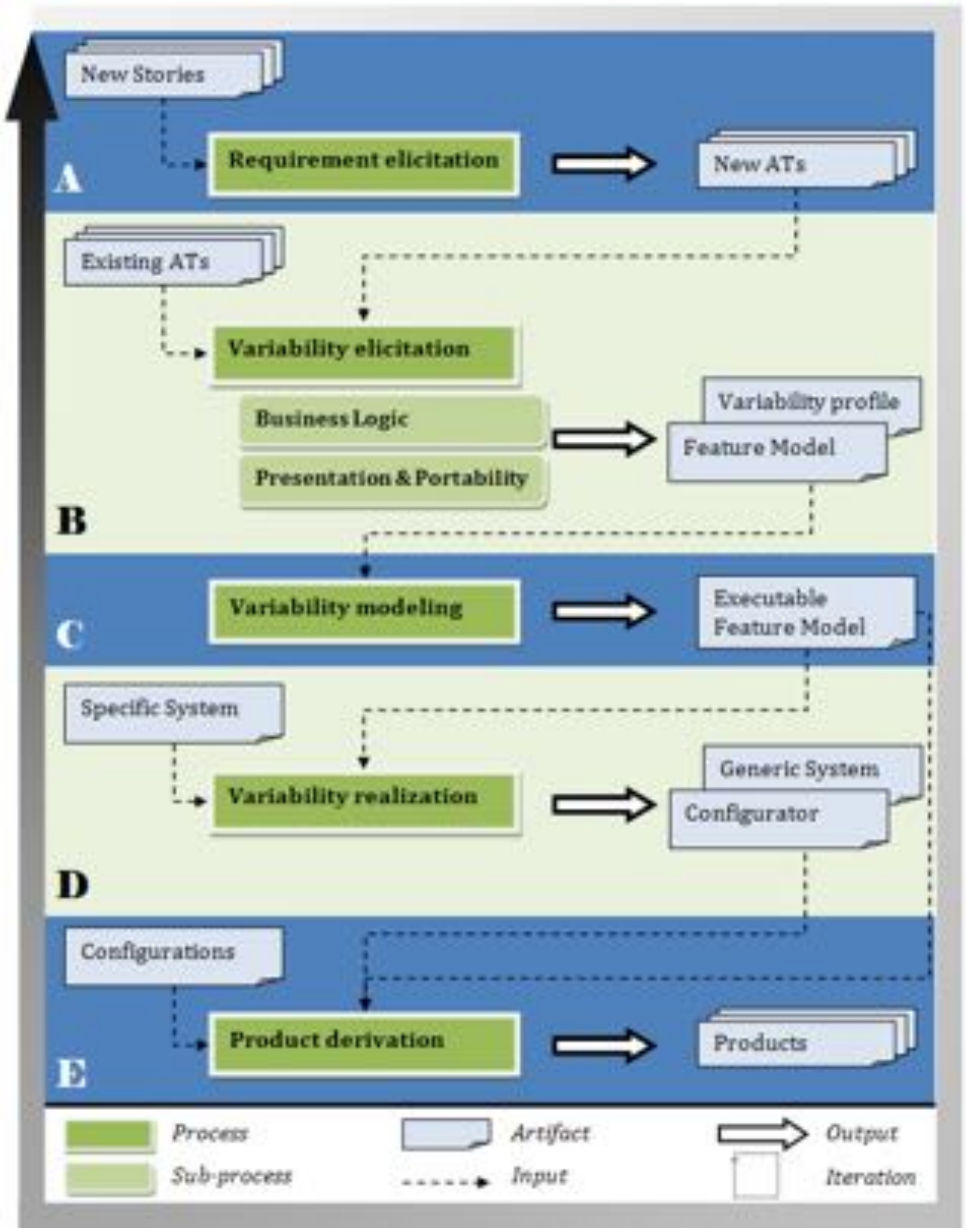


Figure 7 - The proposed framework

4.2.1 Stage A: Eliciting new requirements

This stage is no different from the normal requirement elicitation activities that occur in a typical ASD process as discussed in chapter two. At the beginning of every increment, new requirements are collected from the customer in the form of user stories [Kerievsky2010]. These user stories – in collaboration with the customer – are made more concrete by translating them into acceptance tests (ATs) [Reppert2004]. In the framework, I am mainly interested in the ATs that come out of this stage as input to the following stages. The ATs can be existing ATs that have already been used to implement the existing system, or they can be new ATs that are specified to guide the implementation of new features in the system.

4.2.2 Stage B: Variability Elicitation

The purpose of this stage is to elicit variability from the available requirements in an evolutionary and lightweight manner. This is traditionally done using a proactive documentation-intensive approach. But given that in agile contexts documentation is limited, I use ATs as an alternative. Also, a reactive as opposed to proactive approach is sought. The inputs for this stage include the newly elicited ATs which reflect the new requirements, as well as the existing ATs which describe the already existing functionality in the core system. These ATs usually describe business logic aspects of the system and can easily be automated. In every iteration, new ATs and existing ATs are analyzed in order to determine any sources of variation in the business logic (i.e. variation points), the different variants, and any accompanying constraints. The key research question I address in this stage is:

RQ1. Can ATs be used to elicit variability due to business logic requirements in an iterative manner?

Having used ATs to analyze variability in business logic requirements, I investigate how variability can be iteratively analyzed in presentation and portability requirements in a lightweight manner that does not require heavy documentation or processes. This leads to the second research question in this stage which is:

RQ2. How can variability due to presentation and portability requirements be elicited in an iterative and lightweight manner?

The outcome of this stage is an updated variability profile. Variability profiles resemble a systematic way of describing the capabilities of the product line (i.e. variation points, variants and constraints). One of the main advantages of maintaining variability profiles is that they enable the automation of validity checks among features in the system and product instantiation. As new variability is incrementally elicited, these variability profiles will need to evolve in a consistent manner to support the systematic aspect of the framework. The variability profile can also be used to produce a feature model as discussed in the Background chapter.

4.2.3 Stage C: Variability Modleing

For all features in the system where variability exists, it is necessary that such variability be communicated to interested stakeholders in the organization. Traditionally, feature models (the outcome of the previous stage) are used for this purpose. However, traditional modeling approaches use intermediary requirement and design artefcats as traceability mechanisms to ensure consistency between the model and the implementation. In my research, I investigate how such feature models can be made executable so that traceability and consistency are improved. The question I address in this stage is:

RQ3. Can EATs be used to model variability in a system so that variability becomes communicable across the organization and traceable to the implementation?

4.2.4 Stage D: Variability Realization

The purpose of the realization stage is to implement the variation points and variants at the code level to match their manifestation in the executable feature model. In addition to the executable feature model, this stage takes as input a specific system (i.e. a system that satisfies context-specific requirements) and redesigns this system so that the context-specific requirements become as generic as possible to a range of contexts. The remaining layer that is specific to the different contexts will need a configurator to support the customization process. With the absence of requirement and design documents in agile contexts, a different approach is sought to systemize the realization process. Therefore, in this regard I address the following question:

RQ4. In an agile context, how can variation points and variants be realized at the code level in a reactive and systematic manner?

4.2.5 Stage E: Product Derivation

Deriving different products from a common code base to satisfy the different needs of customers is an essential aspect of SPL engineering. Automating this process is key to its efficiency – especially when mass customization is expected. Traditionally, this process is done during the application engineering phase. Application engineers bind the variation points to the appropriate variants guided by the available documents and design artefacts produced during domain engineering. In my framework, the derivation stage is responsible for producing different instances of the system given as input the generic system, the configurator, and the specific configurations needed by a given customer. In this process, the question I address is:

RQ5. How can the extended feature model support the derivation process of individual products from a common SPL base?

4.2.6 Issues and Challenges in Industrial Contexts

Having developed an agile product line framework in a research-oriented environment, I investigate the issues and challenges associated with adopting a SPL framework in an industrial context. For this reason, in my dissertation, I address the following question in great detail:

RQ 6. Independent of the specifics of the proposed framework, what are the technical and non-technical impediments that need to be taken into consideration before a new SPL framework becomes feasible in an industrial context?

4.3 Evaluation Strategy

In this section, I discuss the evaluation strategy of my research. I begin by explaining the alternate approaches to evaluate the research at hand and the challenges around such approaches. Then, I talk about the specific strategy I use to overcome these challenges.

4.3.1 Evaluation Challenges

The fundamental premise of my evaluation strategy is that SPL adoption is generally a long-term, organization-wide problem [Muffatto1999]. When the organization makes the decision to develop software families using a SPL approach, explicit support for reuse across team boundaries will be necessary [Jandourek1996]. Also, the decision to make the leap to a SPL strategy requires a long-term commitment to allow the product line to evolve in reaction to the market conditions and demands. These circumstances pose real challenges to conducting a holistic evaluation of the proposed framework. For one, an actual implementation of the framework requires that the researchers find an organization that practices ASD, is experiencing the problem the research is trying to address (i.e. similar products in a given domain), and is willing to make the necessary long-term commitment with all the associated risks. Finding such a company has been a big challenge especially given the unfortunate economic downturn in the past few years. One of our industrial partners did have a problem that was very relevant to my research, but they first needed to implement ASD practices in the company before they could consider adopting a SPL strategy. The time horizon for their transformation goes well beyond a timeline for a PhD thesis.

Another important issue in this regard is that an actual implementation of the framework would require the researchers to have a prolonged timeframe to observe the results and factor out the impact of external variables. For example, one opportunity I had considered to evaluate my work was a collaboration with a medium-scale Scandinavian company. The collaboration went relatively well (though very slow) in the first phase of collecting data to customize the proposed framework to the needs of the organization. Nonetheless, the second phase of implementing the proposed framework was interrupted multiple times by business distractions that made it impractical to pursue further collaboration.

The other alternative to evaluate the framework in its entirety was to conduct a small-scale controlled experiment in an academic environment. However, I firmly believe that such experiments are not appropriate when the main research goal is to address a problem that is large-scale by definition.

4.3.2 Evaluation Strategy

Given the challenges mentioned above, in my research I use a divide-and-conquer strategy in which I divide the bigger problem of adopting a SPL strategy in an agile organization into smaller and more manageable problems as described previously – namely: elicitation, modeling, realization, and derivation. For each of these problems, I conducted a separate research study where I use a number of different research and evaluation methods. These methods will be explained in detail in their respective chapters in the dissertation.

Having said that, I do realize that the sum of the parts is not equivalent to the whole, yet I believe it is a reasonable approximation. The accuracy of this approximation is improved by conducting an in-depth study within an industrial context to uncover the issues that need to be taken into consideration before transferring the framework to an industrial context.

4.4 Chapter Summary

In this chapter, I discussed my research goal, and I listed the research questions formulated to achieve this goal. I also discussed how a divide-and-conquer strategy is used to evaluate the different components of this research.

CHAPTER FIVE: VARIABILITY ELICITATION & EVOLUTION IN BUSINESS LOGIC REQUIREMENTS[*]

5.1 Preamble

The first step in variability management is to elicit variability from the available requirements. Traditionally, this is done proactively during the domain engineering phase and relies on documented requirements as the source of input. In ASD, one-shot requirement elicitation up-front is considered impractical, and documentation is minimal. Rather, requirement elicitation is strictly iterative in the sense that every release starts with activities that aim at eliciting requirements from the customer. These requirements are collected in the form of user stories that are then translated to acceptance tests (ATs) as detailed in chapter two. In a product line context, this means that the first releases may target a specific customer, but as soon as other customers demand a similar system, future releases will need to handle the issue of variability in the collected requirements. In this chapter, I argue that variability elicitation and analysis could be done reactively to be in harmony with the iterative nature of ASD. I show how this can be done using AT artefacts. Generally, it is difficult to capture non-functional requirements in an AT format [Melnik2004]. Therefore, in this chapter, I only focus on functional aspects representing business logic requirements. Chapter six will focus on requirements beyond business logic.

5.2 Research Instruments

The first research instrument I use in this chapter is an analysis of traditional variability management approaches. I choose one common approach to discuss in more detail. Then,

[*] This chapter is based on a published paper [Ghanam2011]. Co-author permission is attached to Appendix B.

I provide an analysis of the use of AT artefacts in ASD. Afterwards, I look in more depth at the issue of using AT artefacts to evolve variability in light of the conducted analysis and the following high-level research question:

RQ1. Can ATs be used to elicit variability due to business logic requirements in an iterative manner?

In the context of the framework I propose, this research question tackles the first of two parts of Stage B: Variability elicitation in business logic requirements (as shown in Figure 8).

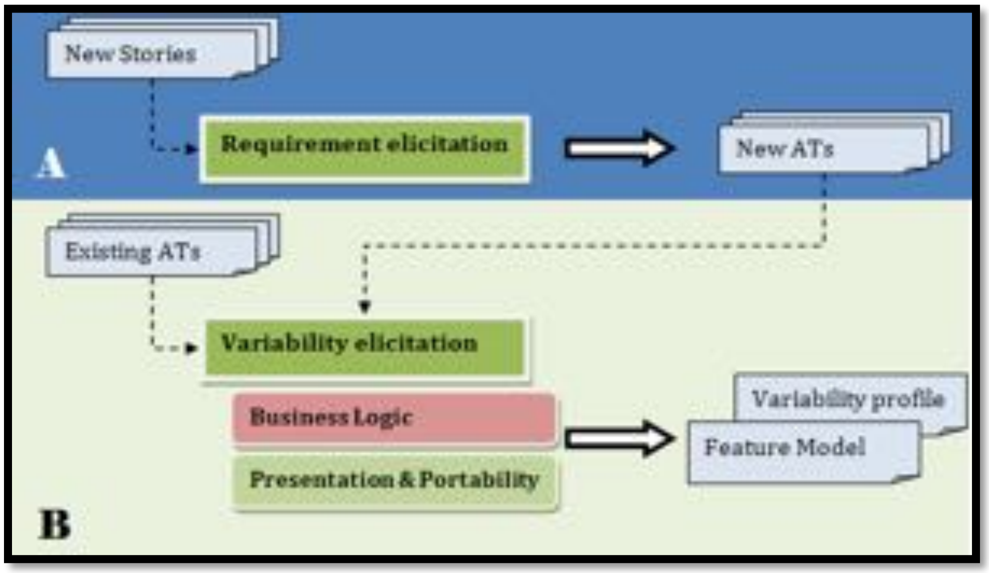


Figure 8 – This chapter tackles Stage B: Variability elicitation – Business Logic

In this stage, I assume that Stage A has already been completed for the current iteration using conventional ASD methods such as iteration planning meetings. The outcome of Stage A consists of a set of ATs that represent the new requirements demanded by the customer. This set along with the existing sets of ATs that have already been implemented in the system serve as input for Stage B where the variability analysis is conducted to elicit any variability sources and determine the needed variants and the required constraints. Consequently, this yields a variability profile that provides an updated list of the variation points in the system, the variants and their constraints (see Figure 8). This variability profile can be modeled using a feature tree as discussed in chapter two. As more increments take place and new ATs are written, this process is repeated to support the evolution of the variability in the system. One of the main advantages of maintaining variability profiles is that they enable the automation of validity checks and product instantiation as will be discussed in chapter seven.

RQ1 is broken down into four more specific research questions, namely:

Q1. Can developers build feature models iteratively and incrementally using ATs as proposed in the approach (Section 5.4)?

Q2. Can developers learn the six-point approach (Section 5.4.2) quickly and easily?

Q3. Does using ATs as building units for the feature model yield consistent variability interpretations across different developers?

Q4. Are ATs sufficient for developers to deduce explicit and implicit constraints?

These research questions are investigated through an exploratory study where I train participants to work with the proposed method, observe how participants perform in certain tasks, and then follow up with participants to get their feedback in retrospect. Given the lack of literature on the topic being investigated, this exploratory approach provided a basis for understanding the problem at hand in more depth and learning about the issues that need to be tackled in the following research efforts.

5.3 Analysis

Normally, SPL engineering starts off with the domain engineering phase. During this phase, engineers proactively plan for products as a family rather than as individual instances. Domain engineers conduct commonality and variability analysis to produce a variability profile for the potential system. This analysis is conducted through a variety of techniques. In this section, I analyze a common technique by Pohl et al. [Pohl2005] which entails four major steps:

1. Define common requirements: use application requirement matrices, priority analysis or checklist based analysis to review the requirements of systems the organization has previously built or expects to build in the future. Extract repeated requirements, requirements likely to become common in the future, or strategically common requirements.
2. Define requirement variability: look at how requirements across different systems might vary and understand why they vary. The objective of this step is to extract variation points, possible variants, as well as any dependencies or constraints.
3. Document findings from (1) and (2): this produces domain requirement documents that explain to application engineers how to instantiate applications.
4. Proceed to the next phases: use the documentation produced in (3) to design, implement, and test the architecture and its constituents.

In my dissertation, I do not argue that such a proactive approach is not viable. But I argue that this approach makes certain assumptions that are directly in conflict with core principles and practices of ASD which makes the adoption of this approach difficult in agile organizations. These assumptions include:

A. **Domain knowledge:** The organization has built a number of systems in the same domain. Or sufficient knowledge is available – at the time of the variability analysis – about the present and the future of the domain (i.e. knowing what will be common and what will be variable). Pohl et al. [Pohl2005] assert that building an SPL "requires sophisticated domain experience."

   Conflict with ASD: This implies that adopting a product line approach might be infeasible for smaller organizations entering a new market. Moreover, ASD considers acting upon predicted future requirements too risky, and thus may not be willing to substantially invest in requirement elicitation upfront.

B. **Requirement engineering:** A requirement engineering phase has been dedicated for each system including traditional practices such as requirement documentation.

   Conflict with ASD: In ASD, development starts early. As for requirements, ASD does not dedicate a requirement engineering phase, but rather preaches a minimalistic way of obtaining customers' needs using story cards and direct collaboration between all stakeholders of the project.

C. **Accurate documentation:** Requirement documents resulting from the requirement engineering phase are available and up-to-date. They accurately map to and are consistent with design, code and test artefacts.

   Conflict with ASD: In ASD, unless requested by the customer, requirement, design and test documents are considered of less value than actual implementation. In case documentation exists, it is generally difficult to ensure documents are up-to-date and consistent. Most ASD teams will not create requirement and design documents to the extent expected in Pohl's approach.

By looking at these assumptions and conflicts, I can summarize the main issues that need to be addressed in an agile variability management framework in the following two points:

1. The proactive elicitation of variability in requirements: For an agile approach, an iterative process is to be sought for the SPL to be reactive rather than proactive.
2. The reliance on documentation in the elicitation and analysis process: This indicates that even if Pohl's approach was used in an iterative manner to solve the issue of proactive treatment, the problem of significant overhead will still need to be addressed in an agile framework.

5.4 The Proposed Elicitation Approach

The previous section showed how variability analysis is conducted in some traditional SPL practices, and how a number of the basic suppositions underlying these practices are not suitable for an ASD culture. In this section, I present an elicitation approach that addresses the main two issues of proactive variability elicitation and documentation. Proactive elicitation is addressed by enabling the evolution of the variability profile through an iterative treatment that is lightweight enough to be repeated as many times as need be. The approach also addresses the issue of reliance on documentation by explicitly recognizing the notion that in ASD, documentation is not produced to describe the system under development. ASD, however, produces test artefacts to describe the system and act as anchor points for traceability relations. In the proposed approach, ATs are utilized in the variability elicitation process as will be detailed in the following section.

5.4.1 Acceptance Tests

In story TDD, specifications are written before writing code in the form of ATs. These ATs are usually written collaboratively by the stakeholders to ensure a consistent understanding of the system. ATs can be automated by tools like FIT [FIT2010], and thereafter they are called executable ATs (EATs). Automation makes it possible to continuously run these tests against the code developers write to measure how complete a feature implementation is. I propose the use of ATs to elicit variability in requirements. The benefit of using ATs is twofold. For one, no burden is added on the ASD team to produce extra artefacts given that ATs are a natural starting point in agile iterations. Secondly, since ASD promotes a refactor-whenever-needed notion, these tests are continuously updated to reflect changes in the system. Hence, it can be assumed that these artefacts represent a sufficiently up-to-date account of the system they test.

In order to use test artefacts as a basis for the proposed approach, it is important to understand in what form these artefacts exist in the test repository. In this analysis, I show how artefacts typically exist in a common tool for writing and running ATs called FitNesse [FitNesse2011]. FitNesse is an AT framework based on a fully integrated standalone wiki. With the help of the user guide provided with the FitNesse tool package, I produced an object model that reflects how test artefacts relate to the system under test (SUT) and to each other. As Figure 9 shows, the production of test artefacts is driven by features requested by the customer. In this context, I use the term feature to refer to a chunk of functionality that delivers business value [VersionOne2011]. There is no restriction on how small or large this functionality is, as long as the customer thinks its existence would add value to the delivered system. Internally, nonetheless, developers may choose to break the feature down into sub-features to make it more manageable and testable. While one or more test artefacts are produced to test a single feature, it is also true that a single test artefact might cut across a number of features in the system.

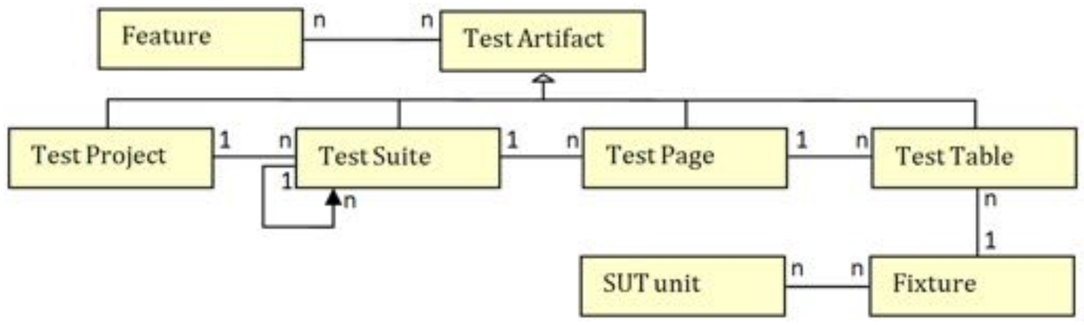


Figure 9 – An object model for test artefacts in a SUT

A test artefact can exist at different granularities. Typically, developers would start by creating a test project for the SUT. The test project has a number of test suites that are optionally used to organize tests into a recursive folder-like structure. Grouping tests into suites might be based on a feature breakdown or might be chronological based on iterations. Each suite consists of one or more test pages. In FitNesse, these pages are files, each of which has a number of tables representing user stories. Test tables can take different formats based on the type of fixture they are linked to (e.g. column or row). In essence, these tables are the specifications of the customer. In order for test tables to be executed, they are linked to a thin layer of testing code called a fixture. It is within these fixtures where the actual production code is tested. A fixture uses a number of code units to execute specifications from the AT tables.

The significance of understanding this object model lies in the fact that capturing commonality and variability in features can occur at different granularities of test artefacts. Some test artefacts can be seen as common across different applications in the family, and thus are considered *default* artefacts. Some other artefacts may be described as *optional* or *alternatives*. For example, a customer may want to exclude a certain scenario or include an additional one in a given feature. In this case, variability is defined within the test page to include, exclude or add certain test tables. Some of these tables may be in conflict; therefore, multiplicity and dependency constraints need to govern the selection process. The following section explains through an example how this can be achieved.

5.4.2 Introducing Variability

The use of ATs to elicit variability and evolve variability profiles occurs in six steps as follows:

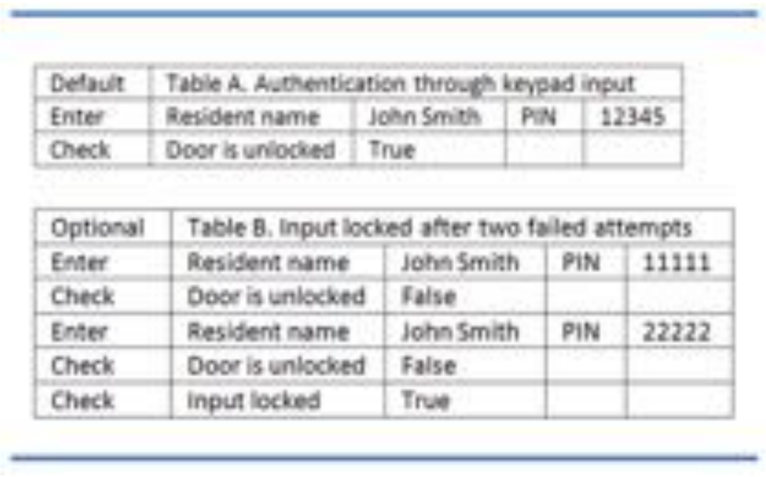
1) The very first system is built in a normal ASD process to satisfy the requirements of the customer at hand without investing into future speculations of what may vary.
2) An initial feature model of the system is produced using ATs as the building units. The initial feature model is a simple decomposition of a given feature into the different scenarios to be supported. It does not necessarily contain any constraints.
3) Upon the demand of a similar system by a new customer, the existing feature model with the associated ATs are made available to the new customer.

4) The new customer picks those ATs that meet their specific needs. The chosen ATs represent a feature instance.
5) If the currently available ATs do not satisfy the customer's needs, the customer defines a change set that can include adding, removing or replacing ATs.
6) Based on the change set produced in 5, the feature model is updated.

Step 2 in the abovementioned process can preferably be delayed to be done alongside with steps 3 to 5 (i.e. only when there is a demand for a second system that varies from the original system). If this is the case, feature modeling can be done incrementally by considering only those features that actually vary with the new requirements.

To illustrate the idea, I will use the example of smart home systems. Smart home systems make it possible to monitor and control the surrounding environment. These systems usually need to encompass a large variety of home infrastructures, devices, security mechanisms and customer preferences. More information on smart home systems is available in chapter two.

In an intelligent home system, test tables in a page describing an access control feature looks like the one in Figure 10.



Default	Table A. Authentication through keypad input			
Enter	Resident name	John Smith	PIN	12345
Check	Door is unlocked	True		

Optional	Table B. Input locked after two failed attempts			
Enter	Resident name	John Smith	PIN	11111
Check	Door is unlocked	False		
Enter	Resident name	John Smith	PIN	22222
Check	Door is unlocked	False		
Check	Input locked	True		

Figure 10 - A test page is composed of a number of test tables

This test page looks almost the same as a traditional FitNesse test page. The only difference is that I denoted some tests as “default” and others as “optional.” Default artefacts are those that are essential to reflect the value of the feature at hand. If removed, the feature becomes meaningless or valueless. The default attribute should not constrain the flexibility of responding to new requirements. It is only an indication, for new customers, that this element was of special importance to previous customers, making it a good candidate to become common across different instances.

Optional test artefacts, on the other hand, are those that can be looked at as add-ons rather than necessities. This might be perceived differently by different customers. Therefore, optionality is only a guide for future customers that an element might be cut out without omitting the value of the feature. This initial assumption might be challenged later on by other customers who deem the optional element to be an indispensible part of the feature. Thus, an optional test artefact could be upgraded to become a default one and vice versa. The initial state of the feature can be modeled in a feature tree as shown in Figure 11. A solid line symbolizes a default artefact whereas a dotted line symbolizes an optional one.

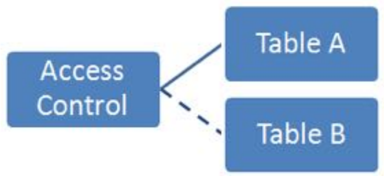
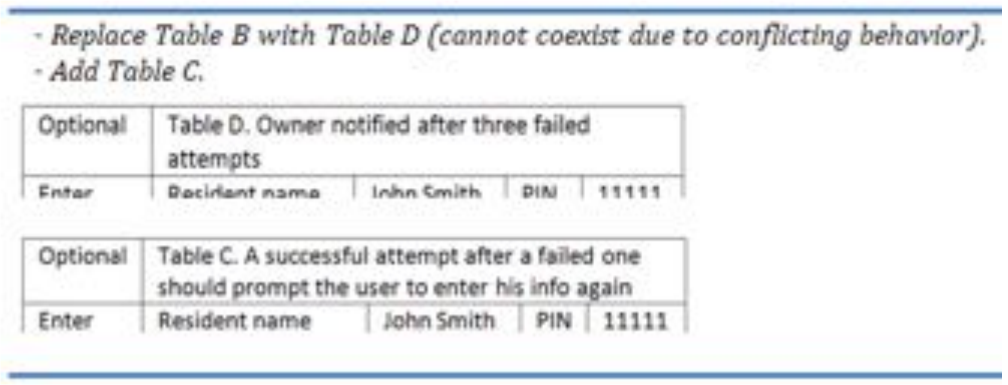


Figure 11 - The initial feature model

Now, say a new customer requests a change to the access control feature via PIN. The customer is given the test page in Figure 10. They have the option to exclude existing tables or add new ones. Say the customer requests the customization shown in Figure 12. Table C is added to the test page as one more option future customers can pick from. However, the addition of Table D is not as straightforward due to its conflict with Table B. That is, according to Table B, the input should be locked for 2 minutes after 2 failed attempts. Whereas according to Table D, the user is allowed 3 attempts after which the owner is notified.



- Replace Table B with Table D (cannot coexist due to conflicting behavior).
- Add Table C.

Optional	Table D. Owner notified after three failed attempts			
Enter	Resident name	John Smith	PIN	11111

Optional	Table C. A successful attempt after a failed one should prompt the user to enter his info again			
Enter	Resident name	John Smith	PIN	11111

Figure 12 - Customization requested by the customer

To solve this issue, one can impose a constraint that Table B and Table D cannot coexist. The new version of the test page can be visualized using a feature model as shown in Figure 13. Multiplicity constraints in the form of [min..max] may be added to govern the selection of artefacts. In this case, a [0..1] indicates that only one element may be selected amongst the set {Table B, Table D}.

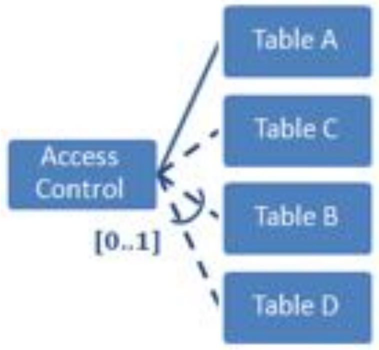


Figure 13 – The evolved feature model

5.4.3 Customer involvement

One of the major aspects contributing to the success of ASD is its focus on customer involvement and satisfaction. By using artefacts that proved to work well to communicate requirements amongst stakeholders, our approach makes sure this principle is not compromised. The approach is an additional interaction technique through which customers can be aware of how a system (similar to the one they are requesting) would typically look like. This is achieved by exposing the customer to previously built systems represented through test artefacts. A traditional problem in requirement engineering is that the customer may not initially be able to weigh the value of different aspects of the system in a consistent manner with their actual needs. Through my approach, the customer can ask questions like: “Why is this aspect of the feature of value to me while it was not as valuable to others?” At the same time, the customer will enjoy the flexibility to build upon existing systems, modify certain aspects of these systems to fit his needs better, remove aspects that he may not be willing to pay for, and select from different alternatives based on his own evaluation of what is deemed more important (e.g. performance versus cost).

5.5 Exploratory Study

5.5.1 Goal & Questions

The goal of the study presented in this section is to examine the foundations of the proposed elicitation approach with the help of independent participants. The study explores four key aspects of the approach, namely: evolution, learnability, consistency, and constraints. Each aspect is related to one of the following research questions:

Q1. Can developers build feature models iteratively and incrementally using ATs as proposed in the approach?

Q2. Can developers learn the six-point approach quickly and easily?

Q3. Does using ATs as building units for the feature model yield consistent variability interpretations across different developers?

Q4. Are ATs sufficient for developers to deduce explicit and implicit constraints?

To examine the approach in light of these questions, 16 graduate students were invited to participate in an observational session. All participants were enrolled in computer science or electrical & computer engineering programs and they had some background in software engineering. As will be detailed later, participants were asked to study a written tutorial on the approach, solve three exercises, and then fill out a follow-up questionnaire. Table 1 lists the aspects reflected by the four questions, and the required observations to answer these questions.

Table 1 - Aspects and the required observations

Q	*Aspect*	*Required Observation*

Q	***Aspect***	***Required Observation***
1	Evolution	Observe if the participants will be able to start at an initial state of the feature and incorporate new requirements as they come in the form of ATs. The final state of the feature should be consistent with the intended one.
2	Learnability	After going through a written tutorial, observe if the participants will be able to: a. Distinguish between a feature instance and a generic feature model. b. Use ATs as building units for feature models. c. Relate instantiation requests to the required ATs.
3	Consistency	Observe if the participants will be able to build hierarchical models that are consistent with the ones I built and deemed to be the intended interpretation (hence, consistent across different participants).
4	Constraints	Observe if the participants will be able to deduce all explicit and implicit constraints from the provided ATs.

5.5.2 Data gathering

Tutorial: Participants were asked to go through a written tutorial on our approach of eliciting variability in requirements through ATs. When they finished the tutorial, participants had to solve a trial exercise to ensure they gained the understanding required to complete the study.

Exercises: After the tutorial, participants were handed three exercises one at a time. Exercise 1 was to measure the learnability of the approach. In this exercise, participants were given a 3-stage scenario. The first stage included the initial customer specifications of a feature in AT format (similar to the one in Figure 10). The two following stages included requests by other customers for the same feature as in the previous stage, but each having their own customizations represented through ATs (similar to the one in Figure 12). In each stage, participants were asked to: draw a feature model representing the state of the feature as requested by the current customer (i.e. as an application instance), and then draw a feature tree of the evolved variability model (i.e. the generic feature model that is used to derive instances).

Exercise 2 was similar to exercise 1 but involved more complex scenarios. It was used to observe the deduction and applicability of constraints through ATs. These constraints were either explicitly mentioned such as: "A and B cannot coexist," or I implicitly planted them within the contents of the ATs to observe if the participants would be able to detect them. Exercises 1 and 2 jointly measured the evolution aspect of the approach.

Finally, Exercise 3 asked the participants to use the feature model built in the two previous exercises to deduce feature instances of minimum cost (i.e. the smallest possible feature instance) and maximum value (i.e. the largest possible feature instance). This last exercise was used to observe the readability of the produced feature model and the ease of determining which ATs are needed to build a certain instance. The output of each of the three exercises consisted of: a) a feature instance tree representing the feature as an instance for a specific customer; and b) an updated generic feature variability model.

Consistency was defined as the degree of similarity (i.e. the percentage of matching edges, nodes and constraints) between the feature models produced by the participant and the ones I built and deemed to be the intended interpretation of the provided AT-based scenarios.

Questionnaire: At the end of the study, participants were given a questionnaire to retrospectively capture their impressions and opinions about the exercises.

5.5.3 Results & Discussion

5.5.3.1 Tutorial and Exercises

The tutorial and exercises together lasted 41.5 minutes per participant on average ranging from 22 minutes to 67 minutes. For each participant, I measured the time they spent on each exercise, and the consistency of their outcome compared to the outcome I had anticipated. Table 2 shows a summary of the results. The full results are available in Appendix C. Participants were asked to take as much time as needed to go through the tutorial which averaged at 13.5 minutes. Exercise 2 took almost double the time of Exercise 1. This is normal considering that Exercise 2 was a more complex one. But what is noteworthy is that the consistency did not change much. I attribute this to the learning effect that is likely to have occurred during Exercise 1. Except for one participant who could not understand Exercise 3, all participants could solve Exercise 3 with 100% consistency.

Table 2 - Summary of the results

Averages	*Tutorial*	*Exercise 1*	*Exercise 2*	*Exercise 3*
Time (minutes)	13.5 ± 4.3	8.1 ± 2.9	15.8 ± 6.2	4.2 ± 1.9

Consistency	NA	88% ± 11%	89% ± 10%	100%

Evolution: Figure 14 provides a closer look at the performance of all 16 participants in exercises 1 and 2. Each dot represents one participant. The y-axis represents the percentage of consistency (all data points are above 60%). In both cases, I observed that participants were in fact able to start at an initial state of the feature and elicit variability from new requirements as they came. The final state of the feature was consistent with the intended one in more than 80% of the cases.

Learnability: Since Exercise 1 was the first exercise to follow the tutorial, I chose to analyze it in more detail. I found that all participants could achieve the three objectives mentioned under the learnability aspect.

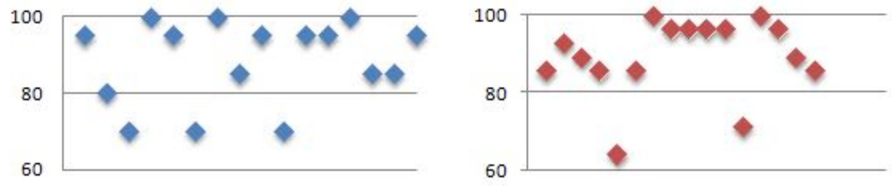
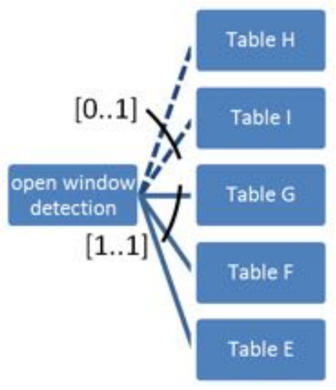
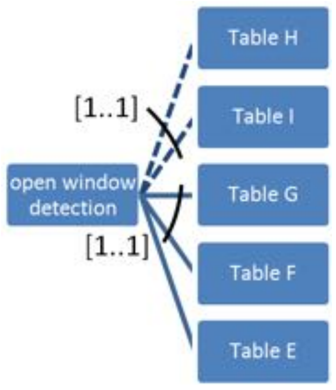


Figure 14 – Consistency (%) in Exercise 1 to the left, and in Exercise 2 to the right

Consistency: To check for the consistency aspect, I combined the results of all three exercises and I found that the interpretations of the contents of the ATs were *mostly* consistent amongst more than 80% of the participants. I did, however, find some discrepancies in the way participants chose to model certain parts. One example was the modeling of two mutually exclusive optional features. I have anticipated that the modeling would incorporate a [0..1] constraint (as shown in Figure 15a) indicating that not selecting any of the features is permissible. Nevertheless, 10 out of the 16 participants chose to model it in a different way where they used a [1..1] constraint (as shown in Figure 15b) rationalizing that the optionality is already accounted for by the dotted line.



(a)

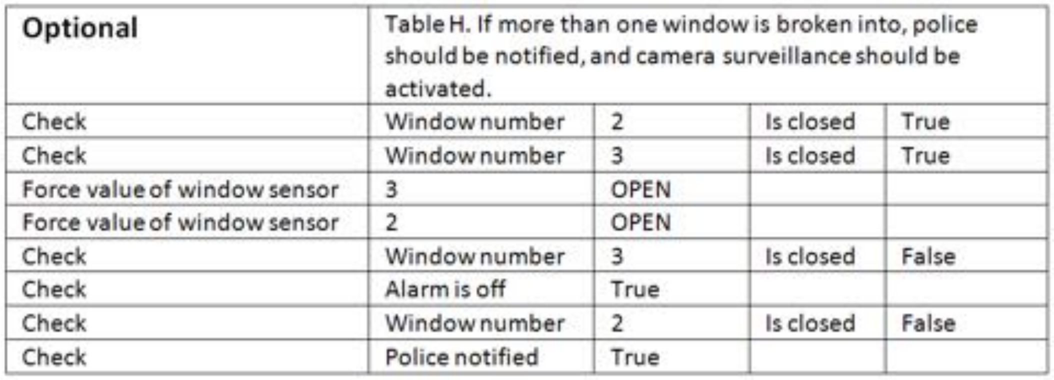


(b)

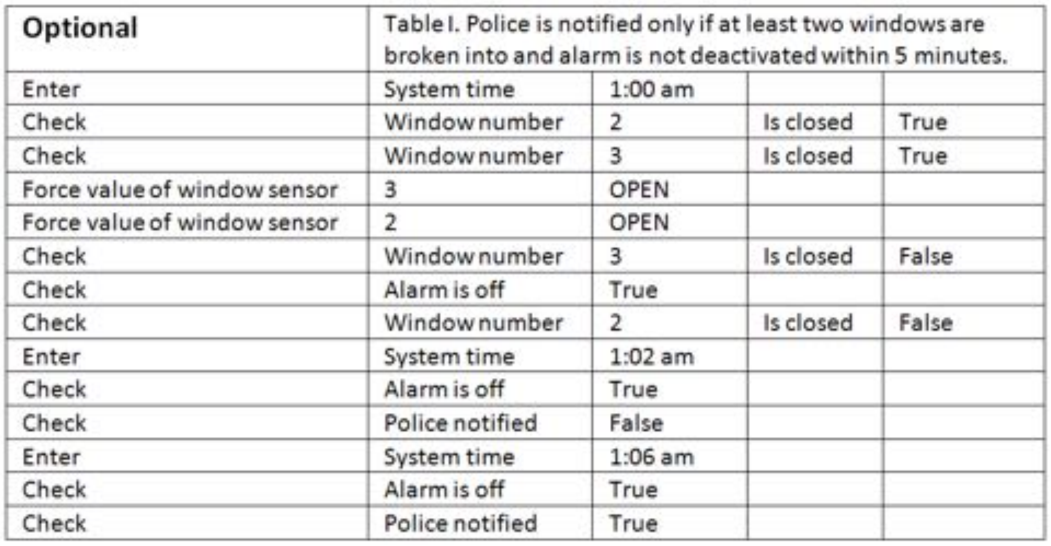
Figure 15 – (a) The expected model as opposed to (b) the produced model with the [1..1] constraint between Table H and Table I

Such a pattern is interesting yet irrelevant to the questions I was interested in examining in this study.

Constraints: When looking at realizing and modeling constraints, I found little evidence that ATs were sufficient to deduce implicit constraints. All but one participant could realize and model explicitly mentioned constrains such as “Remove Table F as it cannot coexist with Table G”. However, half of the participants could not deduce an implicit constraint I had planted in Exercise 2. In this case, the two relevant tables were as shown in Figure 16.



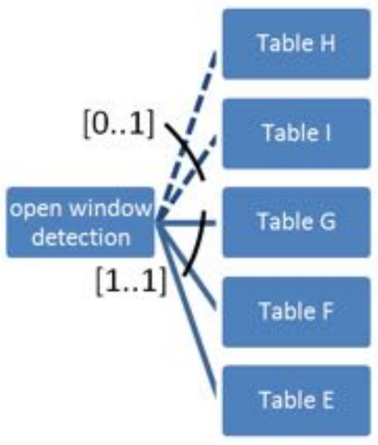
Optional	Table H. If more than one window is broken into, police should be notified, and camera surveillance should be activated.			
Check	Window number	2	Is closed	True
Check	Window number	3	Is closed	True
Force value of window sensor	3	OPEN		
Force value of window sensor	2	OPEN		
Check	Window number	3	Is closed	False
Check	Alarm is off	True		
Check	Window number	2	Is closed	False
Check	Police notified	True		



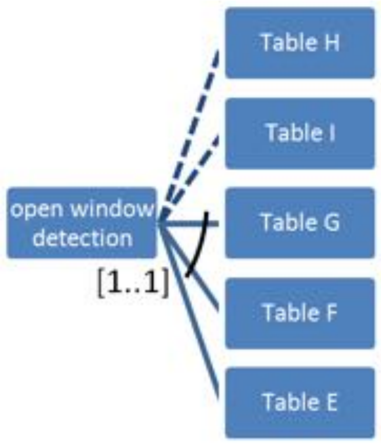
Optional	Table I. Police is notified only if at least two windows are broken into and alarm is not deactivated within 5 minutes.			
Enter	System time	1:00 am		
Check	Window number	2	Is closed	True
Check	Window number	3	Is closed	True
Force value of window sensor	3	OPEN		
Force value of window sensor	2	OPEN		
Check	Window number	3	Is closed	False
Check	Alarm is off	True		
Check	Window number	2	Is closed	False
Enter	System time	1:02 am		
Check	Alarm is off	True		
Check	Police notified	False		
Enter	System time	1:06 am		
Check	Alarm is off	True		
Check	Police notified	True		

Figure 16 – The two tables with the implicit constraint

Since the two tables have different acceptance criteria for what happens when more than one window is broken into, the two tables would be in conflict if they were both selected in a single instance. Therefore, the expected model was as shown in Figure 17a, whereas half of the participants modeled this as shown in Figure 17b.



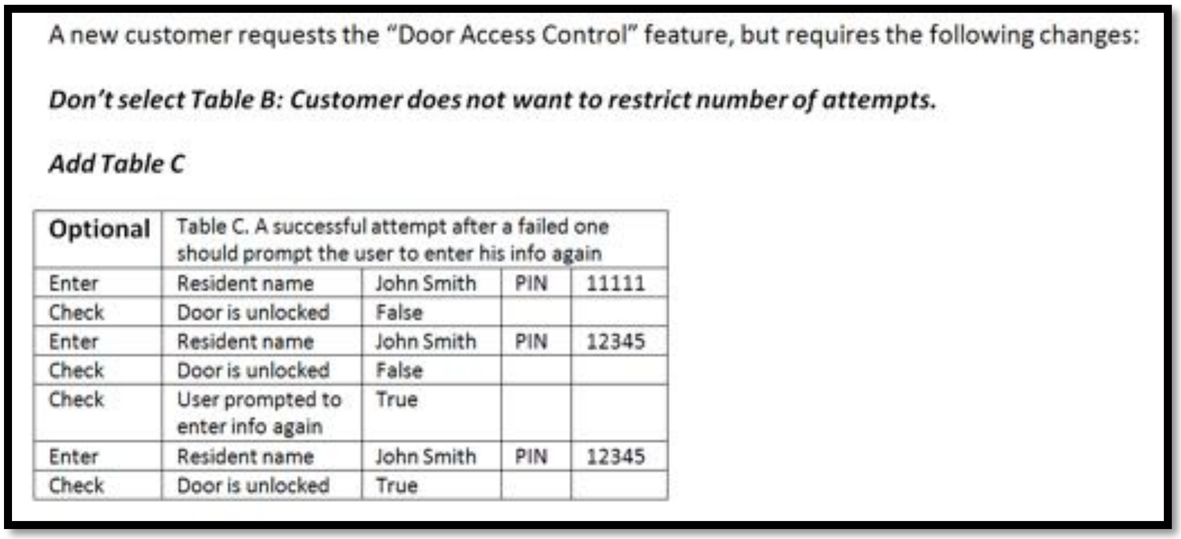
(a)



(b)

Figure 17 – (a) The expected model as opposed to (b) the produced model missing the constraint between Table H and Table I

There were also some instances where constrains were unnecessarily imposed. For example, in the scenario shown in Figure 18, the participant chose to impose a [1..1] constraint between Table B and Table C for the sheer fact that the customer request included one and excluded the other.


A new customer requests the "Door Access Control" feature, but requires the following changes:

Don't select Table B: Customer does not want to restrict number of attempts.

Add Table C

Optional	Table C. A successful attempt after a failed one should prompt the user to enter his info again			
Enter	Resident name	John Smith	PIN	11111
Check	Door is unlocked	False		
Enter	Resident name	John Smith	PIN	12345
Check	Door is unlocked	False		
Check	User prompted to enter info again	True		
Enter	Resident name	John Smith	PIN	12345
Check	Door is unlocked	True		

Figure 18 – A scenario for unnecessarily imposed constraints

5.5.3.2 Questionnaire

The questionnaire had seven Likert-scale items and a space for participants to jot down their comments. Figure 19 shows the responses to the questionnaire. Every item in the questionnaire contained a statement for which the participants chose a decision – anywhere between strongly agree or strongly disagree.

The first 2 questions were control questions to make sure participants did not have a problem with the concept of the feature model itself. According to the questionnaire, participants found the approach flexible and easy to grasp and apply. Dealing with ATs seemed to be a bit of a hassle, but only for participants who had not worked with ATs before. One participant, who did work with ATs for a while, noted in the written comment that he "found dealing with ATs very easy." Other comments mainly reflected the trickiness of dealing with constraints. As one participant put it, "I think dealing with constraints is tricky and might cause confusion." Some participants also commented on the scalability of the approach for larger and more complex systems: "it is easy to handle problems with not many conflicts, but I am not sure whether it would be still as easy when many constraints exist."

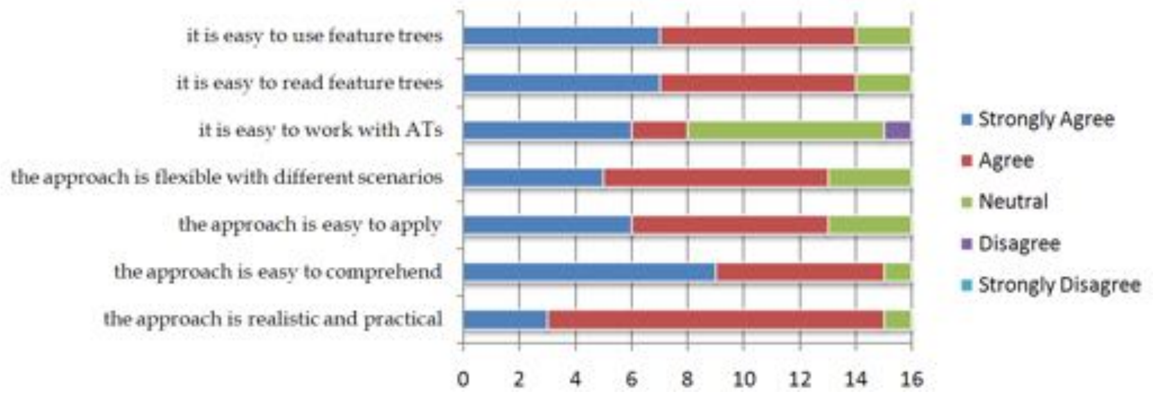


Figure 19 – Reponses to the questionnaire

5.5.4 Threats to Validity

This study was not intended to be a controlled experiment (since I did not have a control group). Nevertheless, I did take a number of measures to mitigate foreseeable biases and provide controls over the variables that could be at play in the study. By providing a tutorial, I tried to ensure all participants started from a common ground. Exercises were designed carefully and were handed to participants in a specific order. Precise wording of questions and non-biasing responses to participants' concerns during the study were also important measures.

I still, however, faced some validity threats. The key-answer that was used as a benchmark to measure the participants' performance against had been developed by ourselves – which might have introduced a bias in the results. The use of ATs as the only instrument to represent specifications posed a threat to the internal validity of the study. That is, had I used another type of requirement specifications (e.g. flat documents) with another group of participants, I might have been more confident in drawing conclusions about the effect of using ATs.

Moreover, in the context of this study, the hypothetical scenarios might not accurately reflect the complexity found in real life software projects. This was intended so that the participants could focus on the process itself as opposed to the complexity of a specific domain. Nevertheless, this might be a threat to the external validity of the study affecting the generalization of the findings.

5.6 Chapter Summary

In this chapter, I argued that variability elicitation and analysis should be done reactively to support the evolutionary nature of ASD, and I proposed the use of AT artefacts to achieve that. The proposed variability elicitation approach differs from traditional approaches in two ways. First, variability elicitation in our approach happens iteratively and incrementally and only when there is enough justification to do so (i.e. actual customer requests); whereas traditionally variability has to be speculated and accounted for in advance during the domain engineering phase. Second, the proposed approach leverages existing test artefacts to elicit variability as opposed to introducing extra overhead such as requirement documents. The exploratory study examined four aspects of the proposed approach and provided interesting insights into its strengths and weaknesses. Strengths included providing support for the evolutionary nature of agile projects, easy learnability, and consistency. The main weakness the study revealed was related to the difficulty of deducing implicit constraints from ATs. In the next chapter, I discuss how this issue could be resolved through the use of a lightweight analysis technique.

CHAPTER SIX: VARIABILITY ELICITATION & EVOLUTION IN PRESENTATION AND PORTABILITY REQUIREMENTS[*]

6.1 Preamble

This chapter is a continuation of the previous chapter on variability elicitation with a focus on aspects other than business logic, specifically presentation and portability issues. As discussed in the previous chapter, the first step in variability management is to elicit variability from the available requirements. Elicitation is traditionally done in a proactive manner and relies on documented requirements as the source of input. For ASD, both issues of proactive treatment and heavyweight documentation are deemed impractical. In

[*] This chapter is based on published papers [Ghanam2010] and [Andreychuk2010]. Co-author permission is attached to Appendix B.

the previous chapter, I showed how ATs could be used in an evolutionary manner to elicit variability due to business logic requirements and update the feature model as needed. In this chapter, I show how presentation and portability requirements can be dealt with in an evolutionary and lightweight manner. I specifically focus on these two aspects because they were the main sources of variability in our application domain (beside variability in business logic).

6.2 Research Instruments

In this chapter, I investigate how variability due to presentation and portability requirements can be elicited and analyzed in an iterative and lightweight manner. I discuss this issue in light of the following research question:

RQ2. How can variability due to presentation and portability requirements be elicited in an iterative and lightweight manner?

In the context of the framework I propose, this research question tackles the second part of Stage B: Variability elicitation in presentation and portability requirements (as shown in Figure 20).

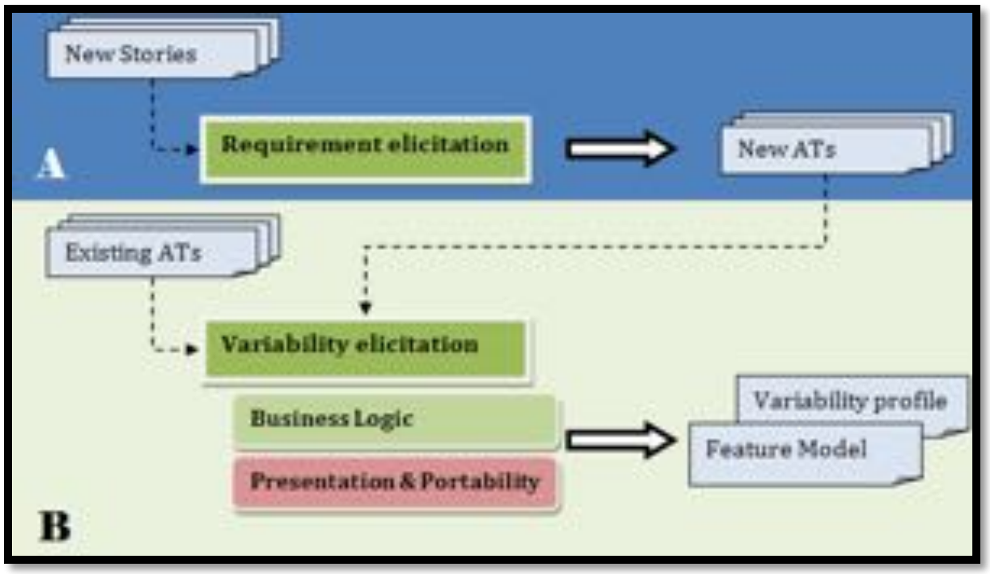


Figure 20 – This chapter tackles Stage B: Variability elicitation – Presentation & Portability

For this part, I hold similar assumptions to the ones in the business logic analysis part. Namely, I assume that Stage A has already been completed for the current increment using conventional ASD methods such as release planning meetings. Nonetheless, the outcome that will be used in this part of the analysis does not necessarily have to be automated ATs – because it is often challenging to automate ATs representing non-functional aspects [Melnik2004]. Rather, ATs could exist in a simple user story format that represents the new requirements demanded by the customer. The new and existing sets of requirements are used as input for Stage B where the variability analysis is conducted to elicit variability. As a result, this analysis yields a variability profile that provides an updated list of the variation points in the system, the variants and their constraints. Unlike business logic requirements, the variability profile for presentation and portability may not be translatable to a feature model because these two aspects usually cut across a number of features. As more increments take place and new requirements become available, this process is repeated to support the evolution of variability in the system.

In this chapter, I follow an action research (AR) approach which is a well established iterative technique that is usually used to solve problems in practice following the notion of "learning by doing" [O'Brien1998]. In AR, a problem is diagnosed first. Then, a proposal (aka. action plan) is prepared to approach the problem. The proposal is evaluated by applying it to the original problem. And then, the findings are identified and incorporated in the following iterations [Susman1983]. Considering the highly practical nature of my research, I use AR as a means to connect research and practice so that the two are aligned well and feed input to each other.

6.3 The Problem: Variability due to Presentation and Portability

6.3.1 Presentation

Variability in presentation as a non-functional aspect can occur due to a number of factors:

- Different users: When the same system is expected to be used by individuals with different technical skills, the interface will need to support both basic and advanced levels.
- Different operation modes: When the system can be operated through more than one mode such as user mode and designer mode, the interface will need to support both modes of operation.

- Different hosting devices: When the system is expected to run on different devices with different display features and capabilities, the interaction techniques may vary according to the operating device.

The effect of each one of the abovementioned factors is exacerbated in a product line context due to economic opportunities availed by mass customization. However, dealing with this kind of variability using a traditional proactive approach is associated with many risks, especially in a domain that is still emerging or when the supportive technologies are rapidly changing. For example, up-front investments into detailed design based on speculations on who will use a given software system may be lost if the speculations turned out to be inaccurate. Facebook, for instance, initially targeted an audience of college students – a user base which is generally more tech-savvy than older generations. However, user base statistics in 2007 have shown that the fastest growing demographic of Facebook users is the 25 and up age group. Such fast changing markets require nimbleness to react quickly without having to worry about huge losses in up-front investments [MacManus2007]. Similarly, investing into software design based on speculations on what interaction techniques will be popular in the future can pose significant risks. As will be shown in the case study later on, display technologies have drastically evolved in a relatively short period of time with many vendors such as Microsoft [Microsoft2011], SMART [SMART2011], and Apple [Apple2011] coming up with various touch and projection technologies as well as a wide range of interaction techniques. In this domain, the agility to respond to the emergence of new technologies in the market is a valuable asset.

6.3.2 Portability

As defined by Boehm et al. [Boehm1976], portability is the quality of a software system wherein the code can be easily run on computer hardware configurations other than its current one. The system can have more or less of the portability quality depending on how many hardware environments it can be run on without any issues. For example, a game written in Java can run on a Windows machine, a Mac machine, and any mobile device with a Java virtual machine. In a software product line context, portability is important but requires special considerations in the design process that increase the cost. This increase in cost, however, can be justified if the hardware platforms being targeted have known and stable specifications and the demand for them is evident. On the other hand, if the hardware platforms are still evolving or if the market is not stable, putting investment into portability becomes very risky. Take the example of horizontal displays. To design a portable application that works on different horizontal display technologies, I need to design for the aspects of these technologies that are likely to vary. Nonetheless, as new vendors keep entering the market with various projection technologies, image recognition techniques, and Application Programming Interfaces (APIs), speculation becomes more challenging and error-prone. Therefore, a reactive as opposed to proactive approach to this variability is more befitting in similar scenarios.

6.4 Handling Variability

This section presents the proposed approach to handle variability due to presentation and portability in a reactive and lightweight manner. The general framework of the approach is captured in Figure 21.

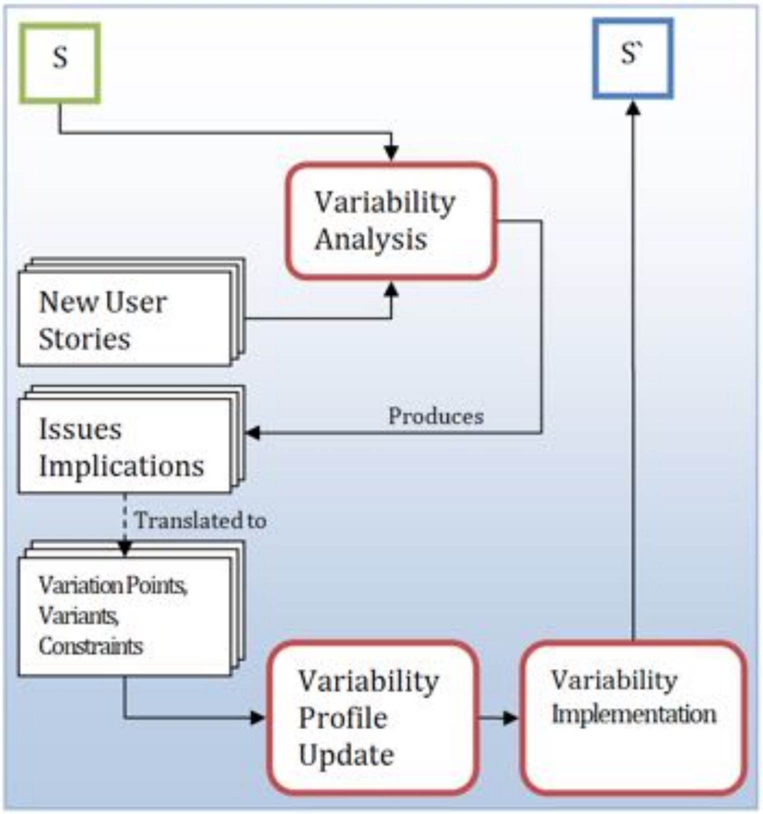

S
S`
Variability Analysis
New User Stories
Issues Implications
Produces
Translated to
Variation Points, Variants, Constraints
Variability Profile Update
Variability Implementation

Figure 21 - Reactive approach to variability due to presentation and portability

Because this is a reactive approach, it is assumed that a system already exists that satisfies the needs of a given customer base but does not yet satisfy a new set of requirements. This system is denoted as S. The goal of the approach is to make the transition between S to S` where S` is a system that satisfies all the requirements satisfied by S in addition to satisfying the new requirements. The transition process encompasses three steps (outlined with red): variability analysis, variability profile update, and variability implementation.

6.4.1 Variability Analysis

The goal of this step is to translate a set of user stories into a variability profile consisting of variation points and variants. As mentioned earlier, variability analysis is traditionally conducted up-front in the domain engineering phase. Elicited requirements are analyzed in terms of what they share in common, and in what aspects they may vary. Sources of variations are determined, along with the allowed values for these variations. In the proposed approach, I avoid one-shot up-front variability analysis, simply because it does not fit within the iterative nature of requirement elicitation in agile methods. Rather, a variability analysis is conducted every iteration between the current requirements in the system and the newly elicited requirements. Each user story is analyzed against the existing core assets and the current variability model in the system to determine the set of issues that need to be taken into consideration. Each of the issues is then studied to determine the implications of the issue at hand on the existing the system. The following definitions are used for the concepts mentioned above:

- User story: a desired addition or change to the existing system as described by the customer. The customer can be either external or internal to the organization.
- Issue: a reason explaining why the current system cannot – as is – satisfy the user story. A single user story usually results in a set of issues.
- Implication: an action that needs to be taken in order to resolve the issues resulting from a given user story.

6.4.2 Variability Profile Update

Having determined the issues and their implications in the previous step, each implication is analyzed further to deduce its impact on the variability profile. Namely, I am interested in uncovering:

- The sources of variability, which yields new variation points in the system or an update to the existing ones.

- The possible values for each source of variability, which yield the variants for each variation point.
- The conditions governing the selection process of variants, which yields the constraints that need to be imposed, if any.

Variability profile refers to the representation of a system in terms of what is constant (*aka*. base system) and what is variable. The variable part of the system contains a list of all variation points in the system and their variants. A number of representation and modeling techniques exist, but in this chapter it suffices to use a simple notation. A system (S) is composed of the union of two parts: a constant part (S_β) representing the set of requirements that do not change in different product instances, and a variable part (S_V) representing the set of requirements that may change in different product instances.

$$S = S_\beta \cup S_V$$

For each requirement r in the set S_V, there exists a nonempty set VP_r containing the variation points of that requirement:

$$\forall r \in S_V, \exists VP_r \text{where } VP_r \neq \emptyset$$

Because this is a reactive approach, one variant is deemed insufficient for an aspect to be considered variable. Therefore, each variation point νρ is expected to possess a set of variants $V_{\nu\rho}$ that includes at least two variants (ν_1 and ν_2) governed by a set of constraints (C):

$$\forall \nu\rho \in VP_r, \exists V_{\nu\rho} \text{ where } V_{\nu\rho} = \{\nu_1, \nu_2, \dots : C\}$$

The selection of a given variant yields a nonempty set of implications *I*:

$$\nu_i \xrightarrow{yields} I \text{ where } I \neq \emptyset$$

Example:

The Weather Watch module can be described as:

$$\mathrm{S}_{\text{Weather Watch}} = \{\text{Weather Model}\} \cup \{\text{Weather Trend Analyzer, UI Panel}\}$$

Accordingly, there exist $VP_{\text{Weather Trend Analyzer}}$ and $VP_{\text{UI Panel}}$.

Take for example:

$$VP_{\text{UI Panel}} = \{\mathrm{v}\rho_1 = \ \mathit{Panel\ type}\}$$

$$V_{Panel\ type} = \ \{\mathrm{v}_1 \ = \text{handheld}, \mathrm{v}_2 = \text{PC} : C = \{\text{v1 and v2 are mutually exclusive}\}\}$$

$$\mathrm{v}_1 \xrightarrow{yields} \{downscaled\ UI, touch\ gesture\}$$

$$\mathrm{v}_2 \xrightarrow{yields} \{normal\ scale\ UI, mouse\ click\}$$

Note: In this example, only one variation point exists for each requirement. It is possible for a given requirement to have more than one variation point (as per the formulation above).

Initially, the system S is described as $S = S_\beta$ but as more requirements are elicited in each increment, variation points may be added to the profile. The variability profile is also updated with any new variants arising due to the new requirements. At any point of time, there should be only one system and a set of variation points that makes it possible to produce different product instances.

It is important to keep a variability profile for the system to make explicit any dependencies and constraints between variation points and variants. They also support traceability of all aspects of variability in the requirements to code artefacts (as will be seen in chapter eight). A variability profile also plays a role in communicating variability to all stakeholders throughout and after the development process, and they help in the product instantiation step as will be explained in chapter nine.

6.4.3 Variability Implementation

In this subsection, I go briefly over some aspects of refactoring, testing and realization to show the direct impact of applying the reactive approach to implement variability in the target system. All three aspects will be discussed in detail in dedicated chapters.

Refactoring: During this step, new architecture layers are introduced to abstract common aspects. Other layers may be specialized to handle variable aspects. The goal of the refactoring step is to merely refactor the architecture to be ready to accommodate the new version of the variability profile, and not to realize this variability. The actual realization of that variability happens at a later step. For instance, suppose a feature X existed in the system before the current increment. If feature Y in the new requirements is just another variation of feature X, then a new variation point is defined. Although there are two different variants X and Y, at this point only the existing variant is considered, not the new variant. Thus, the architecture is refactored to accommodate a variation point with the variant X. This is important to separate the side effects of refactoring from those of adding new functionality.

Testing: To make sure the refactoring process in the previous step did not have any side effects, all the tests in the system are run. This includes executing automated unit tests and acceptance tests as well as running all manual regression tests (usually used to test user interfaces and hardware related functionality). If a test fails, this indicates that the refactoring process needs to be retracted and then fixed to make the tests pass again.

Realization: Having refactored the architecture to be able to realize the new variation (if any), in this step developers implement the new variation. The developers should produce test artefacts either before (using test-driven development) or after writing the production code. All tests for the new variants as well as the older ones have to be run in order to verify and validate the new changes, and to make sure that the old functionality is not impacted by these changes.

6.5 Evaluation through Action Research

6.5.1 Goal and Questions

This section presents a self-evaluation of the approach described above. The goal of the evaluation is to validate the proposed approach against the original problem by applying the approach to the development process of a software application that clearly manifests the problem of interest. The context of this evaluation satisfies the main characteristics of what constitutes AR as described by O'Brien [O'Brien1998][*], namely: the systematic and iterative treatment of the problem at hand, taking into consideration the theoretical foundations, aiming to solve a real problem in a real situation. In this study, the system under change was a real system; and the managed variability was due to requirements coming from a real customer.

Throughout the evaluation, I aim to answer the following questions to assess the approach:

Q1. Can the reactive approach be used to construct a variability profile in an incremental and lightweight manner for a real software application?

[*] The use of the term action research is sometimes restricted to research in industrial settings only. In this work, however, I use the broader understanding of what action research is as described in [O'Brien1998].

Q2. What are the advantages of the reactive approach over other approaches (proactive, clone-and-own, build from scratch, ad-hoc)?

6.5.2 Problem Context

System Overview: The application I discuss throughout this section is called eHome. It is a software system to monitor and control smart homes. Generally, the interface of the application consists of a floor plan representing the smart environment to be controlled, a number of items that can be dragged and dropped on the floor plan, and a set of graphical user interface (GUI) controls. A screenshot is shown in Figure 22.

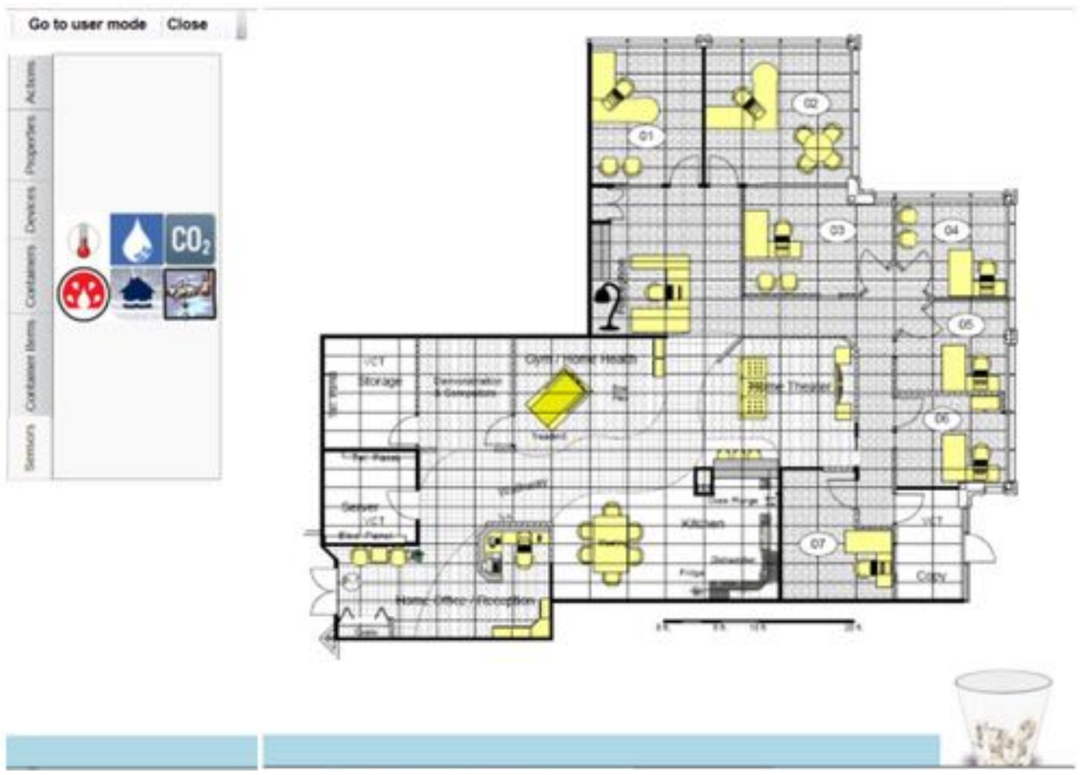


Figure 22 – eHome: a smart home software application

Interacting with eHome occurs in two modes, namely:

a. *User mode:* which allows the dwellers to obtain information about climate variables in the home such as temperature, humidity, CO_2 levels and other sensory information; check the current status of certain devices in the home such as lights being on or off; change the status of devices such as turning lights on and off; and keep track of items in containers such as a fridge or a medicine cabinet using RFID.

b. *Designer mode:* which allows the users to add devices to be monitored and controlled; drop an icon of the device onto the floor plan and attach it to the actual device; add sensors to get climate information; add containers (e.g. medicine cabinet) and add items to the containers (e.g. pill bottles); and define automation triggers and steps.

Initially, the architecture of eHome looked like the one in Figure 23. The Presentation layer included all the view-related elements, whereas the UI Controller managed the communication between the Presentation layer and the Data Object Model. The Hardware Controller was responsible for communication between the actual hardware devices with the Model or the UI Controller. External Resources included the hardware devices, XML configuration files, and web services.

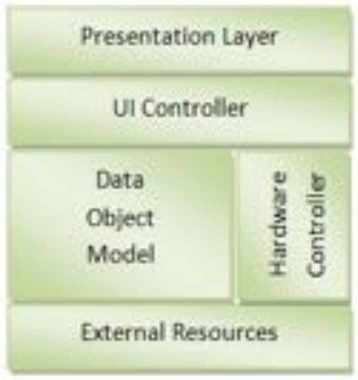


Figure 23 - Initial state of the eHome architecture

Initial Development: The abovementioned features were all requested by our industrial partner. The initial request was to deploy eHome on an HP TouchSmart PC [HP2009] which has a single-touch vertical display. However, actual development of eHome was done on normal PCs with different screen dimensions and no touch capabilities. When we, as a development team, deployed eHome on the HP machine (which happened frequently because we had a testing HP PC onsite), we often needed to adjust certain scaling factors to fit the HP wide screen. We also realized that some decisions that had been made during development on the normal PCs needed to be revisited. Examples are:

- The size and design of some GUI elements made it challenging to interact with eHome using a finger touch because the latter is much thicker and less accurate than a mouse pointer.
- One event in eHome was triggered by a right-click which, on a touch-screen, did not make sense.

New Technologies: As we went along, we deployed eHome on a large-scale SMART DViT Table [DViT2009] using the SMART Board SDK (version 4.1.100.1332 – released on 13-06-2005). A later request from our partner was to deploy eHome on a digital tabletop they had recently purchased. Specifically, it was a multi-touch SMART Table [SMART2009] released with the SMART Table SDK (version 1.3.53.0 – released on 29-10-2009). Later on, we obtained a Microsoft Surface device [Microsoft2011] and we decided to include it within the hardware platforms that we should support. As more platforms were supported, more decisions were revisited and the software design underwent drastic yet incremental changes. These changes were mainly driven by the two non-functional aspects mentioned previously, namely:

- Presentation: e.g. conventional GUI elements like menus and tabs assumed a single orientation (vertical).
- Portability: e.g. three different SDKs that dealt with touch point input, one for each hardware platform.

Sources of Variability in eHome: The presentation and portability issues were not the only sources of variability in eHome. In fact, the first source of variability was business-driven (i.e. due to functional requirements). Smart homes vary widely with regards to what smart devices exist in the home, and what kind of monitoring and controlling is requested by a given customer. This variation in requirements often results in delivering a different application for each smart home. However, in spite of the differences between these applications, they share a lot of underlying functionality and business logic. Therefore, it is more economical to think of these applications as a family of systems that are somewhat similar yet not identical – which is the general understanding of what a SPL is. Nevertheless, in this chapter, I only focus on variability due presentation and portability as non-functional requirements.

6.5.3 Applying the Reactive Approach

When dealing with a new and fast-changing technology like digital tabletops, uncertainty about future needs can be too high. This in turn might render useless any efforts to speculate about these needs. In the development of eHome, big design up-front was avoided; and instead an incremental and reactive approach was followed to develop and maintain variability in the system. On the non-functional aspect, we incrementally embraced new variations as needed, and allowed the common platform to evolve gradually using the approach described above. The following sections illustrate how the approach was followed to satisfy three user stories.

6.5.3.1 User Story 1

"As a user, I should be able to use eHome on a touch-screen."

The developers had normal PCs as their workstations to develop eHome, as opposed to machines with a touch screen. Because the customer wanted eHome to be deployed on an HP TouchSmart PC with a touch screen, a round of variability analysis was needed. The analysis I conducted shows that the differences between the two groups of machines are mainly due to the mouse-versus-touch input. Table 3 shows an issue-implication analysis for the current user story.

Table 3- Variability between a normal PC and an HP Touchsmart PC

Issues	*Implications*
Right-click events do not make sense on a touch screen.	An alternate trigger is to be used. The HP machine captures right-click events on the touch screen using a 'press-&-hold' trigger.

Issues	***Implications***
The tip of the mouse cursor is tiny and accurate compared to the tip of a finger.	All GUI objects have to be larger to accommodate the finger touch more precisely.
When applying a touch on the vertical surface, the body of the finger covers some content on the screen (Figure 24a).	The vertical sliders used to control the intensity of lights should be changed into horizontal sliders (Figure 24b).

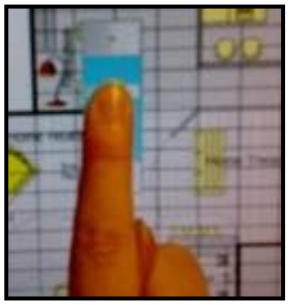
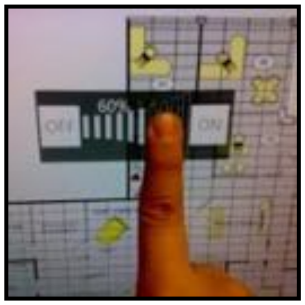
(a)  **(b)** 

Figure 24 - (a) Part of the vertical slider is blocked by the body of the finger, (b) The horizontal slider solves this issue.

Using the list of issues and implications mentioned above, one variation was deduced due to differences in the “Input” requirement. Under this requirement, a single variation point “input mechanism” was defined. The variation point has the two variants “mouse” and “touch”. The variability profile I had so far could be described as:

$$\mathrm{S}_{\mathrm{eHome}} = S_{Common\ Platform} \cup \{\mathrm{Input}\}$$

Where:

$$VP_{\mathrm{Input}} = \{\mathrm{v}\rho_1 = \text{ input mechanism}\}$$

$$V_{\text{input mechanism}} = \left\{ \begin{matrix} \nu_1 = \text{mouse}, \ \nu_2 = \text{touch} : \\ C = \{\text{v1 and v2 are mutually exclusive}\} \end{matrix} \right\}$$

$$\nu_1 \xrightarrow{yields} \{scale\ factor\ x, vertical\ slider, right\ click\}$$

$$\nu_2 \xrightarrow{yields} \{scale\ factor\ y, horizontal\ slider, press\ \&\ hold\}$$

From an architectural perspective, a conceptual layer was added to reflect the updated variability profile as shown in Figure 25. Previously, input was managed within the Presentation layer. At this point, $S_{Common\ Platform}$ included all the layers except those that had sources of variability – namely: the Input Manager layer and the Presentation layer.

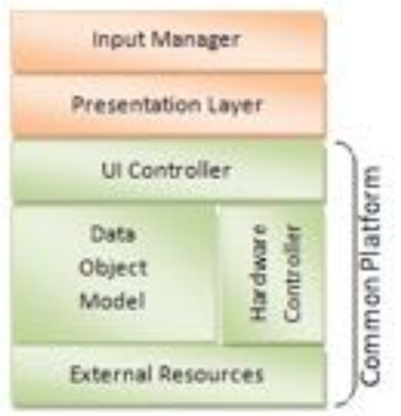


Figure 25 – The impact of the update variability profile on eHome architecture

6.5.3.2 User Story 2

“As a user, I want to be able to use eHome on a large-scale SMART table.”

This user story indicates that the user would like to use eHome – which so far only considers vertical displays – on a horizontal display (i.e. SMART table). This warranted a second round of variability analysis. The analysis started with an initial hypothesis that the system could be migrated to the horizontal display without any changes. The hypothesis was rejected after a number of observations I made regarding presentation as I went back and forth between the vertical display and the horizontal one.

Table 4 captures the results of the issue-implication analysis. For the purpose of this research, I am not interested in finding whether the said implications improve usability; but I use these findings as evidence that presentation issues do introduce usability issues that constitute new sources of variability to be explicated and managed.

Table 4 - Variability between vertical displays and horizontal displays.

Issues	***Implications***
Horizontal displays are, typically, physically larger than vertical ones.	A new scaling adjustment factor should be defined for UI objects to make them bigger, and hence easier to interact with, on larger displays.
Horizontal displays deal with multiple touch points not only single touch points or mouse clicks.	This new input mechanism needs to be incorporated into the Input Manager layer as a new variant.

Issues	*Implications*
Conventional GUI elements like buttons, menus and tabs were oriented in a top-down fashion, which for a horizontal surface did not seem natural because people sit on different sides of the table.	The conventional GUI elements should be replaced by panels available on each of the four sides of the tabletop, as shown in Figure 26. Instead of one Exit button on the top left corner of the screen, an Exit button should be added on each corner of the tabletop. The "change mode" button (user/designer) should be removed. Instead, the change of mode on the digital tabletop can be made so that it is triggered implicitly when the user opens or closes the design panel.
Feedback to the user was provided using a status bar at the bottom of the screen, which was not suitable for a multi-oriented surface (i.e. horizontal display).	Alternative ways to provide feedback are needed. For example, when a certain operation executes successfully, the corresponding icon on the surface glows.

Issues	***Implications***
When using a slider control, vertical and horizontal sliders seemed counterintuitive if there were people sitting around the table (e.g. if you go up in a vertical slider, it seems as if you are going down for a person sitting opposite to you).	A circular slider can be used with clearly flagged ON/OFF positions, as shown in Figure 27. Regardless of where you sit around the table, if the handle of the slider is moving towards the ON button, then the intensity is increasing and vice versa.
Some features were not readily easy to use for everybody around the table because the UI controls were closer to a certain part of the screen.	Instead of a single trash can on the bottom right corner of the screen, redundant cans should be made visible on the corners of the screen when the user touches an object while in the designer mode.
Readability of text on the horizontal display was limited because of the presumed top-down orientation.	The horizontal interface should include far less text than the vertical one. Descriptive icons and UI controls, animations, as well as visual cues like pulsation or glowing are needed to replace text.
With dual-touch capabilities, horizontal displays provided new interactions that were not possible on vertical displays.	On horizontal displays, it should be made possible to zoom in and out of the floor plan using two finger touches.

Issues	***Implications***
On a large-scale tabletop, drag-and-drop became difficult due to the physical limitations on the reach of an arm.	Gestures should be made available as additional (not substitutional) ways of executing certain features. For example, to delete an object, one can use a scratch gesture.

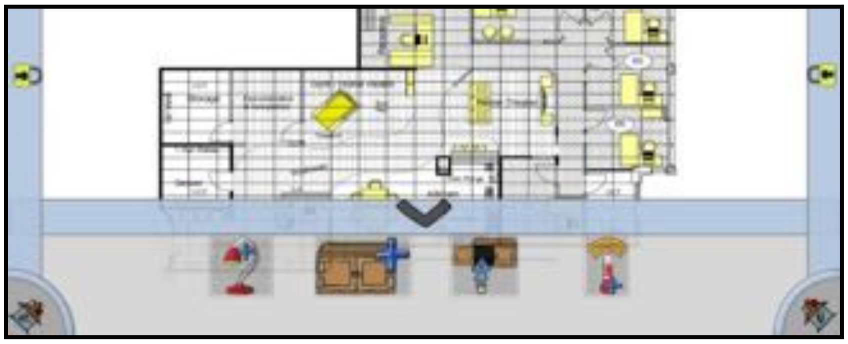


Figure 26 – redundant GUI elements are needed on horizontal displays to support multiple orientations.

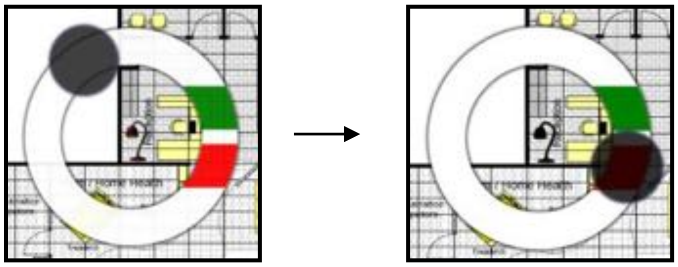


Figure 27 - Circular slider to control light intensity on horizontal displays to support multiple orientations.

Using the list of issues and implications, the variability profile is updated to reflect the new variability sources. With this round of variability analysis, it was clear that variability is occurring due to two factors, namely: the input mechanism being a mouse, a single touch or dual-touch; and the orientation of the display being vertical or horizontal. The update variability profile is as follows:

$$\mathrm{S}_{\mathrm{eHome}} = S_{Common\ Platform} \cup \{\mathrm{Input}, \mathrm{Orientation}\}$$

Where:

$$VP_{\mathrm{Input}} = \{\mathrm{v}\rho_1 = \text{ input mechanism}\}$$

$$V_{\text{input mechanism}} = \begin{Bmatrix} \mathrm{v}_1 = \text{mouse, } \mathrm{v}_2 = \text{single touch, } \mathrm{v}_3 = \text{dual touch :} \\ C = \{\mathrm{v}_1, \mathrm{v}_2 \text{ and } \mathrm{v}_3 \text{ are mutually exclusive}\} \end{Bmatrix}$$

$$\mathrm{v}_1 \xrightarrow{yields} \{scale\ factor\ x, right\ click, vertical\ slider\}$$

$$\mathrm{v}_2 \xrightarrow{yields} \{scale\ factor\ y, press\ \&\ hold, horizontal\ slider\}$$

$$\mathrm{v}_3 \xrightarrow{yields} \begin{Bmatrix} scale\ factor\ z, press\ and\ hold, circular\ slider \\ dual - touch\ gestures\ (zooming) \end{Bmatrix}$$

And:

$$VP_{\mathrm{Orientation}} = \{\mathrm{v}\rho_1 = \text{ orientation}\}$$

$$V_{\text{orientation}} = \begin{Bmatrix} \mathrm{v}_1 = \text{vertical, } \mathrm{v}_2 = \text{horizontal :} \\ C = \{\mathrm{v}_1 \text{ and } \mathrm{v}_2 \text{ are mutually exclusive}\} \end{Bmatrix}$$

$$\mathrm{v}_1 \xrightarrow{yields} \{conventional\ GUI\ controls, textual\ feedback\}$$

$$\mathrm{v}_2 \xrightarrow{yields} \{redundant\ GUI\ controls, textless\ feedback\}$$

From an architectural perspective, at this stage, new variability occurs at the same two layers of the architecture. All the other layers are left intact.

6.5.3.3 User Story 3

"As a user, I want to be able to use eHome on the new SMART table and Microsoft Surface."

In the previous sections, I discussed variability due to differences between vertical displays. I then discussed variability due to the migration of eHome from a vertical display into a horizontal one. This section will discuss variability due to differences between horizontal displays. By horizontal displays, I specifically refer to three hardware platforms: SMART DViT table, new SMART table, and Microsoft Surface. The three tabletops are shown in Figure 28.

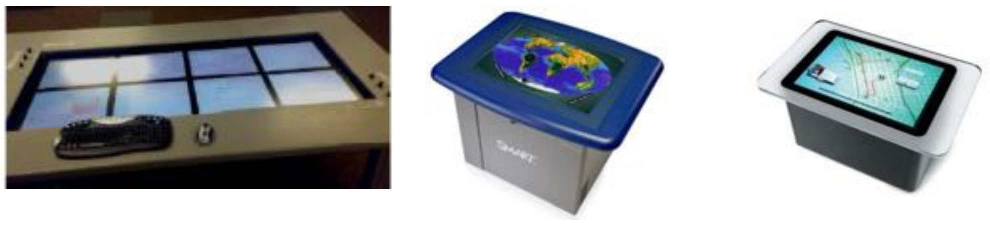


Figure 28 – SMART DViT Table, new SMART Table, and MS Surface (in order).

As illustrated in Table 5 the issues that were taken into consideration are related to the different dimensions, the number of simultaneous touch points, and the different SDKs. Two of the SDKs were different versions from the same vendor.

Table 5 – Variability between horizontal displays

Issues	***Implications***
The aspect ratio (AR = long side/short side) of the SMART DViT table is 2.4 which is very high compared to 1.33 for the new SMART table and 1.56 for MS Surface).	This introduced challenges in treating all four sides of the table equally. That is, on the large-scale SMART DViT table, there are two long sides and two short sides. Therefore, only two control panels can be accommodated – one on each long side of the table.

Issues	***Implications***
	Because of its rectangular shape, the floor plan should not rotate with its full size on the large-scale SMART DViT table except for a full 180 degrees.
Each tabletop uses a different SDK to deal with touch points (SMART SDK old version, SMART SDK new version, Surface SDK).	An abstraction layer is required to embrace the different ways the SDKs deal with touch points.
The number of simultaneous touch points is different for each table (2 for the SMART DViT table, 40 for the new SMART table, and a large unspecified number for MS Surface).	Multi-touch gestures and simultaneous interaction with the system by more than one user should take into consideration the touch capabilities of the device.

The updated variability profile is as follows:

$$\mathrm{S}_{\mathrm{eHome}} = S_{Common\ Platform} \cup \{\mathrm{Input}, \mathrm{Orientation}, \mathrm{Controller}\}$$

Where:

$$VP_{\mathrm{Input}} = \{\nu\rho_1 = \text{ input mechanism}\}$$

$$V_{\text{input mechanism}}$$

$$= \left\{ \begin{matrix} \nu_1 = \text{mouse},\ \nu_2 = \text{single touch},\ \nu_3 = \text{dual touch},\ \nu_4 = \text{multi touch}: \\ C = \{\nu_1, \nu_2, \nu_3, \text{and } \nu_4 \text{ are mutually exclusive}\} \end{matrix} \right\}$$

$$\nu_1 \xrightarrow{yields} \{scale\ factor\ x, right\ click, vertical\ slider\}$$

$$\nu_2 \xrightarrow{yields} \{scale\ factor\ y, press\ \&\ hold, horizontal\ slider\}$$

$$\nu_3 \xrightarrow{yields} \left\{\begin{matrix} scale\ factor\ z, press\ and\ hold, circular\ slider \\ dual - touch\ gestures\ (zooming) \end{matrix}\right\}$$

$$\nu_4 \xrightarrow{yields} \left\{\begin{matrix} scale\ factor\ z, press\ and\ hold, circular\ slider \\ multi - touch\ gestures\ (zooming) \end{matrix}\right\}$$

And:

$$VP_{\text{Orientation}} = \{\nu\rho_1 = \text{ orientation}\}$$

$$V_{\text{orientation}} = \left\{\begin{matrix} \nu_1 = \text{vertical, } \nu_2 = \text{horizontal high AR , } \nu_3 = \text{horizontal low AR, :} \\ C = \{\nu_1, \nu_2 \text{ and } \nu_3 \text{ are mutually exclusive}\} \end{matrix}\right\}$$

$$\nu_1 \xrightarrow{yields} \{conventional\ GUI\ controls, textual\ feedback\}$$

$$\nu_2 \xrightarrow{yields} \left\{\begin{matrix} redundant\ GUI\ controls\ on\ two\ sides, \\ 180\ rotation\ only, textless\ feedback \end{matrix}\right\}$$

$$\nu_3 \xrightarrow{yields} \left\{\begin{matrix} redundant\ GUI\ controls\ on\ four\ sides, \\ full\ rotation, textless\ feedback \end{matrix}\right\}$$

And:

$$VP_{\text{Controller}} = \{\nu\rho_1 = \text{ SDK}\}$$

$$V_{\text{SDK}} = \left\{\begin{matrix} \nu_1 = \text{SMART DViT, } \nu_2 = \text{new SMART, } \nu_3 = \text{MS Surface, } \nu_4 = \text{operating system :} \\ C = \{\nu_1, \nu_2, \nu_3 \text{ and } \nu_4 \text{ are mutually exclusive}\} \end{matrix}\right\}$$

$$\nu_1 \xrightarrow{yields} \{SMART\ DViT\ SDK\}$$

$$\nu_2 \xrightarrow{yields} \{New\ SMART\ SDK\}$$

$$\nu_3 \xrightarrow{yields} \{MS\ Surface\ SDK\}$$

$$\nu_4 \xrightarrow{yields} \{native\ OS\ libraries\}$$

From an architectural perspective, the portability that was needed for eHome required a number of refactoring steps in the architecture. The first tabletop on which eHome was deployed was the SMART DViT Table. I utilized the dual-touch capability of this table by adding a feature that allowed the user to place two touch points on the floor plan to zoom in and out. This kind of interaction required the hardware platform to support at least two simultaneous touches, which made the interaction irrelevant to the previous hardware platforms that either did not have support for touch interactions or had support for only one touch. For this reason, I chose not to include this interaction with the rest of the interactions in eHome that were common to all platforms. Rather, a specialized controller was introduced in the UI Controller layer to manage all communication between eHome and the touch handlers in the SMART SDK, as shown in Figure 29 – A. By this separation, it was easier to plug this feature in and out. The new controller was responsible for managing three events, namely: TouchDown, TouchUp and TouchMove. In case the touch events were part of a zooming interaction, the specialized controller will handle the zooming. Otherwise, the touch events were rerouted to mouse events I had previously defined in the UI Controller for the previous platforms in order to maximize code reuse and avoid code redundancy.

The second step was deploying eHome on the New SMART Table. The New SMART Table came with its own SDK, and the technology was different from the older table. Therefore, a new specialized hardware controller was also created to manage communication between eHome and the touch handlers in the new SMART SDK. At this stage, I had two different controllers one for each table. These controllers, however, shared common aspects such as the main triggering events and the zooming interaction.

These common aspects were abstracted in a new layer I called “Multi-Touch Library” as shown in Figure 29 – B. The new layer was abstracted in a way so that it was completely agnostic to the target hardware platform – all specificities were kept in the specialized controllers.

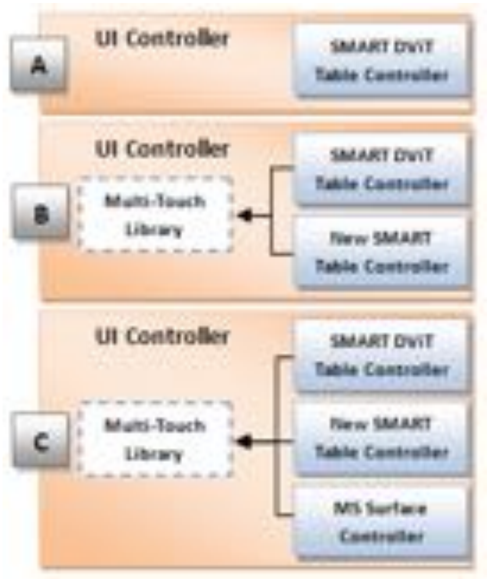

A
UI Controller
SMART DViT Table Controller
B
UI Controller
Multi-Touch Library
SMART DViT Table Controller
New SMART Table Controller
C
UI Controller
Multi-Touch Library
SMART DViT Table Controller
New SMART Table Controller
MS Surface Controller

Figure 29 – Refactoring due to variability in the SDKs.

Later on, this abstraction served well in accommodating the new digital tabletop – MS Surface. That is, it only took about one day worth of work to deploy eHome on MS Surface, because all I needed to do was create a new specialized controller to communicate with the Surface SDK, while all other aspects were managed by the Multi-Touch Library. Figure 29 – C shows the final organization. After this step, $S_{Common\ Platform}$ no longer included the UI Controller layer.

6.6 Discussion of Results

In the previous sections, I discussed an approach to reactively manage variability in systems due to presentation and portability requirements. A detailed validation of the approach was presented next against an application called eHome which clearly manifested variability arising from presentation and portability aspects. In this section, I discuss the results of the evaluation in light of the two assessment questions.

6.6.1 Q1. Can the reactive approach be used to construct a variability profile in an incremental and lightweight manner for a real software application?

The proposed approach was successful in constructing a variability profile for eHome. As seen in the case of eHome, the initial variability profile started in a very simple form consisting of only one variation point with two variants. But as more requirements became clearer, the variability profile was updated with more variation points, variants and constraints. The updates were reflected on the system architecture on demand. The incremental building of the variability profile was supported by a lightweight, issue-implication analysis that normally did not take more than one or two sessions for the development team to accomplish. The issue-implication analysis gave focus to the discussions held throughout the sessions.

The produced variability profile is fully capable of being utilized in the product-instantiation process. For instance, to produce a product that is specific to MS Surface, I use the generic formula:

$$\mathrm{S}_{\mathrm{eHome}} = S_{Common\ Platform} \cup \{\text{Input, Orientation, Controller}\}$$

Where:

For $\nu\rho_1 \in VP_{\text{Input}}$, I select: $\nu_4 = \text{multi touch} \in V_{\text{input mechanism}}$, and

For $\nu\rho_1 \in VP_{\text{Orientation}}$, I select: $\nu_3 = \text{horizontal low AR} \in V_{\text{orientation}}$, and

For $\nu\rho_1 \in VP_{\text{Controller}}$, I select: $\nu_3 = \text{MS Surface} \in V_{\text{SDK}}$

This formal representation is then fed to an instantiation engine through a configuration file or any other mechanism in order to start the instantiation of a specific product.

6.6.2 Q2. What are the advantages of the reactive approach over other approaches (proactive, clone-and-own, build from scratch, ad-hoc)?

Just-Enough Variability

One of the main advantages of being reactive versus being proactive in building variability profiles and acting upon them is the greater ability to justify investment in variability. The issue-implication analysis was only conducted when there was an actual demand to do so. It also promoted the idea of just-enough variability. That is, unless an item could be described as an 'issue' with immediate variability 'implications', the item would not be considered in the variability profile. This ensured that the variability profile is not over-engineered and that it only reflected what is currently needed and is to be implemented in the architecture. In the case of proactive treatment, there are generally no guarantees as to what variants will actually be needed in the market. This results in lost investments if any variant was included in the profile but never used. Also, the risk of speculation is considerably higher when the involved technology is still emerging or the domain is unstable. In the case of eHome, this issue was clearly evident as the horizontal display technology was relatively new. Had we invested heavily in a variability profile up-front, our speculations would have been unlikely to project the new changes in the SMART SDK or the differences between MS Surface and other tabletop technologies.

Opportunistic Reuse & Common Repository

In the case of eHome, about 60% of the code (production and testing) is reused amongst all platforms, which is a big advantage over building separate applications from scratch.

This figure could even be higher for systems that have a thinner presentation layer than the one in eHome. Maximizing reuse is desirable because it lessens the time and effort to produce new products and maintain existing ones.

Moreover, the approach is also superior to clone-and-own techniques in two ways. For one, in clone-and-own techniques, more than one repository exists for similar products; whereas using this approach, only one repository exists for all hardware platforms. This is advantageous because if we need to change a feature or fix a bug in the system, we only need to make the proper modifications once, then re-instantiate different products for the different platforms we support. The process of product instantiation will be discussed more concretely in chapter nine. Also, say a vendor produced a new digital tabletop technology. All we need to do is add a new variant in the UI Controller layer. All other parts of the system are left unchanged. The second advantage of the systematic treatment of variability in the reactive approach over clone-and-own techniques is the ability to combine different variants to come up with diverse products. For example, suppose we want to support a new HP TouchSmart PC that enables two simultaneous touches. We can come up with a new combination of variants as follows:

$$\mathrm{S}_{\mathrm{eHome}} = S_{Common\ Platform} \cup \{\text{Input}, \text{Orientation}, \text{Controller}\}$$

Where:

For $\nu\rho_1 \in VP_{\text{Input}}$, we select: $\nu_3 = \text{dual touch} \in V_{\text{input mechanism}}$, and

For $\nu\rho_1 \in VP_{\text{Orientation}}$, we select: $\nu_1 = \text{vertical} \in V_{\text{orientation}}$, and

For $\nu\rho_1 \in VP_{\text{Controller}}$, we select: $\nu_4 = \text{operating system} \in V_{\text{SDK}}$

That is, by choosing different variants for the variation points, we ended up with a customized product for the new platform. Constraints are usually defined to filter out invalid combinations.

Moreover, compared to ad-hoc techniques, treating variability in this systematic manner has a great value. For instance, before deciding to support a new hardware platform, we need to know what is different about the new platform that cannot be supported by the existing product line. If there is any difference, then decisions need to be taken on where in the architecture this variation should be accommodated and what impact it will have on other platforms in the family. Without having an explicit variability profile of the SPL, taking such decisions becomes more difficult and is accompanied with higher risks.

Having said that, it is important to point out that some of the advantages mentioned above are inherited from the SPL practice in general. Nonetheless, using the reactive, lightweight approach allows organizations to realize the same advantages in a way that is more cost effective (because it is lightweight) and less risky (because it minimizes speculation), and with a faster return on investment (because systems are continuously delivered as opposed to waiting until the application engineering phase).

6.6.3 Limitations

In traditional SPL engineering, domain engineers assume the role of eliciting, managing and updating variability. On the contrary, the approach described in this chapter lacks a clear definition of the roles needed in the different steps. For example, who in a typical agile organization should conduct the variability analysis? Can developers assume the responsibility of updating the variability profile? This is vital because variability analysis and profiling require a wide knowledge of existing requirements in the system. Therefore, a developer who only worked on a certain aspect of the system may not be qualified for this role, nor is a project manager who is not completely familiar with other projects in the company. Unless variability management is conducted as a collaborative activity that involves program managers, project managers, architects and developers, it is difficult to make predictions as to how effective and practical the approach will be in an organizational context.

Also, in the case of eHome, the number of variation points was small. If the approach is to be applied on a large-scale system with many variation points, many variants and many constraints, the scalability of the approach may become an issue as the variability profile undergoes drastic increases in complexity. Scalability has always been an issue in variability management [Chen2009b].

Another limitation of the approach is the subjectivity in the decision-making process that might affect the systematic aspect of the approach. For example, deciding which variant should include what presentation aspect can be tricky sometimes. In the case of eHome, I decided that the "circular slider" should be associated with the variation due to VP_{Input}. This makes sense because the idea of a slider that is different from the vertical one is a result of the variation coming from the different touch capabilities. Nevertheless, it can be argued that the circular slider should be associated with the variation coming from $VP_{\text{Orientation}}$ to clearly communicate that the aspect is a direct implication of variability due to differences in the orientation. Deciding where to associate a given aspect may limit the ability to form certain combinations of products at a later stage. For example, with the decision I made regarding the circular slider, I will not be able to support a multi-touch vertical display with a horizontal slider. When the need to do so arises, the decision will have to be revisited.

It is also important to point out that the reactive approach presented in this chapter only considered a specific architectural style – namely, a layered architecture. For other architectural styles including plugin-based architectures as seen in highly extensible products such as Eclipse, Facebook, and Android phones, a proactive approach for core features and services might be more sensible. Also, variability in the presented study was approached by conducting small changes to the user interface in an incremental manner. This treatment is useful in many situations; however, it does not consider cases where a new device might warrant for a completely different design or an entirely different approach to usability. In such cases, the user interface – in its entirety – should constitute a variation point with a separate variant for each device. Each variant would have a distinct user interface and a set of interaction techniques.

Furthermore, there exist some threats to the validity of the evaluation presented in this chapter. These threats come mainly from the fact that it is a self-evaluation where bias and subjectivity are inevitable. An attempt was made to reduce this bias by involving individuals other than the researcher in the development and analysis of the studied case (eHome). The second validity concern arises from the specificity of the domain which might deteriorate the generality of the findings. The application domain used to validate the approach is characterized by the volatility of its pertinent technologies which made it a perfect fit for evaluating the reactive aspect of the approach. Moreover, the application had a thick presentation layer with novel interaction techniques which also might have exacerbated the significance of presentation as a non-functional aspect.

6.7 Chapter Summary

This chapter presented an approach to reactively elicit and evolve variability with a focus on non-functional aspects. Specifically, I discussed variability arising from variations in presentation and portability requirements. I showed how variability profiles could be built in an incremental manner using a lightweight issue-implication analysis. Using action research as a research tool, I presented a self-evaluation of the approach against the original problem. The approach was in fact successful in constructing variability profiles incrementally, and it provided advantages over other approaches but with its own set of limitations.

CHAPTER SEVEN: VARIABILITY MODELING*

7.1 Preamble

In the previous chapters, I showed how variability in business logic requirements as well as in presentation and portability requirements can be elicited and evolved in an incremental and lightweight manner. This chapter sheds more light on the issue of variability modeling. That is, having constructed variability profiles in the previous steps, we need to take further steps to make variability visible to different stakeholders in the organization. Modeling variability with feature trees is one way this could be achieved. Nevertheless, feature trees do not natively provide a means to trace the requirements at the feature level to the implementation at the code level. Traditional variability management approaches address this issue by using intermediary requirement and design artefacts to ensure consistency between the model and the implementation. In this chapter, I discuss why this approach is not ideal especially in an agile context. Then, I address the problem in a different way and I provide a comparative evaluation.

7.2 Research Instruments

In this chapter, I investigate how variability can be modeled in a way that takes into consideration the absence of traditional requirement and design documents, and ensures consistency between the model and the implementation at all times. I specifically look at how executable acceptance tests (EATs) can be used to serve this purpose. The research question I address in this regard is as follows:

---

* This chapter is based on a published paper [Ghanam2010c]. Co-author permission is attached to Appendix B.

RQ3. How can EATs be used to model variability in a system so that variability becomes communicable across the organization and traceable to the implementation?

In the context of the framework I propose, this research question addresses Stage C: Variability modeling as shown in Figure 30. For this stage, I assume that a variability profile has already been constructed in the previous stage, and it can be translated into a feature tree. This implies that the modeling approach discussed throughout this chapter is concerned with functional requirements only – since translating non-functional variability profiles into feature trees is a challenge as discussed in the previous chapter.

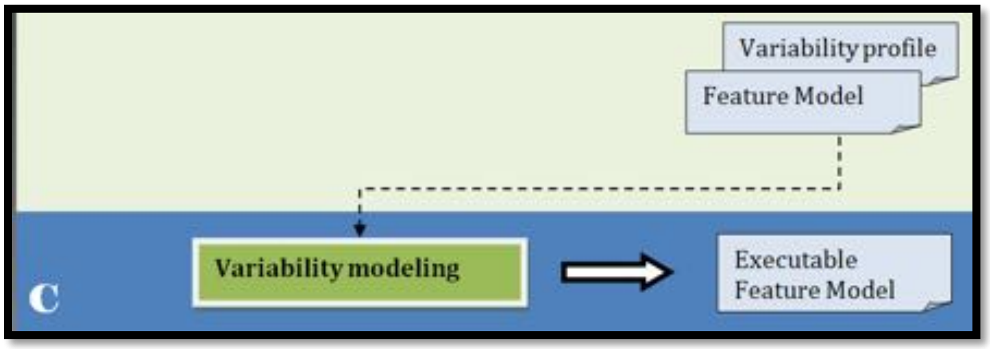


Figure 30 – This chapter tackles Stage C: Variability modeling

Two research instruments are used to evaluate the research presented in this chapter. Firstly, I use comparative evaluation [Vartiainen2002] where a comparative framework is built using criteria from the literature and then used to evaluate the proposed approach in comparison to other traditional approaches. Secondly, I use a running example to illustrate the advantages and identify the limitations of the approach. Using running examples is a well-accepted research technique in the absence of opportunities to apply large-scale research (e.g. SPL research) in an industrial context over a long term (e.g. [Tun2009, Parra2009, Cho2008]).

7.3 Preliminary Analysis

7.3.1 Feature modeling

A feature model is a representation of the requirements in a given system abstracted at the feature level [Riebisch2003]. In variability management, feature models are used in a tree-like format to represent a hierarchy of features and sub-features in a product line. Typically, the feature tree includes notations to describe where variation exists, and the relationship between the different variants. For example, Figure 31 shows a feature tree for a home security system. The white circles indicate that a feature is optional, whereas the arch indicates different alternatives that might be governed by a constraint to control their selection during the instantiation process.

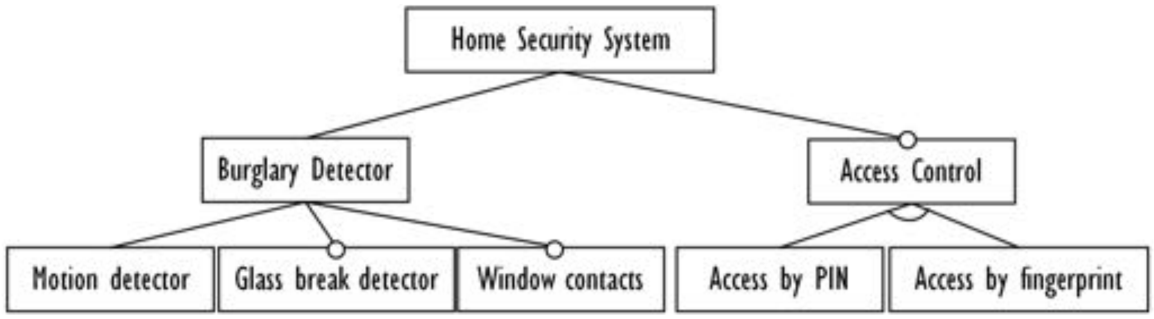


Figure 31 - A feature tree for a home security system

Linking conceptual requirements in feature models to actual implementation artefacts provides advantages such as increased program comprehension, implementation completeness assessment, impact analysis, and reuse opportunities [Antoniol2002]. Nevertheless, traceability is a non-trivial problem. Berg et al. [Berg2005] analyzed traceability between the problem space (i.e. the model) and the solution space (i.e. the development artefacts) in a SPL context. The results suggested that the feature model provided an excellent visualization means at individual levels of abstraction. However, it did not improve the traceability between artefacts across development spaces. Furthermore, in practice, as the product line evolves, traceability relationships between the model and the code artefacts may become broken or outdated [Riebisch2004]. This happens either because changes in the model are not completely and consistently realized in the code artefacts; or because changes due to continuous development and maintenance of the code artefacts are not reflected back in the model. This problem is not unique to SPLs. In fact, outdated traceability between requirement specifications and other development artefacts has always been an issue in software engineering [Gotel1994, Cleland-Huang2004]. Also, in an agile context, artefacts that are typically needed to achieve traceability (e.g. UML design documents, specification documents) are minimal or even absent.

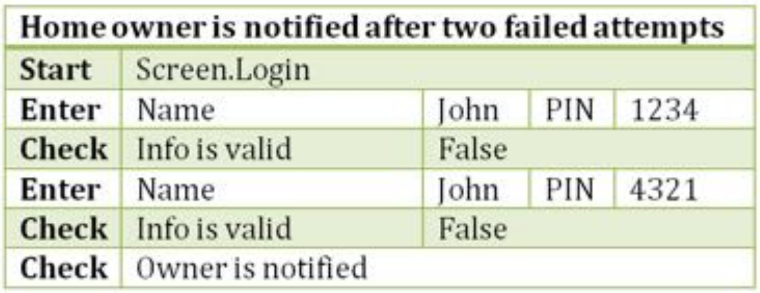
Traceability links provided by some commercial tools (e.g. DOORS [DOORS2010]) mitigate this issue, but leave some other problems unsolved. For example, say feature A and feature B are independent features in the product line. During the maintenance of feature A, the developer introduced a change that unintentionally caused a technical conflict between feature A and feature B. Although the tool will maintain the traceability links between each piece of code and the correspondent feature, it cannot, uncover the newly introduced conflict in order to reflect it back in the model.

To address the issues mentioned above, I investigate the use of EATs as a direct traceability link between feature models and code artefacts. The next section elaborates more on EATs and their characteristics.

7.3.2 Executable Acceptance Tests (EATs)

In agile software development, the specifications of the system are captured in the form of user stories. These stories are then translated to tests that specify the acceptance criteria of a given user story [Cohn2004]. ATs aim to reduce ambiguities and inconsistencies found in traditional specification documents, particularly when they are made executable. An example is shown in Figure 32. The two main characteristics of EATs that support my proposal to use them as a communication medium are the following:

- Because EATs are essentially deduced from user stories, they exhibit a language that is readable to non-technical stakeholders. Therefore, they serve as cohesive documentation of the specifications of a given feature.
- Being executable, EATs can be run (executed) directly against the system in order to test the correctness of its behaviour. Therefore, they serve as accurate and up to date validity tests.

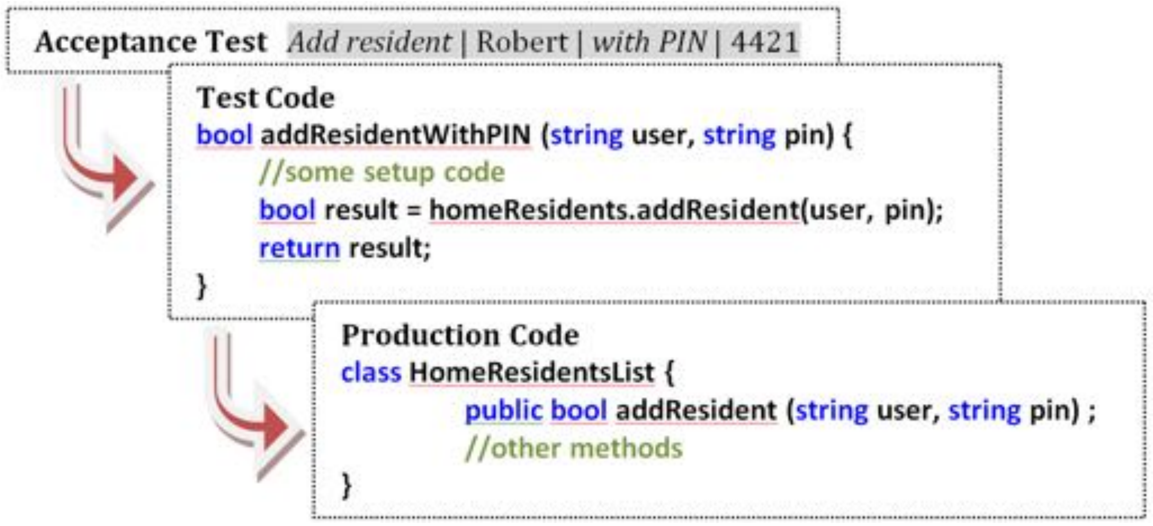


Home owner is notified after two failed attempts				
Start	Screen.Login			
Enter	Name	John	PIN	1234
Check	Info is valid	False		
Enter	Name	John	PIN	4321
Check	Info is valid	False		
Check	Owner is notified			

Figure 32 - Example of an EAT

7.3.3 Traceability from EATs to Code Artefacts

The fundamental basis of the approach I describe in this chapter is that EATs natively provide the necessary links to code artefacts. ATs can be made executable against the system by linking them to a thin layer of test code (aka. a fixture), and from there to actual production code. Figure 33 shows an example of this traceability. At the first layer, only one row of an EAT is shown for simplicity. This row is linked – by a test automation framework (e.g. FIT) – to a method in the test code called *addResidentWithPIN()*. This method in turns uses the *addResident()* method in the production code, specifically in the *HomeResidentsList* class. When the test is executed, an attempt to add a resident with the given parameters will be made. In this scenario, if the attempt is not successful – for a variety of reasons such as the PIN being too short or too long – the EAT will fail. Otherwise, it will pass. Usually, a suite of EATs is executed rather than a single EAT. Moreover, with appropriate test coverage, tools generate reports stating which methods were involved in the execution process of a given EAT. In the remaining of this chapter, I discuss how this traceability is useful in linking features models to code artefacts.


Acceptance Test *Add resident* | Robert | *with PIN* | 4421

Test Code
```
bool addResidentWithPIN (string user, string pin) {
        //some setup code
        bool result = homeResidents.addResident(user, pin);
        return result;
}
```

Production Code
```
class HomeResidentsList {
                public bool addResident (string user, string pin) ;
                //other methods
}
```

Figure 33 - Traceability through EATs

7.4 Using Feature Models with EATs

I propose extending feature models by including EATs as concrete descriptors of features at the lowest level of the feature tree. EATs should be associated with features that originally would be considered leaf nodes in the tree as shown in Figure 34. For instance, the feature "Access by PIN" is associated with three EATs. These EATs describe scenarios that need to be satisfied in the implementation of this specific feature.

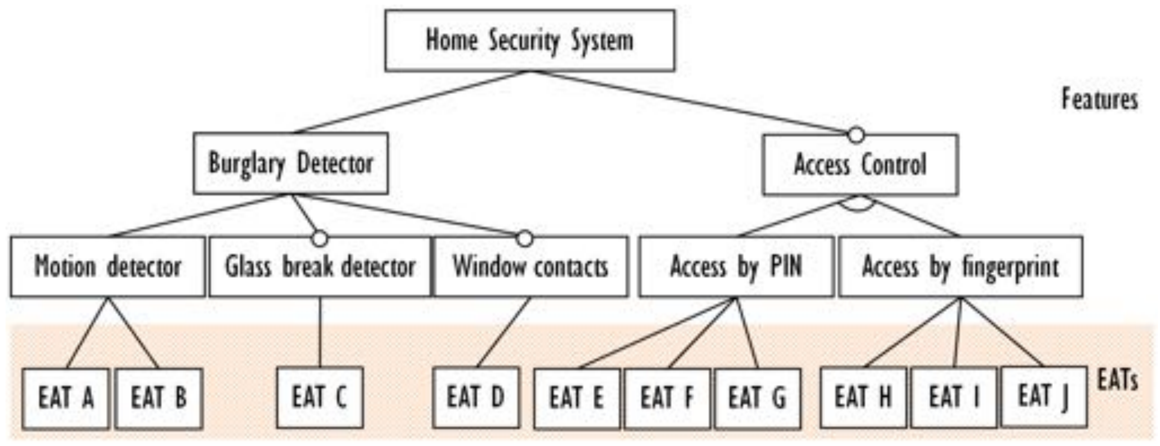


Figure 34 - The proposed extension to feature models

Linking between an EAT node in the model and the actual specification happens by associating a test unit to the EAT node. An EAT node can link to a test table, a test page, or a test suite. No constraints are put on the granularity of the test unit to leave it flexible for various contexts. Nevertheless, a single test table may be insufficient given that typically more than one table is needed to specify some behaviour. This makes a single table less cohesive than desired. On the other hand, a test suite may be too large because it involves more than one feature creating dependencies between test units. Therefore, to achieve reasonable cohesion and independence I suggest the use of a test page as a usual test unit which in turn may include one or more test tables. For example, EAT G can be linked to a single test page that includes three test tables. Depending on the testing tools, test pages can take various formats such as html files or excel sheets.

Following the earlier definition of a feature as a chunk of functionality that delivers value to the end user, one EAT generally is not sufficient to represent a feature in a system. In practice, a group of EATs represent the different scenarios or stories expected in a given feature in a system. This implies that in order to somehow link features in a feature model to EATs, one-to-one relationships are not practical. Rather, each feature in the feature model should be linked to one or more EATs as depicted in Figure 35.

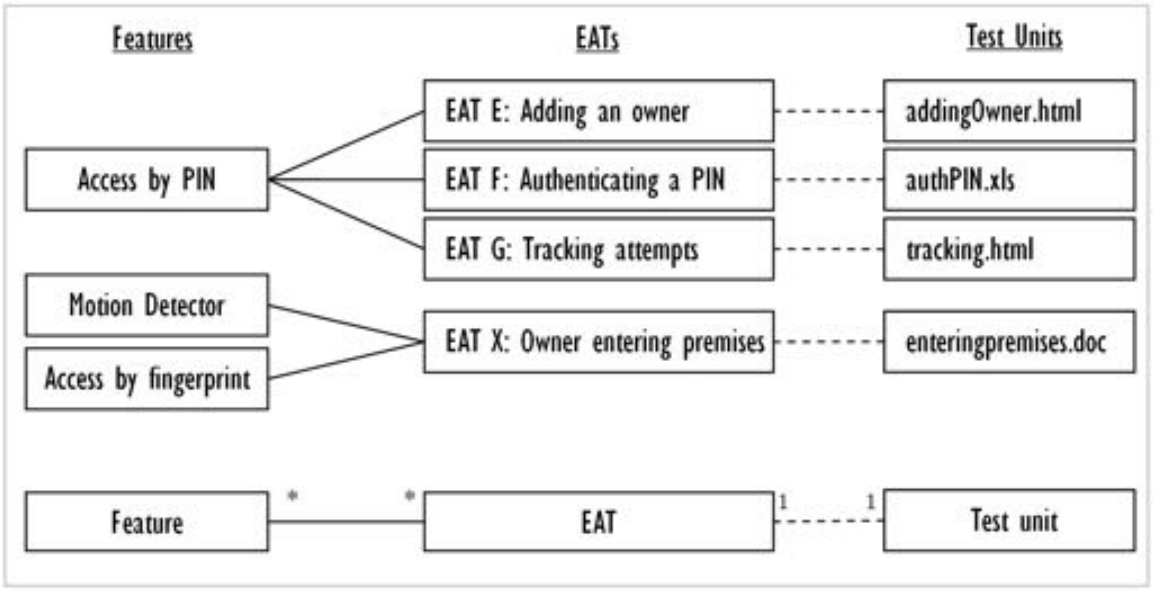


Figure 35 - Relationships between features, EATs, and test units

The "Access by PIN" feature is specified using three EATs. In order for the behaviour of this feature to be deemed correct, all three EATs should pass. Moreover, in some cases, a single EAT can be at a level high enough to cut across a number of features in the system. Consider, for example, a high-level EAT such as "Owner entering premises". Say in order for the scenario specified in this EAT to pass, more than one feature should be involved (i.e. EAT X cuts across a number of features). This implies that a many-to-many relationship is needed in order to accurately represent the relationship between EATs and features in a feature model.

Linking features to EATs has the following consequences:

1. The selection of a feature in the product derivation phase automatically implies the inclusion of all its EATs. For example, if the customer chooses to have the “Access by PIN” feature, this implies that all EATs in the group {EAT E, EAT F, EAT G} should pass.
2. EATs shall inherit all the dependencies and constraints originally imposed on their parent nodes. For example, according to the model in Figure 34, the two features “Access by PIN” and “Access by fingerprint” are mutually exclusive. This implies that the groups: {EAT E, EAT F, EAT G} and {EAT H, EAT I, EAT J} are mutually exclusive too.

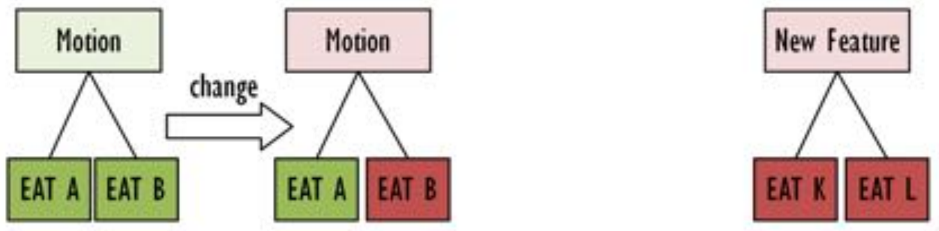
7.5 Implications of Using EATs as Traceability Links

In the previous sections, I showed how features in the feature model can be linked to EATs in order to provide traceability links between the feature model and the code artefacts. This section analyzes the implications of using EATs by highlighting three main ways through which EATs provide significant contribution to feature models.

7.5.1 Consistency between the Feature Model and the Code Artefacts

EATs provide a means to ensure that the problem space (i.e. the specifications), and the solution space (i.e. the implementation) are consistent. This consistency is due to the fact that these specifications can be executed against the implementation, and the result of their execution gives an unambiguous insight of whether or not the intended requirements currently exist in the system. Within a SPL context, the following advantages are identified:

Continuous two-way feedback. Maintaining a practice where every feature in the feature model has to be associated with some EATs provides certain advantages. Changes due to continuous development and maintenance of the code artefacts are reflected back in the model, because – at any point of time – the EATs are either in a passing state (visualized as green) or a failing state (visualized as red). For instance, Figure 36 shows how a change in the code (e.g. bug fix) caused EAT B to fail – also causing the "Motion Detector" to be denoted as incomplete.



(a) a change in the code caused an EAT to fail providing immediate feedback in the feature model.

(b) when adding a feature to the model, initially EATs fail indicating incomplete implementation.

Figure 36 - Continuous two-way feedback

The opposite direction of feedback occurs when introducing a new feature to the model. The accompanied EATs will initially be in a failing state indicating that the feature is not implemented yet.

Exploiting hidden variability concerns. Using EATs helps in revealing unwanted feature interactions that otherwise might be hidden. It also supports the realization of common aspects of features. I illustrate these points further by going through a number of scenarios.

<u>*Scenario 1*</u>: In some cases, the same EAT can be used as part of the specifications of two different features. If the features are originally mutually exclusive, and the same EAT passes in both, then this EAT is agnostic to the source of variation in the features. This means that the specifications in this EAT are part of the common portion of the parent node, which exploits a commonality aspect that was not originally apparent. Figure 37 shows that because EAT G and EAT J are the same (I use a dashed line to denote this – it is also possible to give them the same name), it is possible to abstract the commonality as a mandatory sub-feature under "Access Control".

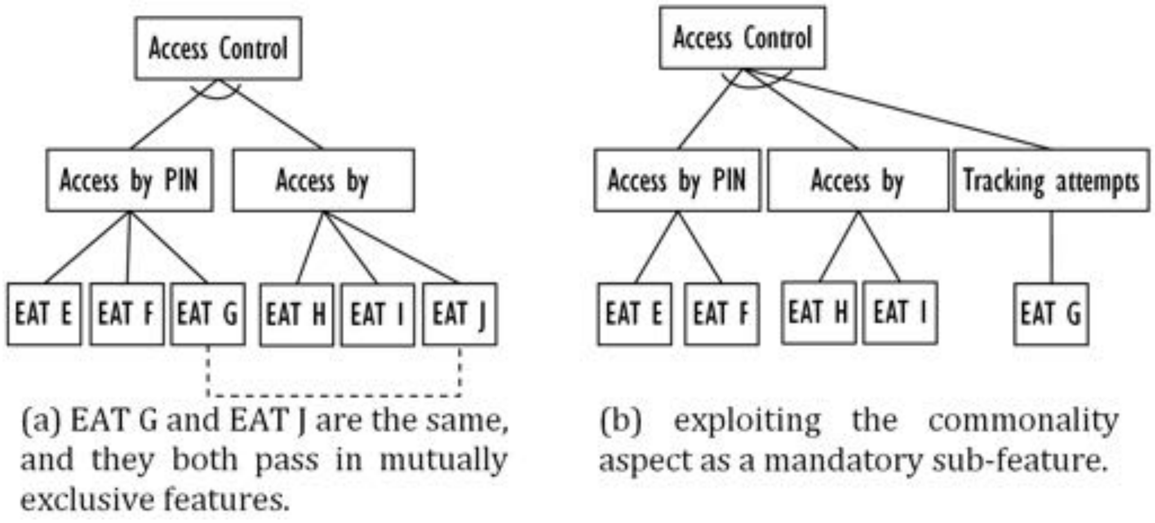

Access Control
Access by PIN
Access by
EAT E
EAT F
EAT G
EAT H
EAT I
EAT J
(a) EAT G and EAT J are the same, and they both pass in mutually exclusive features.
Access Control
Access by PIN
Access by
Tracking attempts
EAT E
EAT F
EAT H
EAT I
EAT G
(b) exploiting the commonality aspect as a mandatory sub-feature.

Figure 37 – Abstracting the commonality as a mandatory sub-feature

<u>*Scenario 2*</u>: Using EATs allows finding unwanted feature interactions. EATs for independent features may pass when the features are selected separately; but fail when selected together. This is indicative of an unwanted feature interaction. This conflict is either a problem in the implementation and should be resolved, or an unavoidable real conflict that should then be reflected in the model as an "excludes" dependency or using a multiplicity constraint.

Scenario 3: Some EATs for independent features fail when these features are selected separately, but when selected together, they pass. This is indicative of a dependency between the features. It can be either due to unnecessary coupling in the implementation itself that should be resolved, or due to a necessary "requires" dependency that should then be reflected in the model.

7.5.2 Supporting the Evolution of Variability in the Extended Feature Model

Using EATs as a basis for evolving variability in the feature model is rewarding in a number of ways. Consider the following scenarios:

Scenario 1: A new feature or sub-feature is added to the feature model. In case the newly added feature causes EATs of other features that were originally passing to fail, this is a sign that a new conflict was introduced by the new feature. Without the direct feedback of failing tests, it is less likely for this conflict to be immediately exposed.

Scenario 2: An existing feature or sub-feature is removed from the feature model. If this feature was originally related to other features, then all dependencies are to be resolved before removing the feature safely. However, in case there was a hidden (unexploited) dependency between this feature and other features, removing this feature and its corresponding code might have a destructive effect on the other features. The fastest way to discover such effects is by looking for EATs that started to fail only after removing the feature.

Scenario 3: A new variant is to be added to a group of variants under a given feature. For developers, using EATs provides guidance on where and how this new variant should be accommodated in the system. For example, suppose we want to add a new alternative "Access by Magnet Card" under "Access Control". First of all, we may be able to reuse the EATs of the other sibling alternatives and tweak them to reflect the requirements of the new alternative. And because EATs are traceable to code artefacts, one can look at the implementation of the sibling alternatives in order to have a better understanding as to where in the code the new variant should be incorporated, and how it should be handled. With appropriate tool support, we can also automate the process of adding a variant by using the sibling nodes as templates, and directing the developer to the exact place in the code base where the new logic should be added (as will be explained in chapter eight). This is particularly important for legacy systems with poor or outdated design documentation or for development environments where design documentation might not be available at all.

Scenario 4: Abstracting a variability aspect to the common layer. Say an EAT is used as part of the specifications of two mutually exclusive features, and this EAT passes in both. This means that the specifications in this EAT can be abstracted to become part of the common layer of the parent node as a mandatory sub-feature (this was also mentioned in the previous section).

7.5.3 Deriving Products using the Extended Feature Model

In a SPL context, feature models are used to select features and variants that constitute a product instance. The selection process should take into consideration the constraints and dependencies between features and variants, as conveyed in the feature model. Nowadays, tool support is available to make this process easier, faster and less error-prone. Once the features and configurations have been selected, an instance is derived that has the required feature composition and configuration.

The extended feature model described in this chapter provides great benefits in this regard. It provides support for two different methods of product derivation, namely: selecting configurations, and extracting required artefacts**.** A dedicated chapter on product derivation (chapter nine) will elaborate on these methods in greater detail.

7.6 Tool Support

In order to realize the benefits discussed in the previous sections, we built a tool that supports traceability links between the feature model and code artefacts via EATs [Riegger2010][*]. To avoid reinventing the wheel, an open-source modeling tool was chosen in order to be extended. We used Feature Model DSL as the basis (available online [André2010]). The tool provides a feature modeling toolbox integrated in the Visual Studio environment. It includes a visual designer to create and modify models. It also provides a configuration window that allows the creation of configurations based on the feature model. We extended the tool in two ways, namely: allow the linkage between features and EATs, and define a course of action to complete the derivation process of individual instances after the configuration process. The remaining of this section will explain the currently available features.

The user can model features and the relationships between them following the typical feature modeling notation as shown in Figure 38. Each node in the tree represents a feature or a sub-feature including options and alternatives.

[*] The implementation of the tool was done collaboratively with Felix Riegger [Riegger2010].

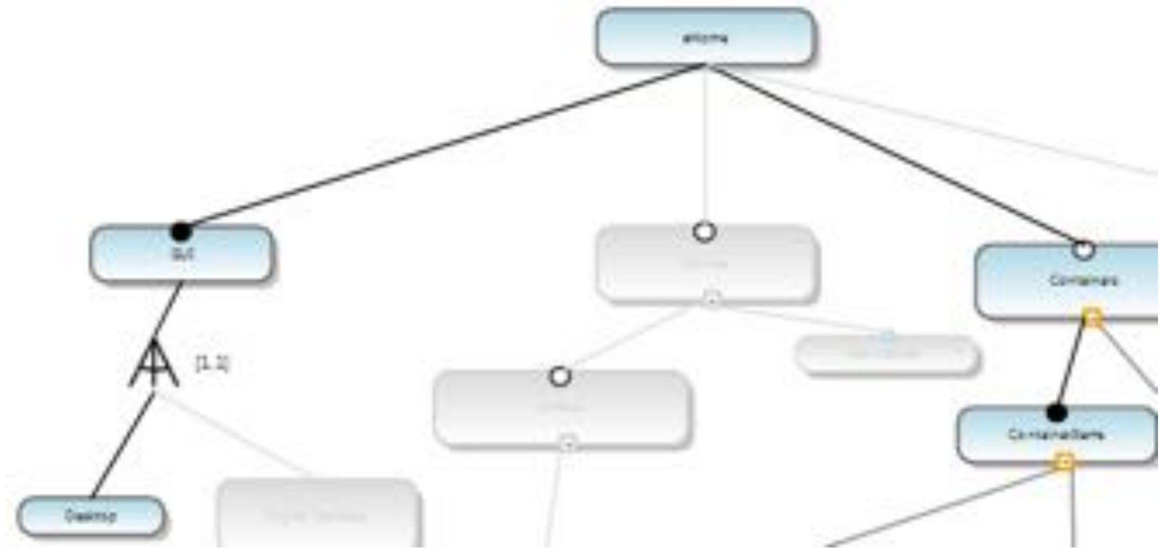
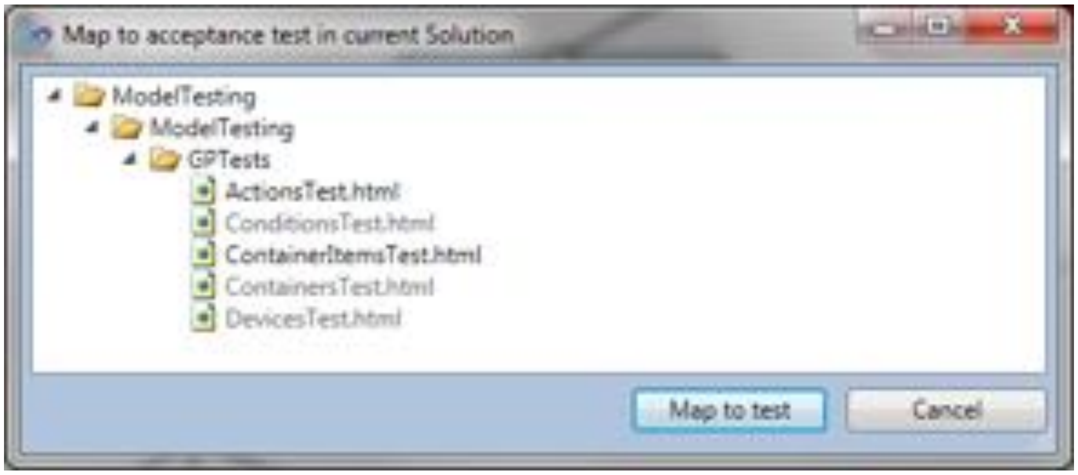


Figure 38 - The user can model features and their relationships

In our extension of the tool, the leaves of the feature tree can be mapped to EATs. When the user clicks on a leave node, a new dialogue pops up showing a list of all EATs found in the solution as shown in Figure 39. Tests that have already been mapped are shown in grey. The user then selects an EAT to accomplish the mapping.


Map to acceptance test in current Solution

- ModelTesting
  - ModelTesting
    - GPTests
      - ActionsTest.html
      - ConditionsTest.html
      - ContainerItemsTest.html
      - ContainersTest.html
      - DevicesTest.html

Figure 39 – The leaves of the feature tree can be mapped to EATs

Through the use of a Visual Studio extension that executes GreenPepper [GreenPepper2010] acceptance tests, the tool allows the user to run EATs directly from the feature model as shown in Figure 40. This action runs all EATs in the feature model regardless of their relationship to each other. This implies that mutually exclusive EATs can be run together but they cannot pass simultaneously. Nodes that have passing tests are coloured in green and those with failing tests are coloured in red as shown in Figure 41.

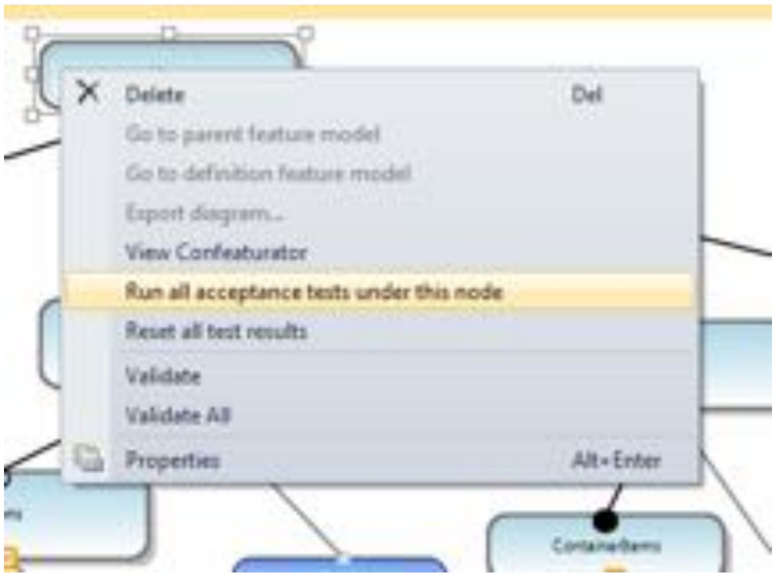


Figure 40 - The tool allows the user to run ATs directly from the variability model

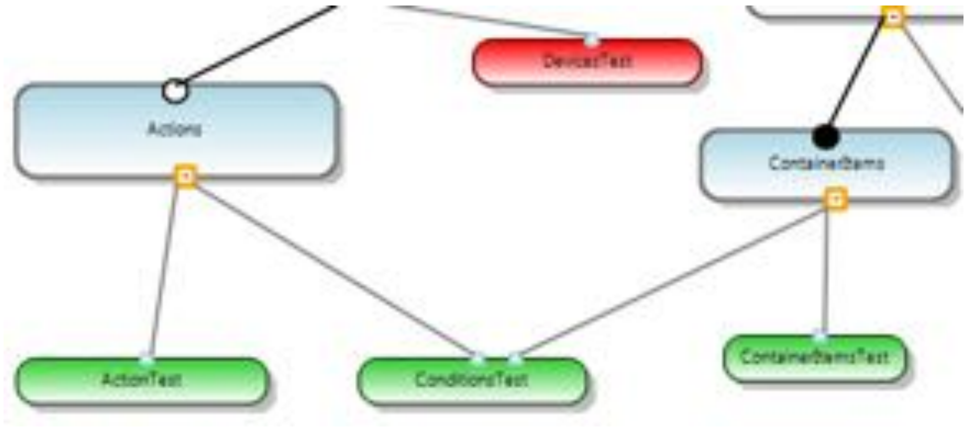


Figure 41 – Passing tests are coloured in green and failing tests are coloured in red

To run the tests for a specific instance of the product line, the user can use the configurator window to select the wanted features as shown in Figure 42.

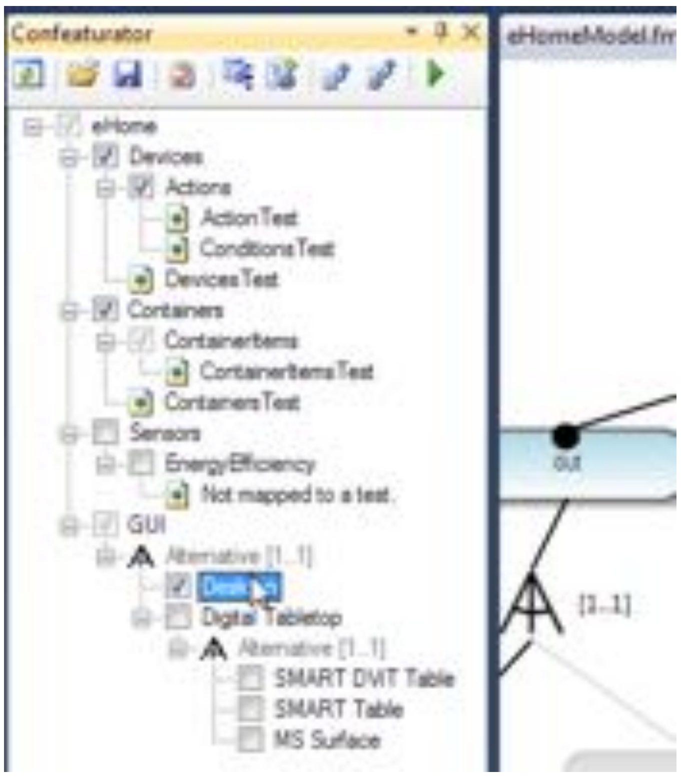


Figure 42 – The user can use the configurator window to select the wanted features

After feature selection (i.e. defining a new product instance), the tool checks the constraints to ensure the validity of the selected subset of features. This includes checking that all mandatory features have been selected. But as a proactive measure, the tool – by default – selects mandatory features and does not allow deselecting them. The tool then runs only those EATs that are relevant to the new instance. This is shown in Figure 43.

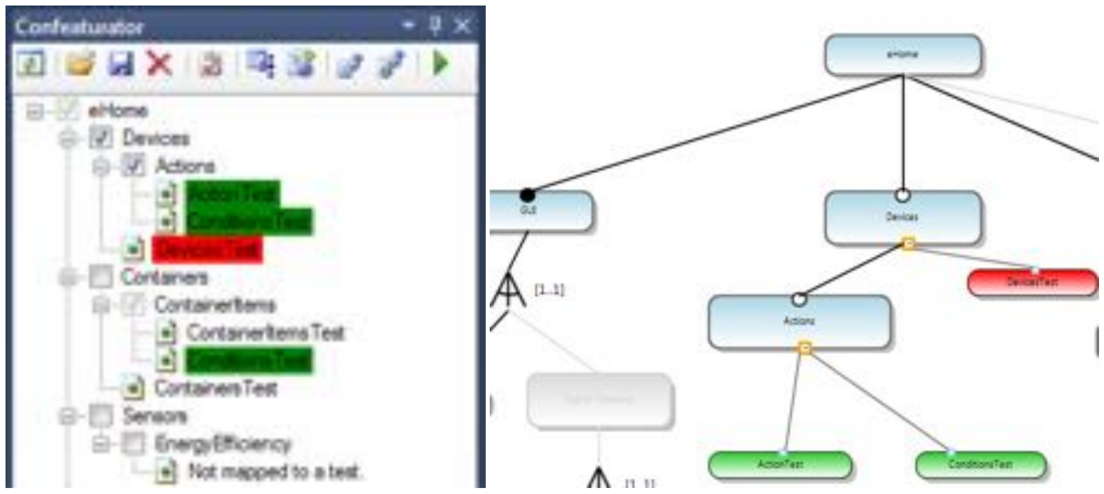


Figure 43 - The tool runs only those EATs that are relevant to a given instance

7.7 Evaluation

In this chapter, I showed how EATs can be used to link feature models to code artefacts. This section presents an evaluation of the proposed approach. I evaluate the approach in two different ways. First, I compare the approach with traditional requirement traceability approaches and other approaches that involve feature models. Then, I use the running example presented throughout this thesis to list the limitations of the approach. To avoid redundancy, I do not list the advantages of the approach as they have already been discussed in section 5. Using a running example for validation and evaluation purposes is a well-accepted technique in the SPL community [Tun2009, Parra2009, Cho2008].

7.7.1 Comparative Evaluation

For the purpose of this evaluation, I use the comparative evaluation framework provided by [Vartiainen2002] which consists of four steps, namely: selecting the objects to be compared, identifying the level of comparison, providing a common conceptual comprehension, and discussing the findings.

7.7.1.1 Object selection

In this evaluation, I conduct the comparison between my approach on one hand and two categories of approaches on the other hand. One category includes general requirement traceability approaches that are not necessarily specific to SPL contexts. The second category includes traditional approaches that use feature models to achieve traceability.

7.7.1.2 Level of comparison

The evaluation presented in this section lies on the extreme left of the continuum provided by Vartiainen [Vartiainen2002] as shown in Figure 44. That is, I am more interested in looking at the differences between approaches that are fundamentally similar in their purpose (i.e. achieving traceability).

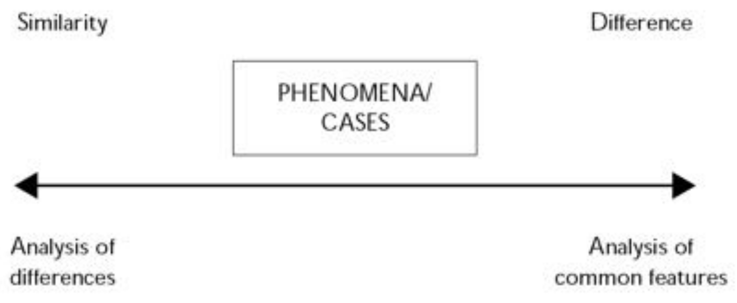


Figure 44 – Similarity vs. difference of the objects compared (obtained from [Vartiainen2002])

I use five units of comparison as evaluation criteria, namely: the number of links, the quality of links over time, system evolution, impact analysis, and program comprehension. These criteria are based on guidelines obtained from previous work in the literature such as [Riebisch2004] and [Antoniol2002] except that I apply them in a SPL context.

7.7.1.3 Conceptual comprehension

The goal of this step is to provide a common understanding of the concepts used in the comparative evaluation. The "number of links" criterion refers to how many links need to be maintained to achieve traceability for a given feature in the system. The "quality of links over time" criterion refers to how well the established traceability links cope with the inevitable changes in the system to stay relevant and up-to-date over time. The "system evolution" criterion describes how traceability links are affected with the evolution of a system such as adding or removing requirements. In the case of feature model approaches, I am more concerned with the evolution of variability such as adding or removing variation points and variants. The "impact analysis" criterion describes the depth of the information an approach can provide in regards to the impact a change in one part of the system has over other parts of the system. The "program comprehension" criterion is used to describe the ability of the developers to form a mental model of the variability definition as described in the feature tree as well as the realization of that variability at the code level.

7.7.1.4 Findings

The findings of the comparative evaluation conducted as per the framework described above are captured in Table 6. The evaluation in general is limited by the subjectivity arising from the criteria being considered. Also, the fact that the researcher cannot guarantee full impartiality in self-evaluation may introduce some bias in the findings.

Table 6 - Comparison between the different approaches of traceability

	Traditional Requirement Traceability	Traceability through Feature Models	Traceability through Feature Models and EAT
Number of links	Very large, because every requirement is linked to relevant design, code and test artefacts.	Somewhat large, because every feature is linked to relevant design, code and test artefacts.	Fairly small, because every feature is only linked to the EATs files specifying that feature. The links to code artefacts are embedded within the EATs themselves.
Quality of links over time	Links become broken or/and outdated without appropriate manual revisions and updates.	Links become broken or/and outdated without appropriate manual revisions and updates.	It is easier to keep links consistent and up-to-date because of the immediate feedback on broken or outdated links as there is a clear indication where/when revisions and updates are needed.

System evolution	If a feature is added, we need to provide the links to all the relevant requirements and other artefacts manually (because one requirement may be used in more than one feature). If a feature is removed, we need to check if the linked artefacts are still relevant to other requirements.	If a feature or a variation is added, we need to provide links to all the relevant requirements and other artefacts manually. If a feature or a variation is removed, we need to check if the linked artefacts are still relevant to other requirements. Also, there are no automatic checks for new hidden conflicts in the feature model.	The addition or removal of features and variations is supported by a safety net of EATs. Also, failing EATs may indicate newly introduced conflicts.
Impact analysis	Provides information on the artefacts that can be potentially impacted by a change. No details on the actual impact.	Provides information on the artefacts that can be potentially impacted by a change. No details on the actual impact.	Provides information on the artefacts that are actually impacted by a change, and provides immediate feedback on the actual impact of that change.

Program comprehension	Improved over systems with no traceability. But requires an effort for developers to link requirements with code tasks (reading requirement trace matrices is not simple). Also, given that variability is not modeled explicitly, handling each type of variation in code is not straightforward.	Reasonable, because requirements are conceptualized at a more comprehendible level of abstraction (i.e. features), and variability is modeled explicitly.	Good, because features are linked directly to code artefacts, and hence variants can be traced to code easily. Also, developers get instant feedback on changes to the code.

7.7.2 Running Example – Limitations

Having illustrated the advantages of our approach in comparison to other traditional approaches, there is a raft of issues that limit the practicality of the approach. For one, it is currently difficult to predict how scalable this approach is – especially when dealing with a large number of variation points and variants. This problem is inherited from the scalability issues associated with feature modeling in general [Chen2009b]. Furthermore, despite the fact that EATs provide an elegant way to specify functional requirements in software systems, they have not yet been widely used in specifying non-functional attributes such as presentation and portability as explained in the previous chapter*. For feature models that contain variability due to non-functional aspects, this approach may not be sufficient. Moreover, the most common practices involving EATs focus on code artefacts much more than other development artefacts. For organizations that consider design artefacts, for instance, to be essential, the adoption of this approach may result in these artefacts becoming rapidly outdated – mainly because from a developer's perspective there will be no need to maintain their details anymore. However, the organization can solve this problem by requiring that some EATs be used as placeholders to associate important information such as links to design documents, standards or data files [Park2008]. Another critical point that may be a real challenge in some organizations is the commitment and discipline needed to provide sufficient EAT coverage for all features in the system in a sustainable manner. Adopting test-driven development practices is one way to deal with this issue.

* other non-functional attributes like performance can be specified and executed as described in [Marchetto2010].

It is important to point out that contrary to the initial impression that this approach may lead to architectural drift, the approach may actually improve adherence to the architecture. This is because of the transparency and traceability between the model artefacts and the code artefacts, which provide the developers with a holistic and consistent understanding of the product line. This, however, is still an open issue to investigate.

7.8 Chapter Summary

This chapter discussed the use of EATs as a means to link feature models to code artefacts. Linking conceptual requirements in feature models to actual implementation artefacts provides for many advantages such as increased program comprehension, implementation completeness assessment, impact analysis, and reuse opportunities. The approach proposed in this chapter provides traceability links in a way that ensures consistency between the feature model and the code artefacts, enables the evolution of variability in the feature model, and supports the product derivation process (explained in detail in chapter nine). The valuable implications of these three characteristics on SPLs were illustrated in detail. The approach was compared to traditional approaches and a number of advantages and limitations were identified.

CHAPTER EIGHT: VARIABILITY REALIZATION[*]

8.1 Preamble

In the previous chapter, I showed how EATs could be used to achieve traceability links between the feature model and the code artefacts in a SPL context. This chapter elaborates on how variability can be realized (i.e. implemented) at the code artefact level to reflect variability in the feature model so that the relevant EATs pass. The work presented in this chapter is motivated by two reasons. The first is related to the minimal documentation or even the absence of it in an agile context. This triggers the quest for an approach to systemize the realization process that is different from traditional approaches which generally rely on requirement and design documents. The second reason is related to the reactive aspect of the SPL framework I propose in this dissertation. That is, as variability has not been proactively accounted for in the architecture, there is a need to investigate a bottom-up approach that allows injecting variability on-demand.

8.2 Research Instruments

In this chapter, I investigate how variability can be introduced at the code level in a systematic and reactive manner. The research question I address in this regard is the following:

RQ4. In an agile context, how can variation points and variants be realized at the code level in a reactive and systematic manner?

[*] This chapter is based on a published paper [Ghanam2010b]. Co-author permission is attached to Appendix B.

In the context of the framework I propose, this research question addresses Stage D: Variability realization as shown in Figure 45. That is, upon the demand to introduce a new variation point or variant in the system, a process will need to take place to change the system from a specific system (i.e. one that satisfies a specific set of requirements), to a generic system (i.e. one that satisfies more than one set of requirements) with configuration capabilities.

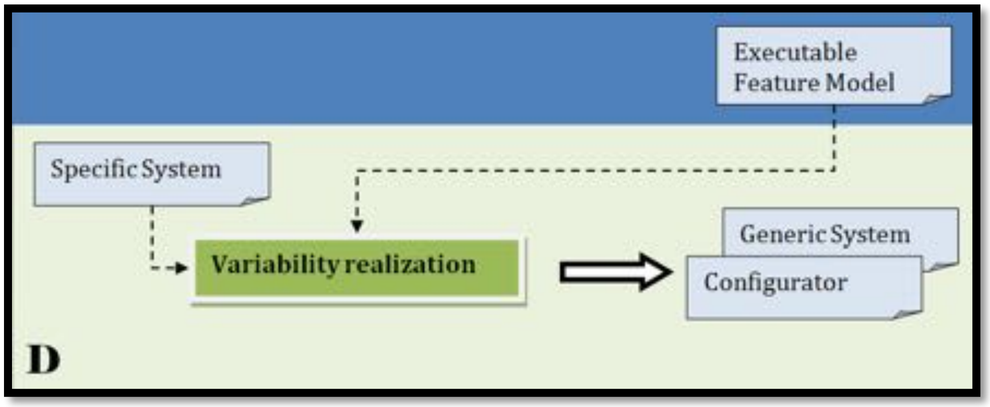


Figure 45 - This chapter addresses Stage D: Variability realization

The first research instrument I use in this chapter is a preliminary analysis to articulate the different variation types and the different ways these variations are usually handled. I then build my proposal upon this analysis to address reactive and systematic variability realization in agile contexts.

The tool support provided to automate the proposed approach is evaluated in two phases for feasibility and practicality. In the first phase, I conduct a proof-of-concept evaluation to study a mock-up system and iteratively refine the provided automation. In the second phase, I use a case study approach to examine the practicality of the automated approach on an independent third-party system. The design of the study includes a number of components, namely: the selection criteria of the case to be studied, the procedure followed to conduct the study, the assessment questions used to evaluate the results, and an analysis of the findings.

8.3 Preliminary Analysis

8.3.1 Variability Realization Techniques

In a SPL, variability in a given feature occurs in two different ways:

a. The feature requires various implementations for different customers (aka. alternatives). For example, the same "Secure Connection to Server" feature may need to be implemented using two different security protocols to satisfy the different needs of the customers. One way[*] to implement this is by using a factory pattern [Gamma1995]. This pattern provides a mechanism to create different concrete classes through a single factory.

b. The feature has optional extensions that are needed by some but not all products (aka. options). For example, the "Access Control" feature supports "Fingerprint Authentication" but only for those customers who want this kind of authentication. For options, I use a decorator pattern [Gamma1995] which allows extending the behaviour of an existing object at runtime.

When a SPL practice is not adopted, embracing variability is usually done by one of three techniques, namely:

[*] There are other techniques and patterns to realize variability for alternatives as well as options. Choosing the appropriate technique for each case is beyond the scope of this thesis. Our approach, however, should work fine with any code-based variability implementation technique.

a. Building the application from scratch. Some code can be reused in an ad-hoc manner, but this practice results in multiple repositories that need to be maintained and supported separately which is highly inefficient and error prone.
b. Clone-and-own techniques where the base code is copied and then customized to satisfy the new variation. This practice has the same problems as in (a).
c. Ad-hoc refactoring, where it is left up to the developer to refactor existing code to satisfy both the new as well as the existing variation. In this case, there is neither a systematic way to refactor the code nor a way to convey knowledge about the existence of variation points. This causes variability in the system to become too cumbersome and expensive to maintain, and may render the instantiation process vague.

On the other hand, SPL engineering deals with variability in a systematic manner through variability management practices. Traditional SPL approaches manage variability in a proactive manner in order to make the architecture capable of accommodating the different variation points and their variants.

8.3.2 Premises of the Proposed Approach

The fundamental premise of the approach I propose in this chapter is that variability in an agile context should be handled in a reactive manner. Being reactive means that unless requirements about variations in the system are available up-front, the agile organization should not proactively invest into architectural design to accommodate what might vary in the system. Rather, the normal course of development should take place to satisfy the current needs of the customers. Later on, should a need to introduce a variation point arise – whether during development or after delivery – agile teams should have the tools to embrace this variability.

Having said that, the approach addresses the problems found in ad-hoc refactoring by emphasizing the systematic aspect. This means that the variability realization process should not be left completely up to the developer's intuition as practiced in ad-hoc approaches. Rather, the developers should be guided by a certain procedure that ensures consistency in the implementation of different variation points and variants in the system, and that provides a systematic way to configure and instantiate products. The proposed approach utilizes test artefacts in the existing system to achieve the abovementioned objectives.

8.3.3 The Role of Test Artefacts

In agile approaches like Extreme Programming [Beck2004], automated tests are deemed essential. There usually exist two types of tests: unit tests (UT) and acceptance tests (AT). In the previous chapter, I discussed the role EATs can play in a SPL context. In this chapter, I focus on the use of UTs. UTs verify the correctness of the behaviour of an individual unit, or the interaction between units. In test-driven development [Beck2003], UTs are written before writing production code. UTs are automated to be executed frequently and help in refactoring the code.

In the proposed approach, using EATs in the feature model is a step that comes before implementation. That is, EATs serve as a high-level, customer facing representation of the needed variability. After that, UTs become relevant in three ways:

- UTs are used as a starting point to drive the variability realization process to implement the variability prescribed in the relevant EATs. This point will be discussed further in the upcoming sections.
- When a variation point is realized along with its variants, UTs ought to exhaust all the different variants, and therefore they are part of the variability realization process.
- UTs serve as a safety net to make sure the variability realization process did not cause any destructive side effects.

8.4 The Proposed Variability Realization Approach

8.4.1 Refactoring for Variability

To illustrate the proposed approach, I use a simple example. Say, within a smart home security system, we have an electronic lock feature on every port (door or window). The diagram in Figure 46 illustrates the current state of the system. The Lock class is tested by the *LockTest* class. Arrows show the call hierarchy. For instance, LockTest.*testSetPassword()* calls the public method *Lock.setPassword()*, which in turn calls the private method *Lock.isValidPassword(String)*.

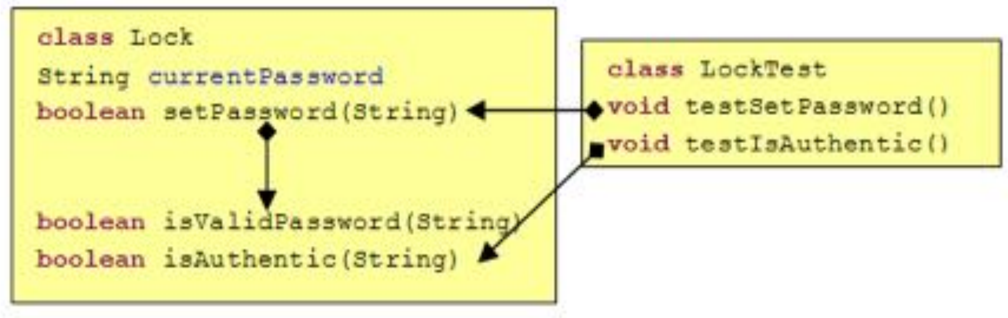


Figure 46 - Current state of the Lock feature.

Currently, the system allows the user to set a 4-character password for the locks. The password criteria are checked in the *Lock.isValidPassword()* method shown in Listing 1. Say we need to introduce a variation point to the lock feature to allow customers to choose the desired security level needed on the locks before they purchase the system. Variants include a 4-char lock (the original lock), an 8-char lock (a new alternative), or a non-trivial 8-char lock (another new alternative - characters cannot be all the same and cannot be consecutive).

```
class Lock {
      String currentPassword="";

      public boolean setPassword(String password) {
            if(isValidPassword(password)) {
                  this.currentPassword = password;
                  return true;
            }
            return false;
      }

      boolean isValidPassword(String password) {
            if(password.length()== 4) return true;
            return false;
      }


      public boolean isAuthentic(String password) {
            if(password == currentPassword) return true;
            return false;
      }
}
```

Listing 1 – Password criteria are checked in the *Lock.isValidPassword()* method

To implement this, I use a factory pattern as mentioned previously to reach the configuration shown in Figure 47. In this case, this pattern was chosen because I am dealing with a feature that provides a common implementation of two of its methods across all product instances (implemented as an abstract class), but the third method will vary according to the chosen alternative (implemented differently in the inheriting classes).

One can abstract the method that is responsible for password validation (i.e. *Lock.isValidPassword(String)*) and provide three different implementations for it. Refactoring the code to reach the configuration in Figure 47 has consequences. First of all, UTs need to be written to reflect the new changes. Also, we need to change all instantiations of the old Lock class to use the new factory instead. And before every instantiation of the Lock class, a specific implementation is to be selected. For this to work, an implementation selector class is needed to return the proper implementation.

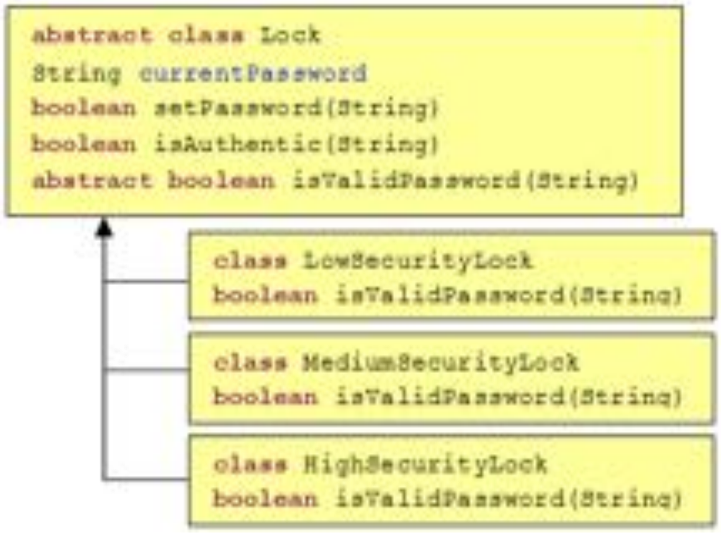

abstract class Lock
String currentPassword
boolean setPassword(String)
boolean isAuthentic(String)
abstract boolean isValidPassword(String)
class LowSecurityLock
boolean isValidPassword(String)
class MediumSecurityLock
boolean isValidPassword(String)
class HighSecurityLock
boolean isValidPassword(String)

Figure 47 - The new state of the Lock feature

To support the refactoring process and all its consequences, I leverage the readily available traceability between UTs and production code.

8.4.2 Formalization

Formalizing the variability realization process is important in order to automate it. Given that this realization process is test-driven, the process starts with an existing system that consists of a set of UT classes C_u. Each testing class c_{ui} has a set of testing methods.

$$C_u = \{c_{u1}, c_{u2}, c_{u3}, \dots\} \; where \; c_{ui} = \{m_{ui1}, m_{ui2}, m_{ui3}, \dots\}$$

Each testing method m_{uij} may instantiate or use a number of classes in the system – set C. Each class c_k consists of a set of methods.

$$C = \{c_1, c_2, c_3, \dots\} \; where \; c_k = \{m_{ck1}, m_{ck2}, m_{ck3}, \dots\}$$

Each method in c_k may instantiate or use a number of other classes in the system. Over the previous sets, I define a process to introduce a variation point to the system. The process consists of five functions as described in the following sections.

8.4.3 Variation Initialization Function

This function determines two attributes:

1. The UT of interest as a starting point $m_{uij} \in c_{ui}$. This feeds into the next function.
2. Variation details needed for code refactoring and generation. This includes providing a name for the new variation point, selecting one of two variation types: alternatives or options, and providing names for the wanted alternatives or options.

These two attributes should be determined by the developer. The developer chooses the UT that tests the scenario where variability needs to exist. In the example above, it is the

LockTest.testSetPassword() method shown in Listing 2[*], because this is mainly where the setting password part of the feature is tested.

```
public void testSetPassword() {
      Lock lock = new Lock();
      Assert.assertFalse(lock.setPassword(""));
      Assert.assertFalse(lock.setPassword("Hello"));
      Assert.assertTrue(lock.setPassword("Helo"));
}
```

Listing 2 – The UT that tests the scenario where variability needs to exist

The developer then decides whether the new variability is due to the need to provide alternate implementations or to add options to the feature at hand. In the example, I choose alternatives. The developer provides the names of the wanted variants. For example, “low”, “medium” and “high”. The first variant is assigned by default to the original implementation (i.e. before variability existed).

8.4.4 Call Hierarchy Function

This function determines the transitive closure of the UT in m_{uij} (the first attribute in the previous function). This includes all methods in the system that are invoked due to the invocation of m_{uij}. The transitive closure is calculated by searching for declarations of all methods that are called within the test method. This also includes searching in the hierarchies of the declaring types to look for overriding methods because with static code analysis it is not possible to know which method is invoked when a method is called using a base class reference. The obtained methods are added to a list of candidates to be considered for refactoring. Then, within each of the obtained methods, I search for

[*] This UT is for illustration only. It is understood that in real life best practices like one-assert per test should be observed.

declarations of methods being called and so on until no more methods are found. All of the obtained methods are also added to the list of candidates. The search ignores methods that cannot be candidates for refactoring such as methods that are external to the application (e.g third party library). Abstract methods (declared in interfaces and abstract classes) are not added to the list of candidates per se, but the methods that implement them are added [Salbinger2010].

In the example above, the call hierarchy of *LockTest.testSetPassword()* includes the methods: *Lock.setPassword(String)* and *Lock.isValidPassword(String)*. At this stage, developer's input is needed to identify where in the call hierarchy the variation point should exist. This determines the method that is causing the variation to happen. For example, because the variation point I need to inject is pertaining to the validation of the password criteria, I choose *Lock.isValidPassword(String)*.

8.4.5 Variability Trace Function

Given the method $\mathrm{m}_{\mathrm{ckn}}$ determined in the call hierarchy function, the variability trace function determines all the classes and methods that can potentially be affected by introducing the variation point.

In the example above, say there is a class *Port* that instantiates the *Lock* class. This instantiation needs to be updated to use the new factory.

8.4.6 Code Manipulation Function

This function performs the refactoring and the code generation needed to introduce the variation point and the variants based on the variation type determined in the call hierarchy function. Given the method $m_{ckn} \in c_k$ from the variability trace function, the following code manipulations take place:

- Refactoring c_k so that m_{ckn} is abstracted as a variation point as in Listing 3. The original class name and method name are left unchanged. The class and the selected method are made abstract, and the definition of the method is removed.

```
abstract class Lock {
        String currentPassword="";
        public boolean setPassword(String password) {
                if(isValidPassword(password)) {
                        this.currentPassword = password;
                        return true;
                }
                return false;
        }
        abstract boolean isValidPassword(String password);
        public boolean isAuthentic(String password) {
                if(password == currentPassword) return true;
                return false;
        }
}
```

Listing 3 – Abstracting the method where variability will be introduced

- Generating implementation templates for the variants as in Listing 4. The first generated implementation is the same as the original implementation before variability existed. The other generated implementations are empty templates representing the other variants needed. All implementations extend the abstracted class. The name of each implementation takes the form $VariantNameAbstractClassName$.

```
class LowLock extends Lock {
      boolean isValidPassword(String password) {
            if (password.length() == 4)
                  return true;
            return false;
      }
}
class MediumLock extends Lock {
      boolean isValidPassword(String password) {
            // TODO Auto-generated method stub
            return false;
      }
}
class HighLock extends Lock {
      boolean isValidPassword(String password) {
            // TODO Auto-generated method stub
            return false;
      }
}
```

Listing 4 – Generating implementation templates for the variants

- Declaring a new enumeration to explicate the variation point and its variants as in Listing 5. The enumerations take the form *VP_VariationPointName*, whereas variants are named in the form *V_VariantName*.

```
public enum VP_SECURITY_LEVEL { V_LOW, V_MEDIUM, V_HIGH }
```

Listing 5 – Declaring a new enumeration

- Creating or updating a configurator class as in Listing 6. The configurator serves two purposes. For one, it enables easy configuration and instantiation of products. Every variable in this class represents a variation point. The value assigned to each variable represents the variant of interest. Secondly, the configurator helps explicate the variability profile of the system so that it is visible to the stakeholders. The generated *VariantConfiguration* class has static fields that enable the selection of variants for all the variation points in the system. There should be only one *VariantConfiguration* class that encompasses all introduced variation points. Therefore, whenever a variation point is introduced, I first look for an existing *VariantConfiguration* class before creating a new one.

```
public class VariantConfiguration {
      public static VP_SECURITY_LEVEL securityLevel =
                                    VP_SECURITY_LEVEL.V_LOW;
}
```

Listing 6 – Creating or updating a configurator class

- Generating an implementation selector as in Listing 7. In this case, the selector is a factory class with the naming convention *AbstractClassNameFactory*. The class has a static method *createAbstractClassName()*. This method is responsible for instantiating the right subclass according to the configurations set in the *VariantConfigutation* class.

```
public class LockFactory {
      public static Lock createLock() {
            if (VariantConfiguration.securityLevel ==
VP_SECURITY_LEVEL.V_LOW) return new LowLock();
            if (VariantConfiguration.securityLevel ==
VP_SECURITY_LEVEL.V_MEDIUM) return new MediumLock();
            if (VariantConfiguration.securityLevel ==
VP_SECURITY_LEVEL.V_HIGH)     return new HighLock();
            else return null;
      }
}
```

Listing 7 – Generating a factory to select variants

- Updating affected code segments found in the variability trace function to use the new factory as in Listing 8.

```
Lock lock = LockFactory.createLock();
```

Listing 8 – Updating affected code segments

8.4.7 Test Update function

This function updates affected UTs and generates UTs for the new variants. This not only makes sure the new changes did not have a destructive effect on the original system, but also encourages test-driven development because it generates failing tests for developers to write before writing the logic for the new variants.

For each implementation, a UT is generated under the same test class where the method m_{uij} has been selected as the starting point. Test cases adhere to the following naming convention *OriginalTestClassName.originalTestMethodName_VariantName()*. In the example, the *LockTest.testSetPassword()* method is refactored to *LockTest.testSetPassword_Low()* as a test for the first (original) variant. Two more tests are added to test the other two variants. In each test, the first statement selects the variant to be tested.

Then what follows is the body of the original test method m_{uij} copied as is in all generated test cases (the only change in the body is the change made by the code manipulation function to update how objects are instantiated). This provides a template for the developer to change as required. The test case that tests the original implementation (i.e. before variability existed) should pass without any issues. However, the newly generated tests are initially forced to fail as a reminder for the developers to edit the test and its corresponding production code. The effect of the test update function is shown in Listing 9.

```
@Test
public void testSetPassword_Low() {
	VariantConfiguration.securityLevel = VP_SECURITY_LEVEL.V_LOW;
	Lock lock = LockFactory.createLock();
	Assert.assertFalse(lock.setPassword(""));
	Assert.assertFalse(lock.setPassword("Hello"));
	Assert.assertTrue(lock.setPassword("Helo"));
}
@Test
public void testSetPassword_Medium() {
	// TODO Auto-generated method stub
	VariantConfiguration.securityLevel =
VP_SECURITY_LEVEL.V_MEDIUM;
	Lock lock = LockFactory.createLock();
	Assert.assertFalse(lock.setPassword(""));
	Assert.assertFalse(lock.setPassword("Hello"));
	Assert.assertTrue(lock.setPassword("Helo"));
	org.junit.Assert.fail();
}
@Test
public void testSetPassword_High() {
	// TODO Auto-generated method stub
	VariantConfiguration.securityLevel = VP_SECURITY_LEVEL.V_HIGH;
	Lock lock = LockFactory.createLock();
	Assert.assertFalse(lock.setPassword(""));
	Assert.assertFalse(lock.setPassword("Hello"));
	Assert.assertTrue(lock.setPassword("Helo"));
	org.junit.Assert.fail();
}
```

Listing 9 – Generating UTs for alternatives

In the case of *options*, I generate tests for all combinations of options. An example is shown in Listing 10. In this example, the original test method used as a starting point was *testAlarmOff()*. A new variation point *ALARM_ACTION_OPTION* was introduced. The two options that were added were *NotifyPolice* and *CloseAllPorts*. Therefore, four different combinations were produced, namely:

1) The default implementation without any extra options.
2) The default implementation with the *NotifyPolice* option.
3) The default implementation with the *CloseAllPorts* option.

4) The default implementation with the *NotifyPolice* option as well as the *CloseAllPorts* option. The | operator is used to select multiple options for a given variation point.

In the current state of the implementation, there are no checks for invalid combinations if there were any constraints on variant selection.

```
@Test
public void testAlarmOff_Default(){
  OptionConfiguration.ALARM_ACTION =
                ALARM_ACTION_OPTION_CONSTANTS.DEFAULT;
  BurglaryDetector burgDetector =
                BurglaryDetectorFactory.createBurglaryDetector();
  String returnValue = burgDetector.setAlarmOff();
  Assert.assertEquals("Burglary Detected", returnValue);
 }

@Test
public void testAlarmOff_NotifyPolice() {
  // TODO Auto-generated method stub
  OptionConfiguration.ALARM_ACTION =
                ALARM_ACTION_OPTION_CONSTANTS.NOTIFY_POLICE;
  BurglaryDetector burgDetector =
                BurglaryDetectorFactory.createBurglaryDetector();
  String returnValue = burgDetector.setAlarmOff();
  Assert.assertEquals("Burglary Detected", returnValue);
  org.junit.Assert.fail();
 }

@Test
public void testAlarmOff_NotifyPolice_CloseAllPorts() {
  // TODO Auto-generated method stub
  OptionConfiguration.ALARM_ACTION =
                ALARM_ACTION_OPTION_CONSTANTS.NOTIFY_POLICE |
                ALARM_ACTION_OPTION_CONSTANTS.CLOSE_ALL_PORTS;
  BurglaryDetector burgDetector =
                BurglaryDetectorFactory.createBurglaryDetector();
  String returnValue = burgDetector.setAlarmOff();
  Assert.assertEquals("Burglary Detected", returnValue);
  org.junit.Assert.fail();
 }

@Test
public void testAlarmOff_CloseAllPorts() {
  // TODO Auto-generated method stub
  OptionConfiguration.ALARM_ACTION =
                ALARM_ACTION_OPTION_CONSTANTS.CLOSE_ALL_PORTS;
  BurglaryDetector burgDetector =
                BurglaryDetectorFactory.createBurglaryDetector();
  String returnValue = burgDetector.setAlarmOff();
  Assert.assertEquals("Burglary Detected", returnValue);
  org.junit.Assert.fail();
 }
```

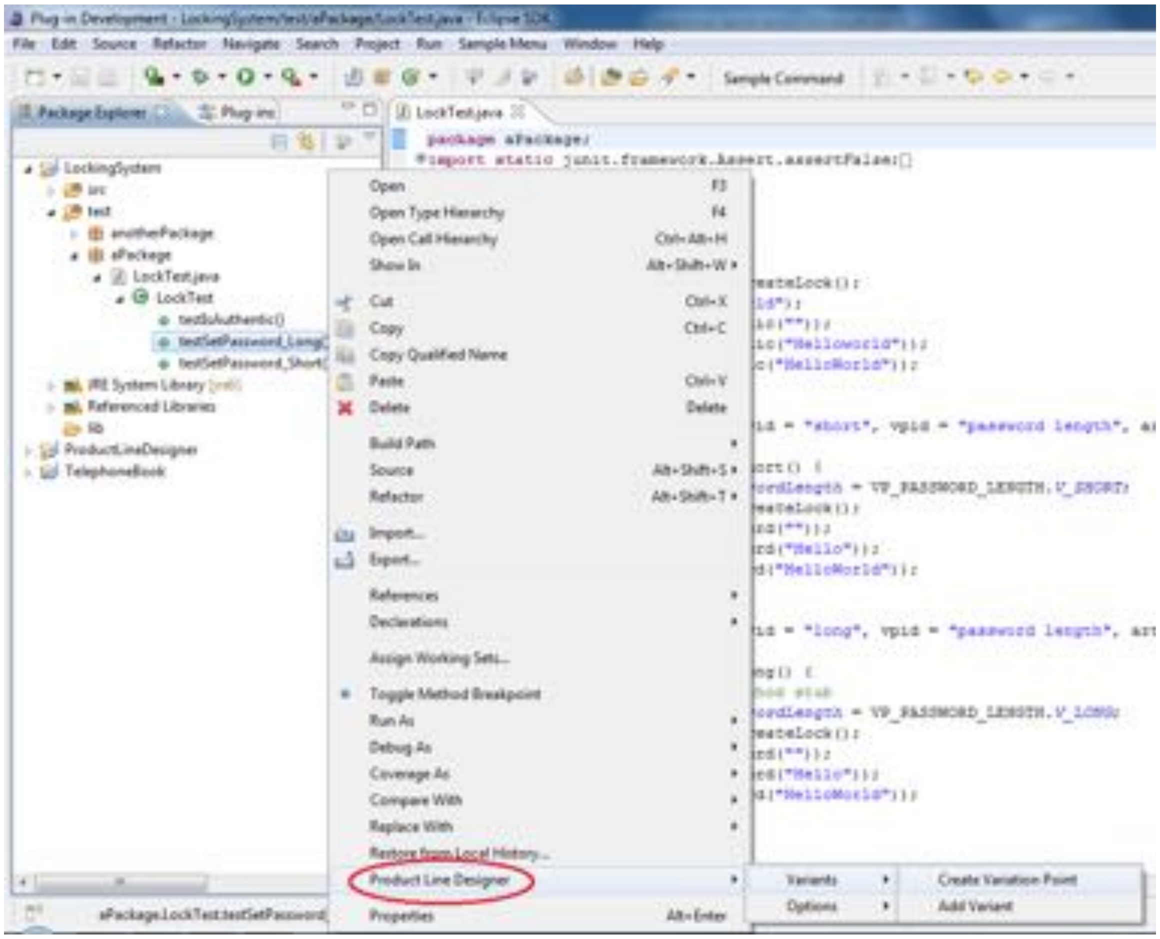
Listing 10 – Generating UTs for options

8.5 Automation

The abovementioned process to introduce a variation point in a system entails nontrivial refactoring and code generation steps. We built an eclipse plug-in that automates the whole process assisted by input from the developer [Salbinger2010][*]. The tool is open source and is available online [PLD2011]. When a variation point is to be introduced into the system, the following sequence of steps take place:

1. The developer navigates to the UT corresponding to the aspect of the feature where the variation point should be added as shown in Figure 48.
2. The developer chooses to add a variation point of a certain type as shown in Figure 48.
3. The tool finds the transitive closure of the chosen test method. The developer selects the method that is considered the source of variation as shown in Figure 49.
4. The developer provides a name for the new variation point. Then the developer adds new variants as shown in Figure 50. The tool assumes that the first variant represents the original implementation (i.e. before variability existed).

[*] The implementation of the tool was done collaboratively with Steffen Salbinger [Salbinger2010].



```
package aPackage;
import static junit.framework.Assert.assertFalse;
```

Figure 48 – Choosing the unit test and the type of variation

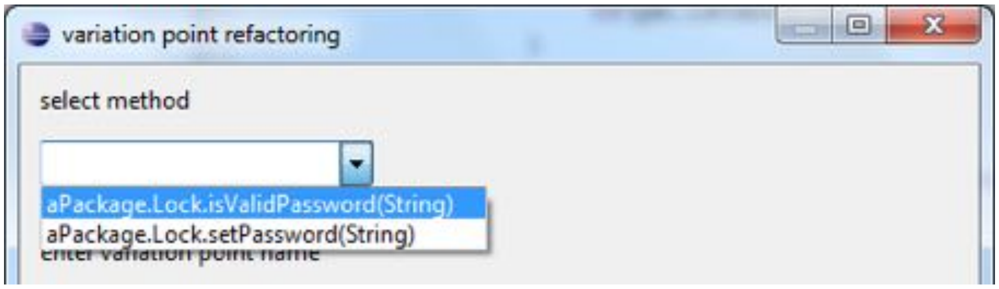

variation point refactoring
select method
aPackage.Lock.isValidPassword(String)
aPackage.Lock.setPassword(String)
enter variation point name

Figure 49 – Selecting the source of variation from the transitive closure results

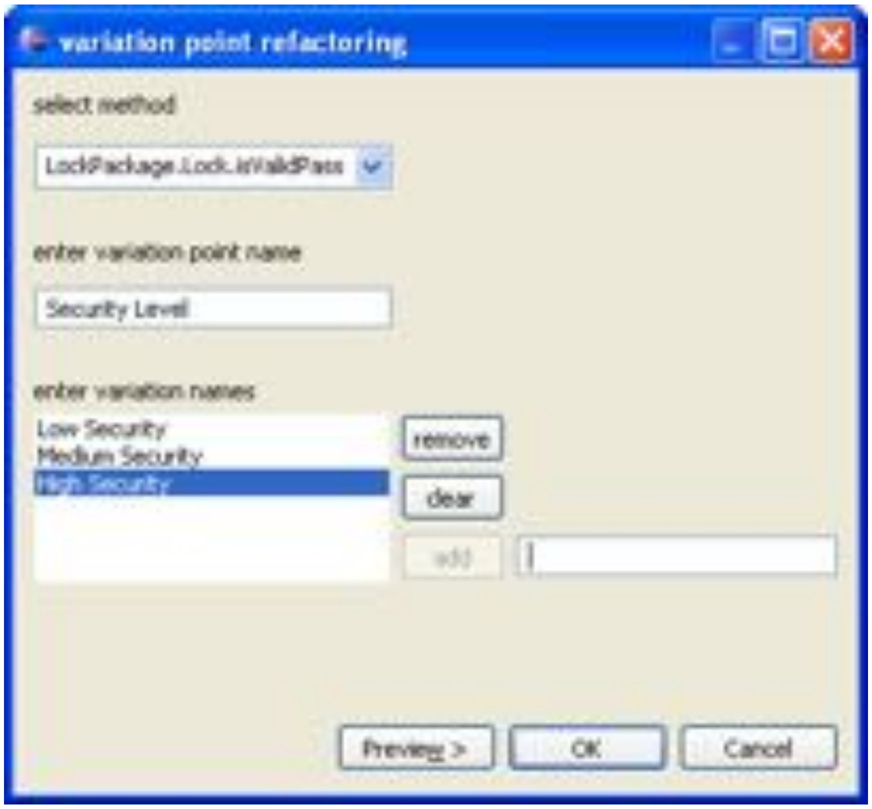
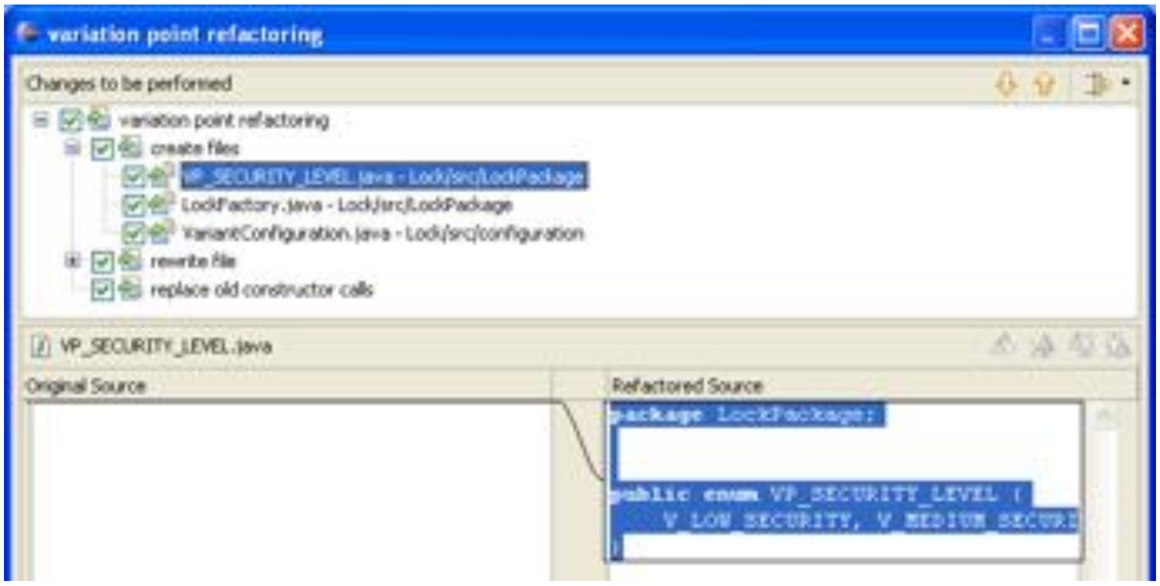


Figure 50 - Expected input from the developer

5. The tool performs the proper refactoring and code generation as described in the previous section. Namely, the tool will:
    - Abstract out the source of variation.
    - Provide an implementation class for each variant.
    - Provide a factory to select the proper implementation.
    - Define an enumeration to enable easy configuration of the system at instantiation time. All variation points will be packaged nicely in a configuration file to convey knowledge about variability in the system and the decisions that need to be made.

As shown in Figure 51, before any refactoring takes place, the developer is made aware of all the tentative changes.


variation point refactoring
Changes to be performed
variation point refactoring
create files
VP_SECURITY_LEVEL.java - Lock/src/LockPackage
LockFactory.java - Lock/src/LockPackage
VariantConfiguration.java - Lock/src/configuration
rewrite file
replace old constructor calls
VP_SECURITY_LEVEL.java
Original Source
Refactored Source

```
package LockPackage;

public enum VP_SECURITY_LEVEL {
    V_LOW_SECURITY, V_MEDIUM_SECURI
}
```

Figure 51 - The developer is made aware of the refactoring steps and the potential changes to the code

6. The tool updates all references to the old class to the correct object instantiation technique.
7. The tool provides UTs for every variant. In case of options, UTs will be provided to test all possible combinations of extensions.

8.6 Evaluation

So far in this chapter, I described a reactive and systematic approach to realize variability in software systems at the code level. To automate the approach, an eclipse plug-in has been built. This section provides an evaluation of the plug-in over two phases. The first phase is a feasibility evaluation, and the second is a practicality evaluation.

8.6.1 Feasibility Evaluation – Proof-of-Concept

In this phase, the goal is to determine whether it is feasible to automate the proposed approach to inject variability into a system through systematic refactoring given all the complications of object-oriented design.

8.6.1.1 Procedure

In this part of the evaluation, I study the capabilities and limitations of the plug-in. Initially problems were diagnosed by applying the tool to different code snippets that vary in complexity. As a starting point, I used the simple mock-up system presented earlier in this chapter. Then, I extended the system in a number of ways including:

- Adding more classes, or adding more UTs.
- Using different types of constructors like parameterless constructors, parameterized constructors and super constructors.
- Applying object-oriented techniques like subclasses and interfaces.

8.6.1.2 Assessment

After each extension to the mock up system, I tried to apply a number of operations, namely:

- Adding variation points as alternatives.
- Adding variation points as options.
- Adding variants to existing variation points.

The assessment questions I used to evaluate the outcome were as follows:

Q1. Was the plug-in able to handle both types of variations (i.e. alternatives and options)?

Q2. Did the refactored code for each operation match the expected outcome in different object-oriented contexts including different types of constructors, abstract classes, subclasses, interfaces, and existing design patterns?

Q3. Did the tool generate new code as prescribed in the approach in different object-oriented contexts including different types of constructors, abstract classes, subclasses, interfaces, and existing design patterns?

Q4. Did the process have any negative side effects such as compilation errors or test failures?

Q5. How did the tool perform throughout the whole process?

8.6.1.3 Findings

The feasibility evaluation enabled the iterative refinement of the automation plug-in before it was actually used on a real system in the second phase of the evaluation. Over a number of iterations, some issues were found such as dealing with instantiations where parameterized constructors were used (Q2), resolving call hierarchies (Q1) for some variations, refactoring instantiations where there already is a design pattern in use (Q2), and dealing with classes that already are abstract (Q3). These issues were then resolved in the following iterations. Another issue that came up in the next iterations, as far as inheritance is concerned (Q2 & Q3), was that the static code analysis we used did not provide a means to know which specific method was invoked when a method is called using a base class reference. This problem was solved by searching the hierarchies of the declaring types downwards for overriding methods.

After resolving the issues above, the tool showed to be reliable in making changes without any negative side effects (Q4). Performance wise, no delays have been observed in any of the steps throughout the process (Q5).

In summary, the results of the proof-of-concept evaluation showed that the approach was feasible with the support of the automation tool after refinements had been made to handle systematic refactoring in certain object-oriented designs as explained above.

8.6.2 Practicality Evaluation – Case Study

The practicality check aims to evaluate how practical the proposed approach is in terms of introducing variability to a real system that was not originally developed with variability in mind. For this part of the evaluation, I used a real open source system available at SourceForge as a case study.

8.6.2.1 Case selection

The following protocol was used to select the case to be studied. First, a list of all projects in sourceforge.net was obtained with "Java" as a programming language filter. This filter was applied because our plug-in is written for Eclipse, and it only supports refactoring in Java. The projects were sorted by the number of downloads in a descending order to make it easier to select a system that is actually in use (to avoid experimental projects). Then, the projects were examined one by one to identify those projects that already had UTs and were easy to import to Eclipse Ganymede 3.4.2 (the version the plug-in has been written for). The availability of UTs was important because I wanted to avoid the bias coming from producing my own UTs.

As a result, I selected a system called Buddi [Buddi2011]. Buddi is a financial management program targeted for users with little or no financial background. It allows users to set up accounts, create spending and income categories, record transactions, and check spending habits. Up till the end of April 2011, Buddi has been downloaded 911,042 times. Buddi has about 24,775 lines of code and 227 classes. This size puts Buddi in the medium-scale category which was a reasonable, yet not intended, size for evaluation purposes.

8.6.2.2 Procedure

Buddi was originally intended for personal use. In order to introduce variability to Buddi, I asked the question: what do I have to do to make Buddi suitable for small businesses? As a result, I developed the following scenarios:

1) Buddi usually accepts credit payments (transactions). To use this feature to support store-credit, one needs to provide the possibility of assessing risks. This yields the first variation point *Risk Assessment* with the variants (alternatives): *None*, *Flexible* and *Strict*. The *None* variant represents the original implementation of Buddi that did not have the notion of risk assessment. The *Flexible* variant puts some restrictions on the credited amount, and checks balance status for the past few months. The *Strict* variant adds more restrictions on any store-credit such as considerations of when the account was opened.
2) Buddi updates the balance of a customer after each transaction. For fraud-detection, one might need to take some security measures. This yields the second variation point *Security Measures* with the variants (options): *Log Balance Updates*, and *Send SMS to Customer*.

For assessment, I used the same five assessment questions as in the previous phase of evaluation.

8.6.2.3 Findings

The automation plug-in was used to refactor the code systematically as per the scenarios mentioned above. The plug-in was in fact able to handle both types of variations (i.e. alternatives and options) without any human interference – except for input from the developer wherever it was prescribed in the approach (Q1). The output of using the approach for each variation was as expected. That is, relevant code was refactored (Q2) and new code was generated as prescribed (Q3). The process did not create any compilation errors or cause a test to fail (Q4).

However, when observing the performance of the tool, I noticed a noticeable delay of about 9 seconds on average during the execution of the call hierarchy function (Q5). This was not an issue in the feasibility evaluation due to the small-scale of the mock up system. Although this delay might not limit the usefulness of the tool, it might, for larger projects with millions of lines of codes, limit its practicality.

8.6.3 Limitations & Threats to Validity

Beyond the five assessment questions discussed in the previous section, a number of limitations were found during the evaluation. For example, the transitive closure for some tests was too large to navigate. This warrants research into better visualizations of large call hierarchies. Also, the current implementation of the plug-in does not support combining variation points in a hierarchical manner. For example, currently the tool does not support scenarios where a variation point of type alternatives need to be defined, and one or more of these alternatives have a number of options to select from. This issue can be resolved with more sophisticated code analysis and refactoring. Moreover, the tool does not support dependencies between variation points and variants. For example, multiplicity constraints between alternatives and options are not taken into account.

The evaluation of the approach with the automation support faced some validity threats. The cases that were developed to be used in the feasibility evaluation might have been subject to internal bias. Also, although attempts were made to cover a wide range of object-oriented configurations, it is understood that the evaluation for feasibility did not exhaust all possible cases, which makes it hard to generalize that the approach is completely feasible for any project and under any circumstances.

Moreover, the evaluation considered two projects of small and medium sizes, which may cause a threat to the generalizability of the findings on other projects – especially large-scale ones. Also, the scenarios used in the practicality evaluation were developed by the investigator which might have introduced confirmation bias.

8.7 Chapter Summary

In this chapter, I presented a reactive and systematic approach to introduce variability to existing systems on-demand. I proposed a test-driven, bottom-up approach to introduce variability in software systems by means of refactoring. Systematic refactoring is used in order to inject variation points and variants in the system, whenever needed. I also presented an Eclipse plug-in to automate this process. An evaluation of the feasibility and practicality of the approach was discussed. The approach, supported by the plug-in was found to be feasible and practical, but suffered some limitations.

CHAPTER NINE: PRODUCT DERIVATION*

9.1 Preamble

The previous chapters showed how variability can be elicited, analyzed, modeled, and realized in a lightweight and reactive manner. This chapter sheds more light on the process of deriving individual products in a given SPL. This process is also known as product instantiation. Deriving different products from a single code base is an essential aspect of SPL engineering. Automating this process is key to its efficiency – especially when mass customization is expected. The reason I look at this issue within the scope of my dissertation is that the derivation process varies according to the variability management approach being used. That is, in traditional variability management, derivation happens during the application engineering phase as a later step in a sequential process that starts with domain engineering. Also, the derivation process traditionally relies on the documents and models produced during the domain engineering phase. However, in the framework proposed in this dissertation, product derivation can happen any time during the development process, and there is considerably less emphasis on documents and models. This chapter provides alternate ways to derive products using the artefacts produced in the previous steps.

9.2 Research Instruments

In this chapter, I look at how products can be derived from a code base given that variability has been realized and modeled as per the proposed approaches discussed previously. The research question I address is as follows:

---

* This chapter is based on published paper s [Ghanam2009] and [Ghanam2010c]. Co-author permission is attached to Appendix B.

RQ5. How can the extended feature model support the derivation process of individual products from a common SPL base?

In the context of the framework I propose, this research question addresses Stage E: Product derivation as shown in Figure 52. The derivation stage is responsible for producing different instances of the system given as direct input the generic system, the configuration engine (aka. configurator), and the specific configurations needed by a given customer. These configurations can be projected on the executable feature model which serves as indirect input that is typically used at a higher business level to select features.

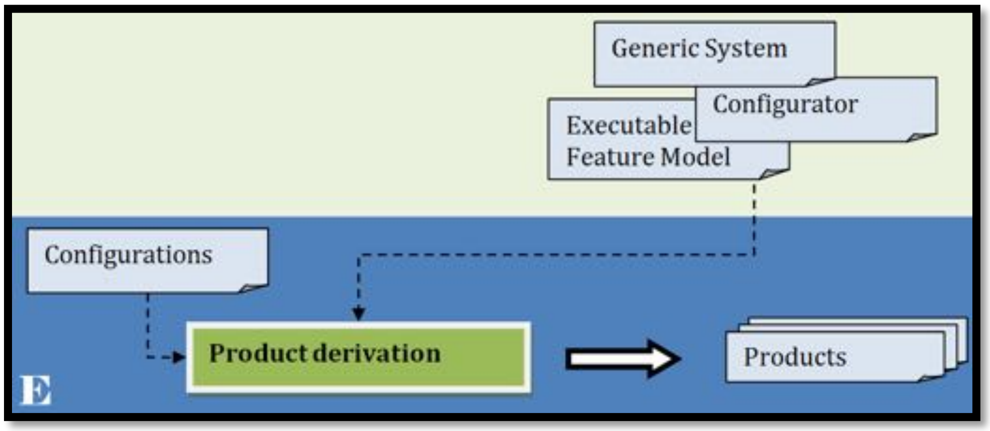


Figure 52 - This chapter addresses Stage E: Product derivation

The evaluation of this step is done by selecting a specific implementation technique and using it as proposed in the approach to derive products from a core system built in-house. An automation tool has been built to be used as an instrument in this evaluation.

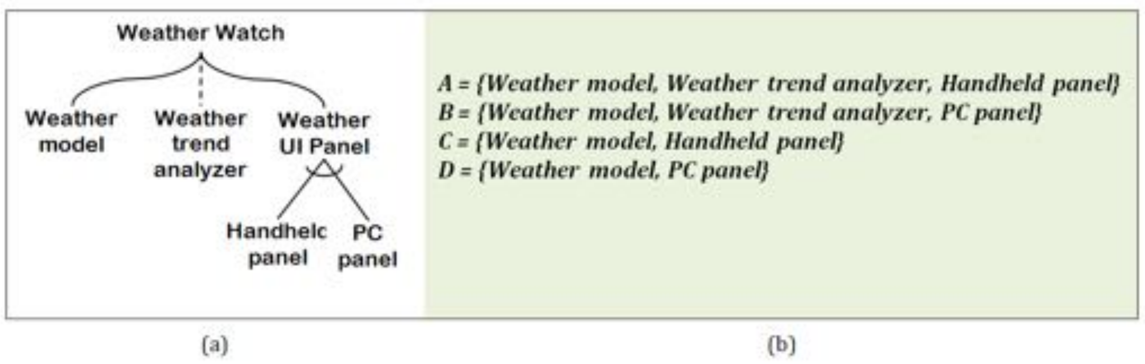
9.3 Preliminary Analysis

9.3.1 Product Derivation Techniques

In a SPL, the product derivation process requires at least the following components:

1. A generic base system S that includes all the reusable components that can be used to compose different products.
2. A set of configurations C that reflects the desired customization.
3. A configurator E that, given S and C, allows the binding of the variation points and variants as described in the variability profile.

In the example shown in Figure 53, the "Weather Watch" module would be the generic base system S. Say we need to produce the instance A (Figure 53b) that supports the weather trend analyzer on a handled device.



A = {Weather model, Weather trend analyzer, Handheld panel}
B = {Weather model, Weather trend analyzer, PC panel}
C = {Weather model, Handheld panel}
D = {Weather model, PC panel}

Figure 53 - Product derivation from a generic base system

The wants captured in the instance A constitute a configuration set C. The set is then used by the configurator E to bind the variation points to the variants. A sample configurator is provided in Listing 6.

```java
public class Configurator {
      public static VP_TREND_ANALYZER trendAnalyzer =
                                    VP_TREND_ANALYZER.V_EXISTS;
      public static VP_UI_PANEL uiPanel =
                                    VP_UI_PANEL.V_HANDHELD;

}
```

Listing 11 – A sample configurator containing multiple variation points

Once the variation points are bound to certain variants, the derivation process starts using one of two techniques:

1. **Instantiation:** in this approach, there is only a single system for the whole product line that behaves differently based on the variants chosen. There is no separate engine to control the derivation process. The variables in the configurator are used directly by creational classes in the system as decision variables in conditional blocks to choose specific implementations of the classes. Compiling the code and building a release is sufficient to instantiate a customized product. That is, in this case, the full code base is compiled and delivered as the product instance and the actual instantiation is done at run time based on the bindings specified in the Configurator class.
2. **Extraction:** in this approach, variability is not managed at run time. Rather, a sub-system is derived from a core system by extracting the reusable assets needed for a given configurations. This process requires a separate engine to control the derivation process so that the relevant assets are extracted to compose a sub-system which then is compiled and built to produce a customized product.

9.3.2 Direct vs. Indirect Configuration

Direct configuration happens when the configurator is used without any intermediary steps. That is, the stakeholders can alter the configurator by directly assigning variants to the different variation points. This level of customization is suitable for tech-savvy stakeholders who have a basic understanding of the programming language and conventions being used. In the example shown in Listing 6, Java is used as a programming language and a number of naming conventions are used.

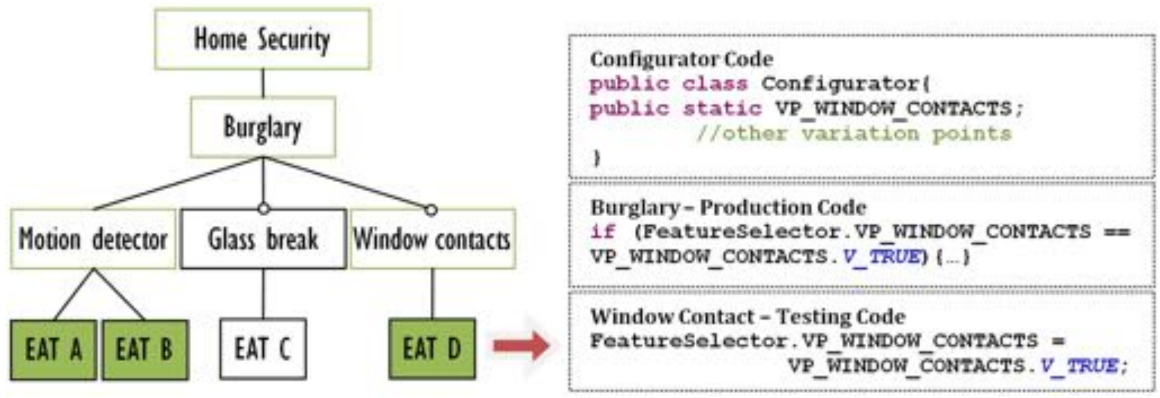
On the other hand, indirect configuration occurs when a high-level model is used as a facade for the configurator in a way that is accessible to non-technical stakeholders (such as customers). In a SPL context, this facade is usually a feature model. Feature models are used to select features and variants that constitute a product instance. The selection process should take into consideration the constraints and dependencies between features and variants, as conveyed in the feature model. Nowadays, tool support is available to make this process easier, faster and less error-prone.

9.4 Derivation in the Proposed Framework

In this section, I show how the proposed framework supports the derivation techniques discussed above.

9.4.1 Product Instantiation

During the derivation process, certain parameters (e.g. compiler directives, configuration classes) need to be set in order to select the desired configurations for the product instance at hand. I use the extended feature model (as described in chapter seven) to provide support for this process. EATs can be used to automatically set up the configuration parameters. This is possible because for an EAT to pass (independently of other EATs), it needs to set the correct parameter before it can execute the production code. When the selection process of features from the feature model is finished, all the EATs that are relevant to the current selection are run. Given that all EATs have passed for the current instance, this means that all parameters in the system have been set properly, and the system is now ready to produce the right instance (Figure 54). Another role of EATs in this context can be described as "*configuration by example*." That is, EATs provide a good starting point for the developers to learn how to configure a certain feature.



Configurator Code
```
public class Configurator{
public static VP_WINDOW_CONTACTS;
        //other variation points
}
```

Burglary – Production Code
```
if (FeatureSelector.VP_WINDOW_CONTACTS ==
VP_WINDOW_CONTACTS.V_TRUE){...}
```

Window Contact – Testing Code
```
FeatureSelector.VP_WINDOW_CONTACTS =
                    VP_WINDOW_CONTACTS.V_TRUE;
```

Figure 54 - Using EATs to select configurations

9.4.2 Product Extraction

In some derivation techniques, a subset of code artefacts are extracted from a core system based on the features selected using the feature model. A core system is one that continuously accumulates assets produced towards the satisfaction of previous customer requests. It is from the core system that family members are produced as instances in the product line.

In this section, I show how the extended feature model as proposed in the framework can play an important role in supporting this process. After the selection process of features in the feature model, all the EATs that are relevant to the current selection are run as shown in Figure 55. Static code analysis can provide details on which code artefacts are needed to produce the desired instance by computing the transitive closure of all calls in the fixture classes used in the EATs of the instance.

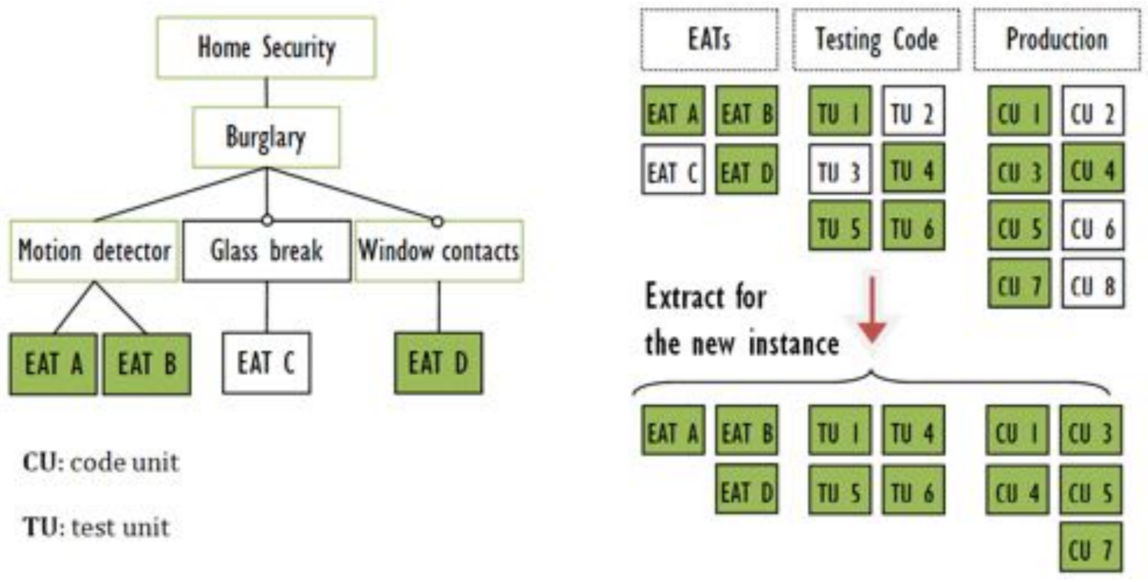


Figure 55 - Using EATs coverage reports to extract artefacts

The discussion to follow assumes that a core system is available and is represented through a library of EATs. The derivation process requires a number of steps, namely:

1. **Select EATs:** upon a new request of the system, the customer is provided with EATs that embody the different features currently available in the core system. Customers are to select only those EATs that match the scenarios they are looking for (highlighted in green in Figure 55). The outcome of this step is a subset of EATs.

2. **Execute ATs:** the selected subset of EATs is run against the core system; and a test coverage report is obtained using a test coverage tool. The coverage report

provides information about what code units or fragments were used to execute the given subset of EATs. This includes modules, namespaces, classes, methods, and files in both the testing code (fixture code) and the tested code (production code).

3. **Extract code:** based on the coverage information provided in step 2, relevant code units and fragments will be extracted from the core system. Any fragments that are not needed in the current instance are eliminated. This is the most complex and crucial step and will need special tool support. The outcomes of this step are two, namely: a subsystem that represents a variant of the core system, and a new test suite that possesses the fixture code needed to provide test coverage for the new system.
4. **Verify and build:** in this step, the newly derived system is compiled and built to make sure the extraction step did not produce any flaws in the code or the references. Then, by utilizing the test suite extracted in the previous step, the selected subset of EATs (from step 1) is run against the new system to verify the satisfaction of acceptance criteria within the new variant.

9.5 Evaluation

9.5.1 Goal and Scope

In the previous section, I showed how the proposed extended feature model can be used to derive products. This section aims at evaluating the strengths and weakness of the proposed derivation approach. Specifically, the evaluation focuses on the second derivation technique which relies on code extraction based on selected EATs. The reason behind this focus is that the extraction technique is significantly more complex and less straightforward than the instantiation technique. A proof-of-concept tool has been developed to aid in this evaluation. The evaluation addresses the following questions:

Q1. Can EAT artefacts be used as feature descriptors for the extraction process?

Q2. Does the coverage report yield accurate results (i.e. no false negatives or false positives)?

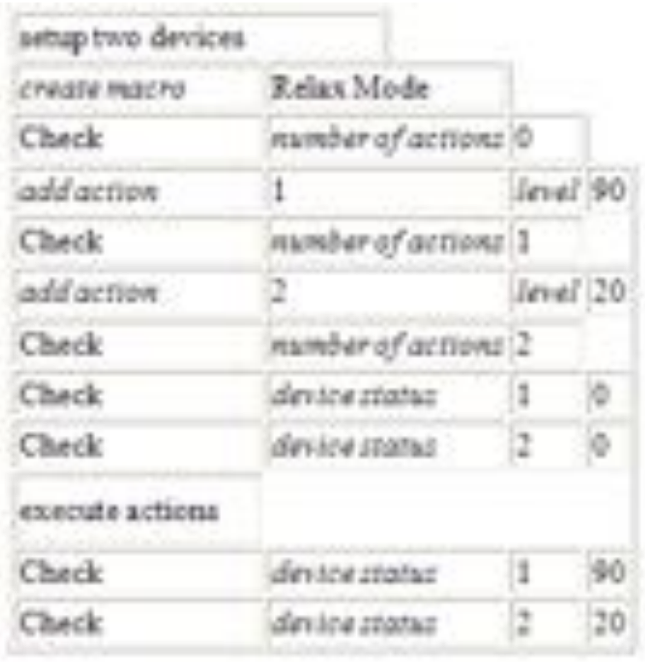
Q3. Does the coverage report provide indicators that are sufficiently accurate to make safe extraction and exclusion decisions?

9.5.2 Context

The approach is evaluated against the smart home application previously introduced as eHome. eHome is the system that I have used throughout this dissertation as an application domain. At the time of this evaluation, eHome had around 100 classes divided into model, view, controller, hardware, and communication layers. There were about 70 test cases covering approximately 90% of the model code. Throughout the eHome lifecycle, I encountered a number of variation points such as: interface touch capabilities (e.g. single touch versus multiple touch), interface orientation (e.g. vertical vs. horizontal), required modules (e.g. light control modules, RFID item tracking).

9.5.3 Core System Status

Our customer requested a feature that would enable the end user to define macros to control devices at the home. A macro is a sequence of actions to be executed on demand. The feature was defined as per the EAT shown in Figure 56.

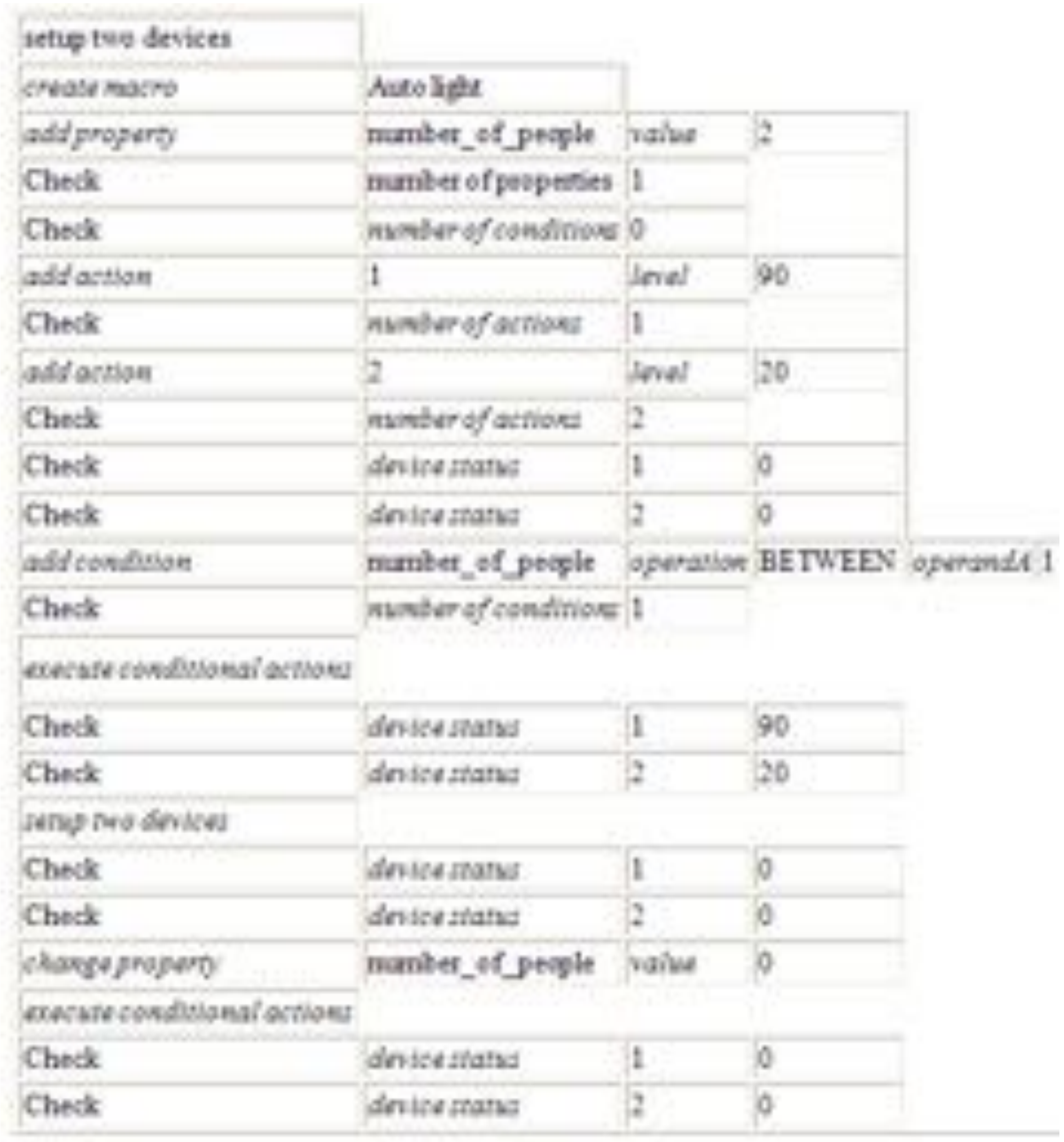


setup two devices			
create macro	Relax Mode		
Check	*number of actions*	0	
add action	1	*level*	90
Check	*number of actions*	1	
add action	2	*level*	20
Check	*number of actions*	2	
Check	*device status*	1	0
Check	*device status*	2	0
execute actions			
Check	*device status*	1	90
Check	*device status*	2	20

Figure 56 - EAT for adding macros

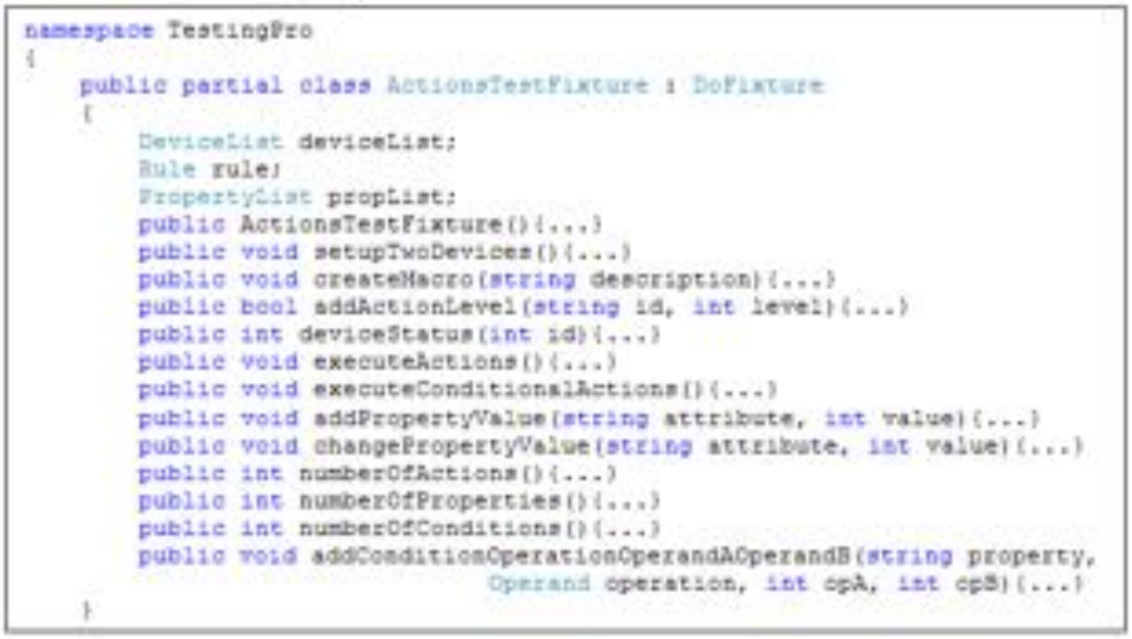
A later request from the customer was to extend the previous feature so that it is possible to optionally constrain the execution of some macros by a set of conditions. These conditions are to be defined by the end user. Figure 57 shows the EAT for this request. This EAT was added to the same test page as the previous EAT because there was a lot of overlapping functionality. A thin layer of fixture code was developed to execute both tables. Listing 12 shows what the contents of this layer look like. Production code units that made both test cases pass are shown in Table 7[*]. To summarize, the "macro addition" feature existed in the core system in such a way that the two EATs were supported. Both the fixture code and the production code incorporated the requirements of both scenarios.

[*] Primitive getters and setters, constructors and other auto-generated units are removed.



setup two devices					
create macro	Auto light				
add property	number_of_people	*value*	2		
Check	number of properties	1			
Check	*number of conditions*	0			
add action	1	*level*	90		
Check	*number of actions*	1			
add action	2	*level*	20		
Check	*number of actions*	2			
Check	*device status*	1	0		
Check	*device status*	2	0		
add condition	number_of_people	*operation*	BETWEEN	*operandA*	1
Check	*number of conditions*	1			
execute conditional actions					
Check	*device status*	1	90		
Check	*device status*	2	20		
setup two devices					
Check	*device status*	1	0		
Check	*device status*	2	0		
change property	number_of_people	*value*	0		
execute conditional actions					
Check	*device status*	1	0		
Check	*device status*	2	0		

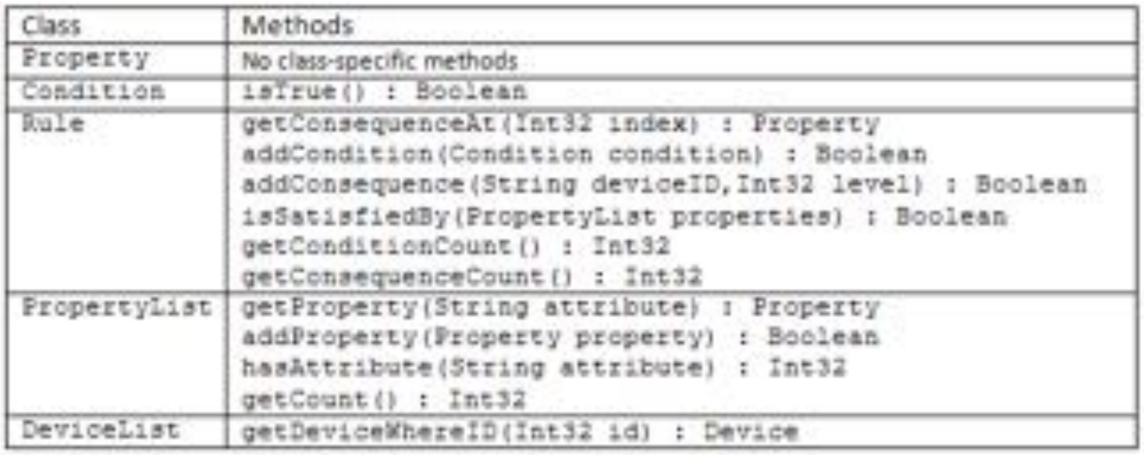
Figure 57 - EAT for adding conditional macros



```
namespace TestingPro
{
    public partial class ActionsTestFixture : DoFixture
    {
        DeviceList deviceList;
        Rule rule;
        PropertyList propList;
        public ActionsTestFixture(){...}
        public void setupTwoDevices(){...}
        public void createMacro(string description){...}
        public bool addActionLevel(string id, int level){...}
        public int deviceStatus(int id){...}
        public void executeActions(){...}
        public void executeConditionalActions(){...}
        public void addPropertyValue(string attribute, int value){...}
        public void changePropertyValue(string attribute, int value){...}
        public int numberOfActions(){...}
        public int numberOfProperties(){...}
        public int numberOfConditions(){...}
        public void addConditionOperationOperandAOperandB(string property,
                                        Operand operation, int opA, int opB){...}
    }
```

Listing 12 – The contents of the fixture code layer

Table 7 - Production code units for the core feature



Class	Methods
Property	No class-specific methods
Condition	isTrue() : Boolean
Rule	getConsequenceAt(Int32 index) : Property addCondition(Condition condition) : Boolean addConsequence(String deviceID,Int32 level) : Boolean isSatisfiedBy(PropertyList properties) : Boolean getConditionCount() : Int32 getConsequenceCount() : Int32
PropertyList	getProperty(String attribute) : Property addProperty(Property property) : Boolean hasAttribute(String attribute) : Int32 getCount() : Int32
DeviceList	getDeviceWhereID(Int32 id) : Device

9.5.4 Instantiation

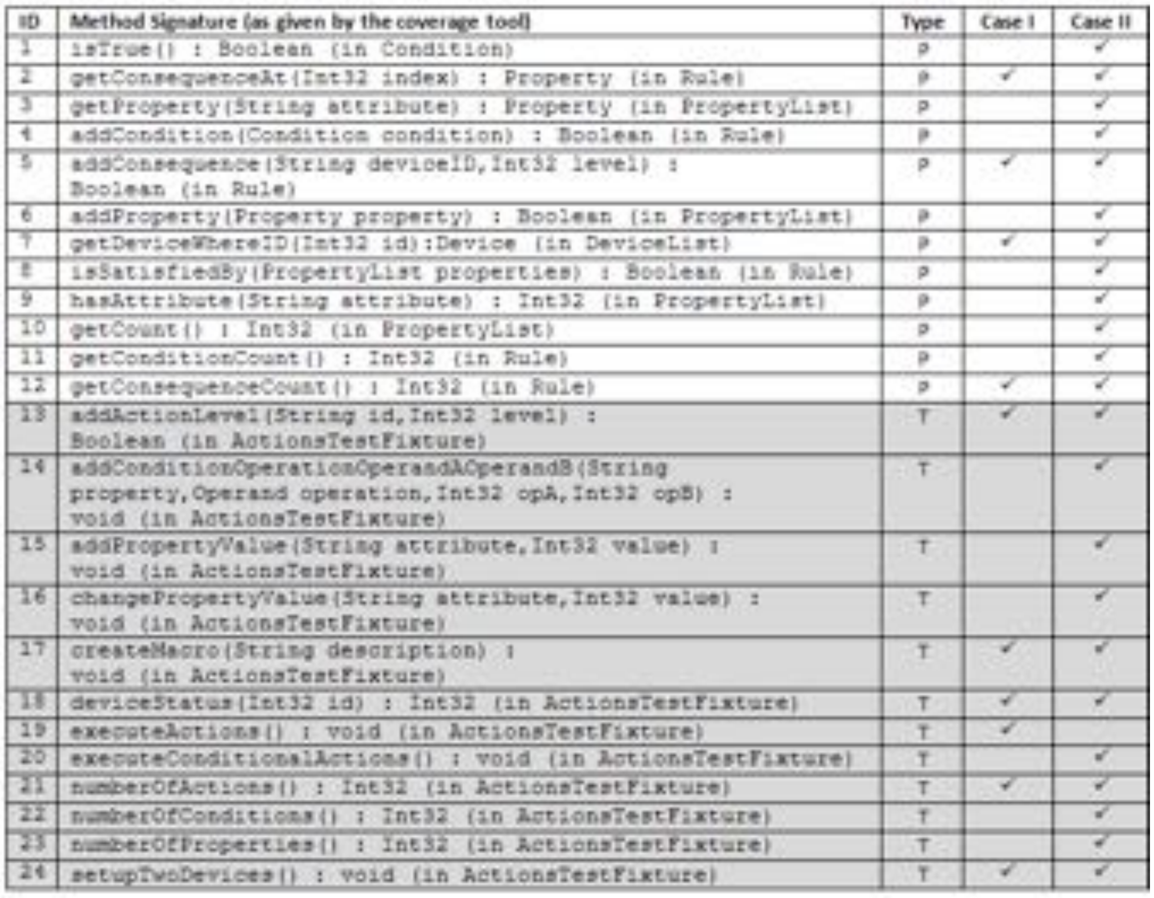
The fact that a customer requested to add a certain extension to the feature (as per the second EAT) does not necessarily mean that the feature has to exist in its fullest version in all variants of the system. Some customers would not like the complication of dealing with rules to constrain macros, and thus they are satisfied with the simpler version represented by the first EAT only. In the proposed approach, customers communicate their preferences through the selection of EATs. That is, customers can choose the scenarios they would like to see in the feature, and may exclude some other scenarios that are unneeded. Therefore, for this feature a variation point has been defined that minimally yields two variants: one that supports the addition of unconditional macros only (i.e. simple version), and another that supports the addition of both unconditional and conditional macros (i.e. complex version).

The goal is to extract just-enough code to instantiate each variant. According to the proposed approach, the selection of EATs is the first step towards this objective. I distinguish two cases:

- Case I: the customer only chooses the EAT shown in Figure 56.
- Case II: the customer chooses the two EATs shown in Figure 56 and Figure 57.

Table 8 shows the coverage results of executing the tests in both cases. The table only shows method coverage to simplify the analysis. Having access to the coverage information, the next step in the instantiation process is extracting the needed code units for the specific variant of interest.

Table 8 - AT coverage report



ID	Method Signature (as given by the coverage tool)	Type	Case I	Case II
1	isTrue() : Boolean (in Condition)	P		✓
2	getConsequenceAt(Int32 index) : Property (in Rule)	P	✓	✓
3	getProperty(String attribute) : Property (in PropertyList)	P		✓
4	addCondition(Condition condition) : Boolean (in Rule)	P		✓
5	addConsequence(String deviceID,Int32 level) : Boolean (in Rule)	P	✓	✓
6	addProperty(Property property) : Boolean (in PropertyList)	P		✓
7	getDeviceWhereID(Int32 id):Device (in DeviceList)	P	✓	✓
8	isSatisfiedBy(PropertyList properties) : Boolean (in Rule)	P		✓
9	hasAttribute(String attribute) : Int32 (in PropertyList)	P		✓
10	getCount() : Int32 (in PropertyList)	P		✓
11	getConditionCount() : Int32 (in Rule)	P		✓
12	getConsequenceCount() : Int32 (in Rule)	P	✓	✓
13	addActionLevel(String id,Int32 level) : Boolean (in ActionsTestFixture)	T	✓	✓
14	addConditionOperationOperandAOperandB(String property,Operand operation,Int32 opA,Int32 opB) : void (in ActionsTestFixture)	T		✓
15	addPropertyValue(String attribute,Int32 value) : void (in ActionsTestFixture)	T		✓
16	changePropertyValue(String attribute,Int32 value) : void (in ActionsTestFixture)	T		✓
17	createMacro(String description) : void (in ActionsTestFixture)	T	✓	✓
18	deviceStatus(Int32 id) : Int32 (in ActionsTestFixture)	T	✓	✓
19	executeActions() : void (in ActionsTestFixture)	T	✓	
20	executeConditionalActions() : void (in ActionsTestFixture)	T		✓
21	numberOfActions() : Int32 (in ActionsTestFixture)	T	✓	✓
22	numberOfConditions() : Int32 (in ActionsTestFixture)	T		✓
23	numberOfProperties() : Int32 (in ActionsTestFixture)	T		✓
24	setupTwoDevices() : void (in ActionsTestFixture)	T	✓	✓

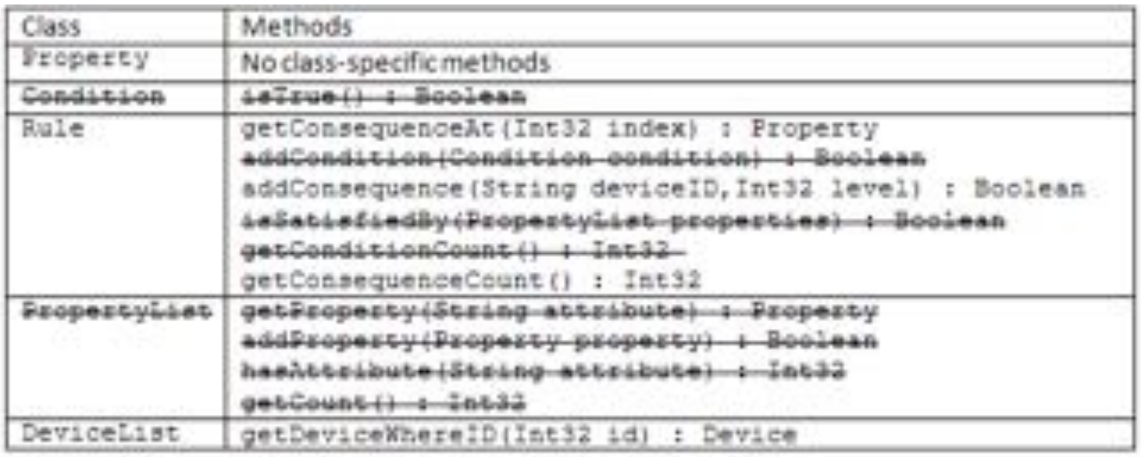
'✓' means the method was visited when the EAT was executed.

'P' stands for production code; and 'T' stands for test code.

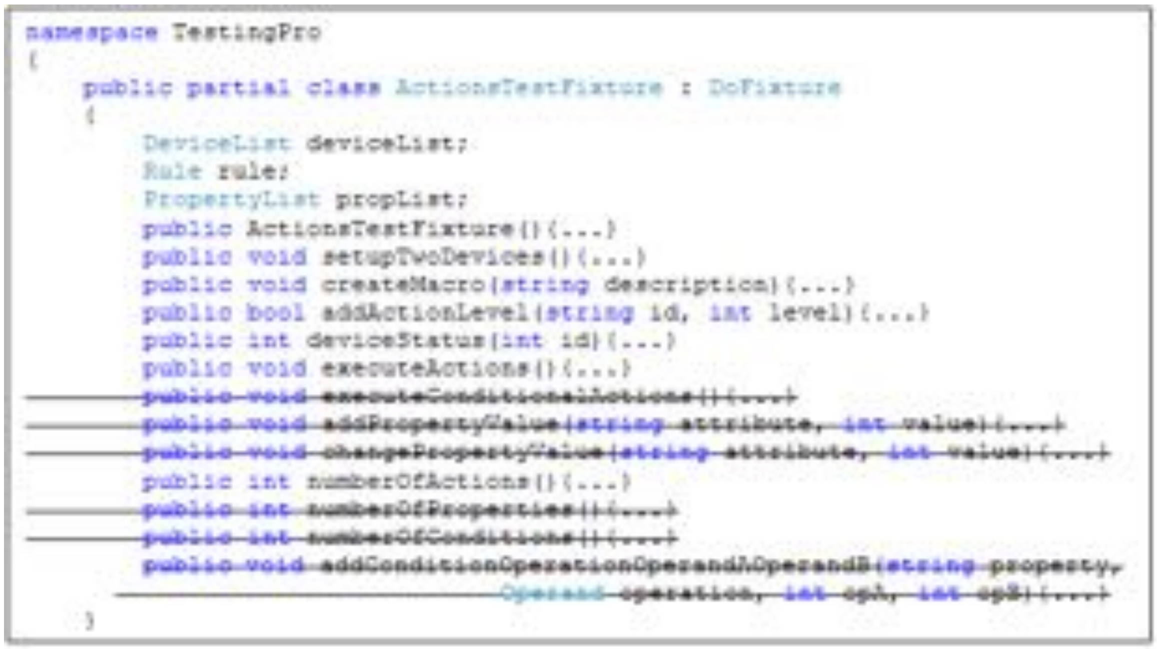
To illustrate, consider Case I where some methods are not needed for a successful execution of the selected EAT. In this case, the required production code units are shown in Table 9 and the extracted fixture code will be as in Listing 13.

As shown in the table, in some cases, all the methods in a given class are not needed. This might imply that the class itself is to be abandoned. Nevertheless, as will be discussed later, this is not always a trivial decision. The current implementation of the tool enables the user to select the tests; and then the tool automatically runs the tests, produces a coverage report, and extracts the relevant units. Automatic compilation and building of instances are not supported.

Table 9 - Production code units for Case I



Class	Methods
Property	No class-specific methods
~~Condition~~	~~isTrue() : Boolean~~
Rule	getConsequenceAt(Int32 index) : Property ~~addCondition(Condition condition) : Boolean~~ addConsequence(String deviceID,Int32 level) : Boolean ~~isSatisfiedBy(PropertyList properties) : Boolean~~ ~~getConditionCount() : Int32~~ getConsequenceCount() : Int32
~~PropertyList~~	~~getProperty(String attribute) : Property~~ ~~addProperty(Property property) : Boolean~~ ~~hasAttribute(String attribute) : Int32~~ ~~getCount() : Int32~~
DeviceList	getDeviceWhereID(Int32 id) : Device



```
namespace TestingPro
{
    public partial class ActionsTestFixture : DoFixture
    {
        DeviceList deviceList;
        Rule rule;
        PropertyList propList;
        public ActionsTestFixture(){...}
        public void setupTwoDevices(){...}
        public void createMacro(string description){...}
        public bool addActionLevel(string id, int level){...}
        public int deviceStatus(int id){...}
        public void executeActions(){...}
        ~~public void executeConditionalActions(){...}~~
        ~~public void addPropertyValue(string attribute, int value){...}~~
        ~~public void changePropertyValue(string attribute, int value){...}~~
        public int numberOfActions(){...}
        ~~public int numberOfProperties(){...}~~
        ~~public int numberOfConditions(){...}~~
        ~~public void addConditionOperationOperandAOperandB(string property,~~
                                    ~~Operand operation, int opA, int opB){...}~~
    }
```

Listing 13 – The extracted fixture code

9.5.5 Discussion

The results of this evaluation indicate that it is indeed possible to use EAT artefacts as feature descriptors for the extraction process (Q1). As observed, the most important characteristic that made EATs a good fit for this purpose is their ability to reflect the involved functionality in a concrete and readable manner. By examining different EATs, the customer can decide which scenarios are likely to be needed in their specific context. Less complex scenarios may sometimes be preferable over more complex ones. Nonetheless, an important issue has been revealed in the evaluation in relation to the UI layer. The implementation of the approach as discussed in the evaluation only handled model classes. Therefore, extracting relevant UI widgets was not possible. It is likely that similar challenges will arise when dealing with communication interfaces and hardware layers. This is because the testing tools currently available do not automate tests for these layers as effectively as they do for model classes.

As seen in the evaluation, using the coverage report provided accurate results (Q2). That is, methods that were actually needed to execute the feature of interest were all included in the report (i.e. no false negatives); and none of the methods included in the report was unnecessary (i.e. no false positives). Also, the coverage provided by the tool (visit coverage for methods) was sufficient to make safe extraction and exclusion decisions (Q3). However, in hindsight, I realize that the accuracy of the code extraction step could be improved – especially when treating fairly complex code. As seen in Table 9, some methods and even classes were not needed anymore. For the specific case studied in the evaluation, simple removal to exclude the artefacts did the job. But in some other cases, the decision to exclude these units and remove their references from the code assembly may not be straightforward. For example, the method might be shown to be uncovered because it is referenced in a condition block that was not executed. This tends to indicate that the uncovered branch is not needed anymore because there is no match for the case in the test artefacts. Still, however, taking into consideration coverage reports at different granularities (e.g. classes, methods, namespaces) and in different forms (e.g. branch coverage, visit coverage, sequence coverage) may be necessary to enhance our trust in such indications.

This evaluation is limited by the fact that it is a self-evaluation which might have introduced some bias in selecting the scenarios to be treated. Another issue comes from using a single case to evaluate the approach, which may limit the generalizability of the findings.

9.6 Chapter Summary

This chapter presented an approach to derive individual products from a given SPL using the proposed framework described throughout this dissertation. The proposed approach takes into consideration the minimal documentation (or the absence of it) in agile environments. Instead, EATs are used in the extended feature model to select those scenarios the customer is interested in. The derivation process can then be achieved in two different ways depending on the implementation technique being followed. The chapter provided an evaluation for the approach using a proof-of-concept tool. The approach has shown to be sufficiently accurate, but certain improvements are needed to make it more practical and trustworthy.

CHAPTER TEN: ISSUES AND CHALLENGES IN INDUSTRIAL CONTEXTS[*]

10.1 Preamble

This dissertation has so far presented a novel product line framework that takes into consideration the philosophies and practices of agile software development. The different parts of the framework have been discussed in detail and assessed through a number of research techniques. Independent of the specifics of my framework, this chapter investigates the issues and challenges associated with adopting a SPL framework in an industrial context. That is, I look more into the issues and challenges that arise should a company decide to adopt a new SPL framework such as the one proposed in this dissertation. The goal of this research inquiry is to construct a comprehensive understanding of the transferability impediments of frameworks developed in academic environments. This chapter tackles this issue in two parts. The first part is concerned with the general issues that are associated with adopting a new SPL practice. The second part is concerned with the specific issues arising from combining agile methods with SPL practices.

10.2 Research Instrument

This chapter addresses the following question:

RQ 6. Independent of the specifics of the proposed framework, what are the technical and non-technical impediments that need to be taken into consideration before a new SPL framework becomes feasible in an industrial context?

[*] This chapter is based on a journal article currently under review [Ghanam2011b]. Co-author permission is attached to Appendix B.

Under the general research question above, the study addresses the following questions:

Q1. What are the general issues and challenges that organizations are likely to face when making the transition to a new SPL framework?

Q2. What are the specific issues and challenges imposed by recent trends in modern software engineering such as agile methods, distributed development, and flat management structures within the context of SPLs?

In the study, data was collected using an ethnographic approach [Hammersley1983]. Ethnography is a data collection approach that involves spending time in the field to make first-hand observations. The researcher interacts with the subjects of interest in a natural (as opposed to controlled) setting in order to obtain a holistic view of the context pertaining to the problem under investigation. The rich data collected over the course of the study – including observations, questionnaires and interviews – requires a methodical qualitative approach to analyze [Dittrich2007]. The collected data was analyzed using Grounded Theory [Strauss1997]. Grounded Theory is a qualitative research method in which the generation of a theory occurs by looking into the collected data for patterns and concepts. The use of Grounded Theory as a powerful tool in qualitative research has been abundant in the past few decades.

10.3 Study Procedure

10.3.1 Study Context

The study was conducted in a software company in Scandinavia. To comply with the non-disclosure agreement signed with the company, I use the pseudonym "Scandin" to refer to the company thereafter. In its domain, Scandin is considered one of the most influential players in Europe, and it has a significant impact on the market in North

America and other parts of the world. The company provides solution packages to individuals as well as corporations and third-party service providers. Scandin has over 800 employees – about half of them work in software development. Scandin can be described as a medium-scale organization (compared to larger organizations in Scandinavia like Nokia). In addition to its headquarters in Scandinavia, the company has four other locations (aka. business units) in other parts of Europe and Asia. The company uses outsourcing for some software projects. Scandin has a flat management structure in the sense that they have cut middle-management layers and provided a less-authoritative organizational structure to ensure the direct involvement of employees in the decision making process. About eight years ago, Scandin took its first steps to adopt agile software development. Software projects were mainly centered on the development and maintenance of a single solution that required high responsiveness to market needs in order to be able to compete globally. Recently, the company – driven by its new business strategy – decided to build a portfolio of products to target new markets and provide a range of service packages to online customers. For that purpose, the technical strategy was to implement a SPL approach where all products in the portfolio are to be built using a common infrastructure. This infrastructure consists of a number of software platforms built in-house. The term platform generally refers to a set of subsystems and interfaces from which a set of related products can be developed [McGrath1995]. Being a SPL technique, using a platform strategy involves reusing relevant artefacts in the platform, and then a customization process to produce unique products [Pohl2005]. Some parts of the platforms are derived from existing products, and other parts are to be built from scratch. That is, the company needed to build platforms on top of which teams across the company should build products and services. For any given product, there is a backend side and a client side. Products in the portfolio have common aspects in both the backend (e.g. licensing, updating) as well as the client (e.g. user interface library). Therefore, platforms were needed on both sides.

There is a raft of factors that made Scandin a good fit for the purpose of my study. For one, Scandin is an agile organization that was trying to implement a SPL strategy. Secondly, in their implementation of SPL practices, technical leads in Scandin were interested in a reactive approach to leverage their existing artefacts. And finally, Scandin had the willingness to share their data and processes for research purposes (under a non-disclosure agreement). At the time of this study, the company was in the transitional phase – some parts of the platform had already been built and used while some other parts were still being constructed.

10.3.2 Data Collection and Analysis

10.3.2.1 Data collection

Data for this study was collected by conducting 3 to 4 full-day visits in the company every week, over a period of 6 weeks. During these visits, I adopted non-participant observation by attending presentations, demos, planning meetings and status-update meetings (aka. scrum meetings). Furthermore, in order to get a first-hand impression of the interactions and communication channels, I arranged with the company to stay in close proximity to people of different roles in the organization, namely: senior managers, architects, team leads, and developers. Over the course of the study, I conducted 16 in-depth interviews with individuals of different teams and roles. The interviews lasted between 25 and 72 minutes each. The interviews were audio-taped and transcribed.

In the selection process of interviewees, the set goal was to get a sample of individuals that covered the different aspects related to my research interest. Specifically, I was interested in the following aspects: management (directors influencing platform-related decisions), platform development (teams developing the platforms), and product development (teams building on top of the platform). The initial group of interviewees was selected collaboratively by myself and a liaison in the company. During this initial phase, I used snowball sampling to prepare for the second round of interviews. That is, I used the collected data as well as suggestions from the interviewees to guide the selection process of other interviewees. I interviewed representatives of 8 different teams, 3 of which were working on 3 separate platforms as part of the common infrastructure.

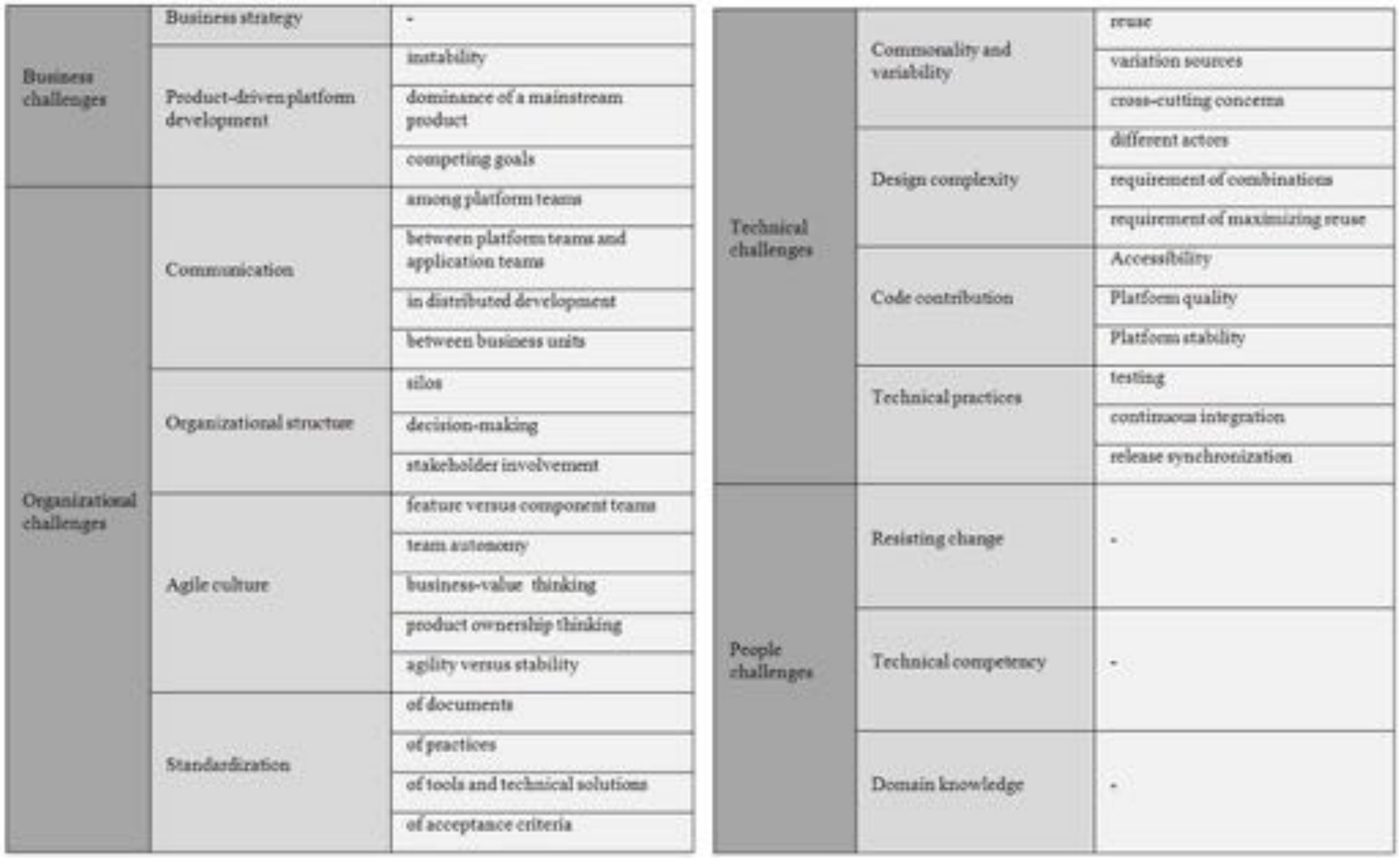
The interviews were semi-structured and took various directions based on the interviewee's responses. The role of the interviewee was also vital in determining the direction and focus of the interview (i.e. managers focused on high level issues, whereas team leads and architects focused on technical details). Generally, interviewees were asked questions to describe their role and team responsibilities, how they relate to other teams, what issues or blockers they have been facing when building or using the platforms, and what things they thought were missing but would be beneficial to have. The interviewees were also asked to explain certain aspects of the platform and sometimes to draw diagrams and figures to illustrate their understanding of the overall architecture. The artefacts produced by the interviewees helped in understanding the problem and the context better and revealed important issues underlying communication within and across teams. These artefacts also helped in the interpretation of the data collected during the interviews. I was also granted access to documented material communicated among the upper management to obtain a better understanding of the company's vision and strategy. Data from the interviews, the documents, as well as my observations and diaries (consisting of hundreds of field notes) were all used to complete this study. The data collection phase stopped when I started to get no new insights from new rounds of interviews.

10.3.2.2 Data analysis

As mentioned previously, Grounded Theory was used as an analysis instrument for the collected data. I started by iterating over the collected data to assign codes, and I refined these codes as more data was coded. This involved renaming, merging, or splitting some codes multiple times. The codes were grouped into larger representative concepts and categories that evolved through multiple iterations by going back and forth between different interviews and the other data sources. The data that had been collected and analyzed during the initial phase of the study was used to conduct selective sampling (as opposed to random sampling) when recruiting participants for the interviews that followed. The taxonomy of issues started to saturate after having analyzed about 70% of the data. Having this taxonomy developed, I compared my findings to the existing body of literature in relevant research areas.

10.3.3 Findings

This study revealed a raft of issues and challenges that medium sized, distributed, agile organizations are likely to face when reuse becomes a strategic objective – especially when their context is similar to the context of Scandin as described earlier. The challenges as manifested in the data are captured in the tree shown in Figure 58. The tree was kept at a manageable size by merging similar concepts and limiting the depth of the tree to three levels. The challenges were classified under four main categories, namely: business challenges, organizational challenges, technical challenges, and people challenges. Each of the categories is divided further into subcategories. The richness of some subcategories as evident in the collected data required that they be divided further, while other subcategories are kept at the second level. In the following sections, I discuss the findings in more detail.



Business challenges	Business strategy	-
	Product-driven platform development	instability
		dominance of a mainstream product
		competing goals
Organizational challenges	Communication	among platform teams
		between platform teams and application teams
		in distributed development
		between business units
	Organizational structure	silos
		decision-making
		stakeholder involvement
	Agile culture	feature versus component teams
		team autonomy
		business-value thinking
		product ownership thinking
		agility versus stability
	Standardization	of documents
		of practices
		of tools and technical solutions
		of acceptance criteria

Technical challenges	Commonality and variability	reuse
		variation sources
		cross-cutting concerns
	Design complexity	different actors
		requirement of combinations
		requirement of maximizing reuse
	Code contribution	Accessibility
		Platform quality
		Platform stability
	Technical practices	testing
		continuous integration
		release synchronization
People challenges	Resisting change	-
	Technical competency	-
	Domain knowledge	-

Figure 58 – Tree of challenges – a '-' means the subcategory is not divided further

10.4 Business Challenges

By “business” I refer to the many aspects involved in running a profitable organization including the organization’s vision and strategy, sales and marketing, and competition. The findings show that there are two main issues that can introduce major challenges to introducing a platform strategy, namely: the business strategy, and product-driven platform development.

10.4.1 Business strategy

Scandin’s new business strategy to target a new segment of customers in their market had a huge impact on platform development. The common platforms needed to adjust the services they had previously provided to products in order to accommodate the new scenarios those products were required to support (e.g. by the marketing department). This resulted in considerable reengineering of some existing components. When asked why a specific component of the platform was undergoing major reformation, one of the platform architects responded that it was due to:

> *“... the new way [Scandin] wants to make business with customers on the retail and OEM level but also with operators…”*

Although this issue is not specific to platform-centric development, the experience of Scandin shows that when adopting a platform-centric approach, the amount of rework and testing that needs to be done is usually multiplied because changing a platform component has consequences on all products that rely on that component.

10.4.2 Product-driven platform development

In traditional models of building platforms, a platform-then-product philosophy is dominant as evident in practices like software product line engineering where there is an emphasis on developing domain artefacts and then application artefacts [Pohl2005]. This sequence means that an organization does not start building products until development of the platforms underlying these products has made considerable progress. On the other hand, in Scandin, I noticed that platform development was product-driven in the since that some platforms were derived from a number of existing products as well as from the requirements of an ongoing project that was considered the first adopter of the platform. In that project, the product that relied on the platform was being developed at the same time as the platform. As explained by technical leads, product-driven platform development was their strategy to:

a) reduce the conceived risk of lost investments due to over-engineering aspects of the platform that cannot be reused later – either because they turn out to be unnecessary or too complex to reuse, and

b) achieve a faster return-on-investment by delivering products to end-customers more quickly than they would have been delivered if a sequential approach had been used. However, the findings show that the latter approach introduces its own risks and challenges, such as:

Instability. Some components in the platform may not be mature enough when they are used in products, which causes products depending on them to be unstable. As one of the managers put it:

> *"[Some products] break every second build… the platform is not stable enough in which they are building their architectures."*

The dominance of a mainstream product. If the platform development is driven by one product that is considered a main revenue stream, which is the case in Scandin, then the priorities in the platform development are likely to be coupled tightly to the needs of such a product:

> *"... we tightly plan our sprints only based on the [mainstream] project priorities... Now it's about the [mainstream] product but then we know that we need to be able to serve the [other] products later on."*

This may cause the platform to become under-engineered – meaning that the components may become too specific to the needs of the mainstream product (e.g. a specific operating system) rendering component reuse challenging across other products. In Scandin, some components – that were supposed to be cross-platform[*] – became specific to the operating system that was required by the mainstream product:

> *"... to further develop [the platform] we have to take it to cross-platform and operating-system platforms... That's not there."*

Furthermore, focusing too much on the needs of the mainstream product causes other teams who have dependencies on the platform-to-be to become ignored and uninvolved.

> *"... because we [platform team] are fully allocated in the [mainstream] project, it is tough to get the time to actually take the other parties into consideration."*

In Scandin, this issue had strong effects on some teams who chose to start implementing their own components resulting in redundant code.

Competing goals. Product teams are pressured by their technical leads to start using the platform as soon as possible (to avoid any redundancy). One of the technical leads explains that:

[*] The term "cross-platform" in the context of this study means that the implementation is agnostic to the operating system.

"... the roadmap and goal [set for the product teams] would be to [reuse] all the [platforms] that have been built for the [mainstream] project."

On the other hand, other product-specific goals such as fast delivery are pushed by the business leads. Considering the overhead associated with making the transition to the platform (e.g. learning, asking for changes, customizations), some teams in Scandin perceived that it was faster (or less burdening) to create their own artefacts than to reuse somebody else's.

10.5 Organizational Challenges

A wide range of issues and challenges arise due to the nature of platform development that requires participation and involvement at the organizational level as opposed to the team or business unit level. In the following subsections, I discuss the organizational issues encountered in the data.

10.5.1 Communication

Platform development introduces more dependencies in the organization than what would normally exist without such a strategy. In Scandin, these dependencies exist between the platform teams themselves, the platform teams and the product teams, and the different business units in the organization. Distributed development exacerbates this communication challenge as will be explained.

Communication among platform teams. Our findings show that platform teams need to communicate for a number of reasons such as: 1) assigning responsibilities to components, 2) resolving dependencies between components, 3) agreeing on protocols and internal interfaces, 4) synchronizing releases, and 5) arranging for resources that need to be shared. In the case of Scandin, one of the main challenges is to motivate the individual teams to talk to each other beyond formal meetings (if any) where things might have been overlooked or misunderstood, and beyond reading documents (if any) that might be outdated. When this motivation is not there, developers resort to their hunches to resolve a dependency or may integrate with other components in a less than ideal way.

> *"The biggest challenge [is] to get people motivated when they have a dependency for outside ... to get the communication started. And even though we have things like scrum of scrums.. but it still does not mean that everything will be brought up there."*

Also, in cases where this communication is not effective, teams may work on overlapping areas of the platform causing redundancy and rework as I observed in the company.

Communication between platform teams and product teams. Teams in Scandin need to communicate at this level because platform teams provide services that are consumed by product teams. For one, product teams need to know how to access and integrate with the platform. Also, product teams provide feedback to platform teams on existing features and report missing ones. As one of the platform developers pointed out:

> *"... for various reasons there might be a product level feature [requested of the platform teams]. There have been a few [cases] where something is needed [from the platform teams] by [the product teams.]"*

Achieving this communication, however, can sometimes be tricky. As I noticed in the company, when an issue arose in product development, some developers found it easier and quicker to find workarounds which might be redundant to what already existed in the platform. This not only caused a lot of rework and redundancy in the code, but it also made testing and maintenance cumbersome in the future. Therefore, for this communication to be effective, product teams need to understand the value of keeping communication channels active at all times (i.e., realize the technical problems associated with redundancy).

Communication in distributed development. In addition to the inherent challenges of communication in collocated development, distribution of teams over the world introduces further challenges. When some distributed teams in Scandin used tools like instant messaging and shared desktop to hold meetings, the communication did not always serve its purpose:

> *"... [name of a unit in the company] seems not to have too much problem using this communicator and shared desktop and so forth. Some other units have serious problems with that."*

In addition to discussing this issue with individuals in the organization, I also attended an online meeting to have a better understanding of the problem. One of the factors that made this communication a challenge was the different time zones which made arranging meetings more difficult, and sometimes resulted in the meeting time being inconvenient to one party. Other factors included the absence of non-verbal cues such as body gestures and facial expressions especially during screen sharing, and the cultural differences between Scandinavia and other parts of the world where the language or social protocols were a barrier. For instance, in Scandinavia where the management structure is mostly flat, a verbal agreement on the phone was sufficient for developers to start executing a plan. On the other hand, in other parts of the world where authority is very hierarchical, the teams could not execute their plans until they got approval from the relevant line of management in their business unit.

Communication between business units. Because business units shared common platforms, they needed to communicate. I will talk about this aspect of communication when discussing "silos" in the next subsection.

10.5.2 Organizational structure

In this section, I discuss the impact of how the organization is structured in terms of business units, teams, and management on platform development. I focus on three main issues that I found evident in the data as follows.

Silos. The analogy to a silo is often used to describe the state of a certain part of the organization that seems to stand alone and not interact enough with the other parts. As illustrated in this study, silo thinking is a result of an organizational structure where business units or teams act as independent entities with their own local management and no motivation to adhere to a centralized decision-making body or to share information with other units. In the context I studied, the silo could be a single team or a whole business unit. The findings show that the silo problem is by far the most serious challenge that faces the organization's transition to a global reuse strategy. Individuals of both management roles and technical roles repeatedly mentioned the term "silo" and complained about the matter almost equally, for instance:

> *"we have business units… How do they communicate today.. not too well. These silos they don't talk too much [to each other]."*

The data revealed a number of reasons for silo thinking as shown in Table 10, and a raft of consequences they have on platform development as shown in

Table 11.

Table 10 - Reasons for silo thinking

Reasons for silo thinking in the organization		
Resource allocation	Business units would rather allocate their limited resources to meet their local deadlines unless they are forced to participate in a corporate level project.	*"And the business units are not forced unless there is a big project that forces them to put their resources aside for this kind of activity."*
Specialization	Specialization in a certain domain makes it difficult to understand the benefits of communicating platform related issues to others (e.g. promoting reusable components).	*"… if it's a business-specific platform there is no communication outside of that business unit."*
Lack of motivation	Unless the business unit sees a direct value of sharing a platform at the corporate level, they are not willing to do so.	*"They don't have any interest whatsoever in taking this [platform] to corporate level unless they have a cost saving reason."*
Competing targets	Focusing on meeting unit targets and disregarding corporate targets makes platforms too specific to the business unit.	*"[the business units] will just build for business target, and when business units disassemble, the assets might be useless."*
Apathy	Some component teams are indifferent about anything outside the locality of their team. The quote is the response of a senior software engineer in a component team when asked who drives the requirements of their component.	*"I really haven't spent that much time to really figure out how this thing goes from up to down."*
Evaluation apprehension	The fear of being criticized, supervised or controlled inhibits sharing and communication.	*"[the business units] feel unease when they have to come and present their plans to the [corporate]."*

Table 11 - Consequences of silo thinking

Consequences of silo thinking		
Missing the big picture	This results in not having a common understanding of the platform architecture which in turn causes other problems such as redundancy and false assumptions.	*"they're working in their silos and the changes are so, that only the projects [they] have been working on lately have good common view."*
Redundancy	Business units and component teams run the risk of duplicating an existing component that has been developed somewhere else. Sometimes the duplication is a result of not being aware of assets outside the silo, or not willing to reuse something that was invented somewhere else.	*"We also have three systems for that, [and] two systems to update the databases, that's awful.. and this is because of the business unit silos."* *"... because somebody thinks they can't use it because it doesn't have their business unit label on it…"*
No long-term thinking	The challenge here is to strike a balance between meeting the short-term goals of the business unit and the long-term sustainability goals of the platform.	*"Business units have to balance their business drive [with] the long-term sustainable architectural base. There is no decision on that."*
No visibility of reusable assets	Platform assets get buried within the business unit or a certain component team which results in a lot of duplication and missed reuse opportunities.	*"we have been digging the assets of the company here for the last year trying to hire and elaborate those platforms, get them on the map..."*

Consequences of silo thinking		
False assumptions	Poor communication with other business units or teams results in false assumptions about assets. In this case, an internal decision could have cost the company a fortune.	*"Later, [a business unit] wanted to do [a service] and proposed doing a new system... The reason was because mobile protocol cannot work the same as Win protocol... which wasn't true. It was an assumption."*
Platform divergence	A given platform can initially be used by different business units but then internal decisions result in different branches of that platform. After a while, the branches diverge so much causing the platform to become too specific to a certain operating system or product, or even causing the loss of a common underlying model.	*"in the common library there are OS adaptations in the code branching which is not too healthy... when they have been building this current architectural base, they have been building it in silos."* *"there was actually separate business units that worked independently and resourced independently... So we have kind of the same base model but there is added [parts]. Many flavors from the same model."*

Decision-making. This issue is partially related to the silos problem, but it encompasses other aspects as well. When the teams and business units have a sense of federalism (which is very evident in the data), making decisions related to the platform becomes a challenge. One case I came across in Scandin involved decisions that needed to be taken on whether to reuse an existing platform or take a different direction such as building an independent variant to satisfy a certain business concern. On the one hand, the corporate had economical reasons to push reuse, but at the corporate level it was often hard to see all the intricate details of the specific business concerns, hence making such decisions challenging. On the other hand, when these decisions were left to the concerned teams and business units, a number of issues caused the decision making process to go astray. For one, the individual units tended to choose the direction where they saw short-term gains as opposed to thinking about the long-term goals (e.g. sustaining the platform). And there was also the "not-invented-here" mentality that biased business units to develop their own components as opposed to reusing others'.

An attempt by the company to have a centralized decision-making body did not solve this problem. Business units were likely to assume ownership of products and therefore they deemed such decisions an internal matter. As a corporate manager explained:

> *"And then if [the business unit's decisions] don't get approved [by the centralized body], they kind of tend to think that well.. this is our internal decision…"*

Moreover, when the corporate made a decision to invest into building a platform, political challenges arose when trying to kill ongoing projects that might have been redundant to the services provided by the platform. Or even more challenging was the attempt to get a business unit to retire their old systems and migrate to the new platform:

> *"This is of course an organizational issue to say to somebody that this thing that you have been building for five years is actually going to be discontinued."*

Stakeholder involvement. Due to the fact that platforms are an enterprise-level concern as opposed to product or team level concerns, it seems to be a challenging matter to get all stakeholders involved from the business side and the technical side. For a medium-scale organization like Scandin with hundreds of engineers and other personnel in sales and marketing, the challenge was to first identify who had a stake in a given platform.

That is, who should actually be involved in planning, building or using the platform? This became more difficult when the platform was to serve different business units with various concerns and competing goals. Distributed development and outsourcing were other factors that added to the complexity of this issue.

For example, in order to build a platform for licensing in a unified way across all products in a given portfolio, the company first needed to involve all the parties responsible for the older licensing models, and the parties responsible for merging these models into a single unified licensing component in the platform. This meant getting on-board the technical leads and architects representing products using the older models, products that will use the new model, and products that are specific to certain operating systems. From the business side, solution managers and business analysts were also involved to make sure the technical solution did not affect a business case in a negative way (e.g. affecting a revenue stream or an agreement with a third-party).

One way the company tried to overcome this issue was by holding workshops to allow teams to discuss common issues and understand their different needs from the platform:

> *"we have had several workshops with [business unit name] guys and the [another business unit name] guys and we have mapped all the differences."*

Unfortunately, in some cases involving all stakeholders at once was infeasible due to scalability issues. In such cases, the company chose to postpone the involvement of some parties to a later stage:

> *"we have [team's name], I don't think [they are] going to be [involved]in the project... at least not right now."*

10.5.3 Agile culture

In the data, I found that there is a need to adjust agile principles and practices before they can be employed in a software platform context. In this section, I list some of the challenges imposed by the agile culture in the organization.

Feature versus component teams. Feature teams usually assume end-to-end responsibilities in a given system by orienting their work around features (aka. stories); whereas component teams focus more on delivering a sub-system (aka. component) that interacts with other sub-systems in order to be useful [Larman2009]. Due to the focus on delivering tangible value to the end-customer, agile advocates promote the idea of building teams around features rather than components [Larman2009]. For some people in Scandin, this idea created the perception that component teams are always disadvantageous. As one of the technical leads denotes:

> *"It's been told to me that it's bad to have component teams in the agile world which cannot be end-to-end responsible."*

However, in the context of platform development in this company, there seemed to be a need for a combination of both. The interviewee here explains that certain services are so fundamental and expensive that they may require a dedicated component team as opposed to being maintained by a feature team as part of an ongoing project:

> *"End-to-end responsibility, very tough to implement... One thing we need to accept as an agile organization is that there are certain services that are too expensive to develop [as part of] the project."*

Team autonomy. The other issue that was evident in the data is high team autonomy. When members of highly autonomous teams stayed together for a long time, those teams gradually turned into silos. This phenomenon often resulted in the consequences of silo thinking as discussed previously. Moreover, in some teams, high autonomy had an impact on decision-making where the team considered certain issues internal without paying much attention to the consequences of their decisions on the underlying platform. The decision-making aspect has already been discussed in more detail in a previous section.

Business-value thinking. In agile organizations, there is a strong emphasis on delivering business value [Schwaber2004]. The challenge, however, is that business value is not always immediately visible or may not influence the customer directly as I saw in Scandin. That is, the transition to a platform strategy provided advantages for Scandin as a business, but from an end-customer's perspective, nothing has changed. The findings show that in an environment where there is strong business-value thinking, it is a challenge to motivate certain teams and individuals to invest into adopting the platform. In this quote, a lead architect explains why some teams could not see the business value in transitioning to the platform strategy:

> *"[The platform strategy] is new for us, but it's not producing any new stuff for the customers... The whole stuff is invisible for them."*

Product ownership thinking. Some interviewees in the study raised the issue that some teams and product owners in different business units had been very protective of their assets and products making the transition to a platform strategy more difficult. This is mainly because teams owning a certain component preferred dealing with that component rather than retire it and maintain a shared component in the platform. A technical executive explains his strategy in dealing with duplicate components:

> *"before [platforms] become de-facto, it requires killing duplicate systems and preventing them from coming up again."*

Another issue was that when it came to some platform components, product ownership was not as explicit and clear as it was in individual products. That is, in many cases it was not clear who owned a component in the platform that was shared across different teams and products.

Agility versus stability. As described earlier, in Scandin's case, the platform is product-driven which means that some platforms were derived from a number of existing products as well as from the requirements of an ongoing project. I noticed that this notion introduced the challenge of striking a balance between the stability of the platform and the ability to change often and add features. On the one hand, platform stability was key, because many products relied on the platform as a common foundation and therefore it had to be trusted and not changed very often. Especially for critical components, being part of an ongoing product development with changing requirements imposed certain risks:

> *"... One thing we need to accept as an agile organization is that there are certain services that are too expensive to develop [as part of] the project."*

On the other hand, it was also important for the company to respond to the need of the products in an agile manner in order to be able to compete in their specific market.

Another issue under this category was raised by some participants. When a specific product requests a change in the platform that involves a cross-cutting concern such as usability, it will be challenging to make a choice from the two possible options. The first is to honour the change request to satisfy the customer at hand (following agility principles), in which case all products relying on that aspect would be affected (i.e. causing instability). The other option is to ignore the request until it proves to be an issue in other products too, but that may come at the expense of the satisfaction of the customer at hand. This challenge, however, was not supported by a specific example, so I consider it more of a concern than an actual problem.

10.5.4 Standardization

Some of the challenges I came across in the data were related to the lack of standardization in the organization which affected communication and made reuse more difficult.

Standardization of documents. Some documents are circulated among platform teams, and between platform teams and product teams. When the documentation practices were not consistent, individuals were less likely to refer to these documents. As one of the interviewees stated, standardizing the retrieval process of documents plays an important role:

> *"if I have to find how this works, I know where to go find the information and everything is in one place."*

Other interviewees pointed out that the inconsistency across teams in the format of their documents and the level of details made finding information more difficult.

Standardization of practices. When different teams and business units contributed to a shared platform, the lack of standard practices such as code conventions and testing practices appeared to have a detrimental effect on collaboration and made reuse difficult. One of the interviewees asserted that the lack of code conventions was one of the reasons it was difficult for his team to reuse others' code:

> *"there is actually nothing really that would be [considered] program wide like code conventions."*

Standardization of tools and technical solutions. When each team in the organization makes their own decisions on what tools and technical solutions they want to use in a given project, as the case in Scandin, platform development and use seems to become more challenging. For example, developers in Scandin need to deal with a number of version control systems and a wide range of testing and continues integration tools before they could contribute to the platform:

> *"I wouldn't know where to find all of these guys' code… It's still not company-wide that there would be even like a nice recommendation that everybody [should] use SVN not GIT or CVS."*

Standardization of acceptance criteria. Teams in Scandin often defined a list of criteria that needed to be met before a feature or a task was considered done. I noticed a range of things that were considered in different teams such as: successful compilation, passing regression tests, having a predefined bare minimum amount of test coverage, and updating relevant documents. The fact that these criteria were not standardized across the organization caused some teams to lose confidence and trust when reusing components developed by others or when referring to documents written by different business units. For example, one team that put a significant emphasis on reliability in their engines refrained from using other code that did not adhere to the same quality standard.

10.6 Technical Challenges

Developing software platforms is an engineering problem that imposes many technical challenges. The collected data shows that many of these challenges are due to the fact that a platform needs to satisfy a range of varying requirements in a certain domain, and that many products rely on the platform as the foundation of their functionality. The major challenges that are identified under this category include: commonality and variability, architectural complexity, code contribution, and practices.

10.6.1 Commonality and variability

As a reuse strategy, platforms provide a common infrastructure on top of which different products can be built. However, components in the platform need to accommodate possible variants so that customization is possible for different business and technical needs [Jianhong2006]. Managing commonality and variability is not always straightforward, and that is why commonality and variability management is a topic by itself in fields like software product line engineering [vanGurp2001]. I discuss commonality and variability challenges around three axes: reuse, variation sources, and cross-cutting concerns.

Reuse. Managing reuse in the organization is essential for a successful platform development. In the study, I found that this entails not only finding opportunities for reuse in new products, but also dealing with existing redundancy. One of the main challenges I came across in this regard was to detect redundancies in legacy code. Developers often use ad-hoc techniques to reuse code such as copy-and-paste (i.e. code cloning); and research has shown that code clones are difficult to trace and often introduce bugs in the system [Li2006]. In Scandin, a particular problem with clones was that if a critical change was made to the original code, the duplicates did not get the update, and when they did, they had to be maintained separately. One of the platform teams explained the problem as follows:

> *"we kept having these pieces of code that were copied [from our platform] and pasted somewhere else [in different products]... then we optimized [the code] and nobody gets to use [the optimized version] because it wasn't in any common place and there was no process [to trace reuse instances]."*

As I noticed, redundancy also resulted from poor communication between teams, which yielded multiple implementations of similar services at times.

After detecting redundancy, the next challenge in managing reuse is actually dealing with redundant solutions. As one of the technical leads explained, the process of retiring redundant components and replacing them with a common foundation requires meticulous care to ensure a smooth and stable transition:

> *"first you need to unify [the solutions]... If we cannot make those [duplicate solutions] coexist, then one of those need to take the whole responsibility, but it means one of those systems continues in production and others are retired and taken to maintenance only."*

After the duplicate solutions have been abstracted into a reusable component in the platform, there is one more challenge of making the new asset visible for future projects. In Scandin, visibility was an issue:

"we are not sharing all the code we could, because it wasn't under [business unit name] before this. So I am not sure if they would even know as well what we already would have available."

Variation sources. Assets in Scandin's platforms had to deal with multiple dimensions of variability in the product portfolio, which imposed a real challenge. Some variations were due to business needs which required different models for different types of customers:

"we have different license models depending on the business case. We have one model that goes into the stores. Then we have the [third-party] model where we actually sell through the [third-party]... Then for corporate, we have a couple of different models."

Operating systems were another dimension of variation:

"...you have Androids, iPod, iPad, Mac, Mobile Win ... if each OS has a different client code, you might have a different backend.. it becomes very tedious to maintain, it becomes a burden."

In Scandin, variation also occurred due to the concept of combinations of services where every product team (or sometimes every customer) should be able to package their own combination of services from the platform.

Cross-cutting concerns. Things that cut across different products that use the platform (e.g. usability in the case of Scandin) become a challenge in scenarios where a change is needed only in a subset of the products but not all. This may require treating this concern as a new variation point which adds to the complexity of variability in the platform.

10.6.2 Design complexity

This issue has been brought up by lead architects as one of the main technical challenges in platform design. I investigated this issue further by looking at design artefacts to identify the reasons of added complexity, namely:

Different actors. When variability in the platform was driven by the business trying to target different markets or customer bases, this yielded multiple actors, each with their own needs of the platform.

The requirement of combinations. Due to the requirement of being able to combine Scandin's components and services to build unique products (aka. suites), ensuring a smooth integration between these components and resolving their dependencies in the different combinations was not straightforward. Therefore, when a software platform strategy was adopted in Scandin, stronger emphasis was given to modularity and clean interface definitions during the design process.

The requirement of maximizing reuse. In the design of the architecture, architects also needed to consider the requirement of being agnostic to the hardware platform, operating system, and other sources of variation as much as possible. As described by one interviewee:

> *"[the components] are not related to any operating system. And we chose them in a way that whatever language on the client side is used there is always the possibility to create clients for the services."*

10.6.3 Code Contribution

This became a real challenge when Scandin decided to not completely separate platform development from product development. That is, in the product-driven platform development model that was adopted, both platform teams and product teams needed to contribute to the platform. This was especially the case in situations where the platform teams could not keep up with the increased number of feature requests by product teams. In the context of this study, the company had adopted an internal open-source model where product teams could assume the responsibility of building features into certain parts of the platform in order to support their products if they did not want to wait in the queue. Other parts that were considered too risky to be open were kept closed within the platform teams. Some of the challenges associated with the internal open-source model were as follows:

Retrievability. Depending on the organizational boundaries between business units, component teams and distributed teams, the platform code were less or more difficult to find. The participants attributed this to poor visibility of the assets, poor communication between teams and business units, and the lack of standardization in source code control solutions.

Platform quality. Because the quality of the platform could be significantly affected when different teams change different parts of the platform on regular basis, Scandin had put an auditing program in place where changes were audited by a code guardian before they could take effect. One of the technical leads explains the process:

> *"Projects delivering the features [product teams] can go and modify [the platform] as long as the [platform team] audits that and makes the release based on that [audit]."*

An issue was raised by some participants regarding this model which is that with a lack of standardization of acceptance criteria, the auditing process might become a bottleneck at certain times.

Platform stability. Ideally, the impact of any change to the platform ought to be tested against all products and combinations that use the platform before it could be released. As a technical lead in Scandin explained, in order to assign this responsibility to product teams, it required a technical solution that duplicated the build environment of the platform locally on their machines so they could see its impact before submitting it for an audit.

10.6.4 Technical practices

Some technical practices that had been successfully implemented in the previously single-product-centered culture in Scandin did not scale well when the transition was being made to platform-development. The data revealed some challenges in such areas as testing, automation, continuous integration, and releases.

Testing. To ensure the stability of the platform in the liberal environment of Scandin with their open-source model, rigorous testing practices were needed. One of the challenges associated with that was to be able to populate the different product instances that had been built on top of a given platform, and test the impact of a certain change set on these instances. When this process was in place, it needed to be highly and efficiently automated in order to be effective:

> *"[teams needed to] test the functionality of the [platform] in all supported product contexts… they can automatically - before releasing products - repeat testing on all supported products and platforms."*

Another challenge that often arises when testing products that share a common platform is identifying what should be tested in the platform and what should be tested in the separate products [Engstrom2011]. Scandin was not any different in this regard. One of the problems I noticed was the diffusion of responsibility among platform teams and product teams. Some product teams assumed that platform teams should be the ones taking care of testing changes in the platform, while some platform teams made similar assumptions about product teams. When I asked a platform team member about the comprehensiveness of the test suites in a certain engine, he noted:

> *"We are partly relying on that common base of code being linked into other engines that are then again tested, and that code [in the platform] is tested in that [reuse] process."*

Continuous integration. Teams contributing to the platform needed to ensure that changes – in the most part – are agnostic to the operating system or hardware platform. Therefore, a practical build process needed to be setup in such a way so that the changes were automatically propagated to all the different relevant build environments in the organization. This required a lot of sharing and effective communication among teams and business units, and at the time of the study, it was still not achieved.

Release synchronization. One of the challenges raised by some architects had to do with managing the versions and synchronizing the releases of different components in the platform to ensure a trouble-free integration at the product level. As for different versions, sometimes the company needed to maintain older versions of components that were still in use by some of their customers. And at any given point of time, it had to be clear to the maintenance and support teams which versions of component A were supported by which versions of component B. This had to be clear also to product teams to ensure they made the transition to newer versions in time for their new product releases. One of the team lead explains why these versions existed:

> *"If we change the logic it shouldn't break the integration. And if we are going to break the integration, then it's a new version here that should be in sync… there will be several [versions] to be able to give time to the client ... to migrate and to take the latest version."*

10.7 People Challenges

Software engineering is one of the fields where the human aspect plays an essential role in the success of any practice. As evident in the collected data, making the transition to a software platform strategy is challenged by a number of factors related to individuals in the organization, namely:

10.7.1 Resisting Change

As I saw in Scandin, making the leap to a software platform strategy required major changes in the organizational culture. While some people found it easier to adapt and take the ride, others seemed to struggle for different reasons as reported by the participants such as: not seeing the value of the change, perceiving the change as inconvenient, and having to make adjustments for the new work environment:

> *"Some people are very set in their ways and these might be reluctant to change the way they work."*

There were also political factors that inhibited adopting changes proposed by others. For example, some business unit managers resisted the decision to join forces with other units to develop a certain component because for them that meant letting go of their own existing components:

> *"Convincing business unit heads to let go of their own asset which they control fully to one joint module is challenge number one."*

10.7.2 Technical Competency

The importance of the technical experience, knowledge and skills that usually play an important role in software teams is exacerbated in the context of platform development. Developers in Scandin needed to be able to cope with the complexity of the platform architecture, write cross-platform code, and contribute sound technical solutions to support testing and continuous integration in a cross-platform environment. When I asked one of the technical leads about some components that were specific to certain operating systems, he attributed that to missing skills such as writing cross-platform code:

> *"We don't really have a large experience - at least in this site office - of writing cross-platform code."*

10.7.3 Domain Knowledge

I noticed in the case of Scandin that a good understanding of the domain is vital in developing a useful and reusable platform. Without sufficient domain knowledge, engineers could not make decisions as to what was common and what was variable in a given component. This also affected decisions pertaining to the discontinuation of certain components if the impact of such decisions on the customer base was not understood.

10.8 Threats to Validity

In the study, I considered three different platforms and interviewed individuals of a wide range of roles in the company to get a holistic view of the subject matter and expand the generality of the findings. Nevertheless, analyzing a single company is in itself a threat to the generality of the findings. Moreover, some reported issues may be specific to the cultural context of Scandinavian companies or the domain of the studied company.

Furthermore, although I assumed that the participants were truthful and honest in their responses and narrations, I noticed that at times the participants were reporting their personal concerns with what might happen as opposed to what was actually happening. I mitigated this issue: 1) by trying to ask for examples and specific incidents whenever an issue was raised, and throughout this chapter I make it clear when such examples were not provided; and 2) by attending meetings and talking to people from other teams to verify certain claims.

Moreover, the validity of the findings might have been biased by my interpretations. This bias usually comes with all qualitative studies. While qualitative studies typically do not strive for statistical significance, they depend on crosschecking and triangulation of different data sources to verify the findings and draw conclusions. In this case, to mitigate this bias as much as possible, I tried to diversify the sources of information used in the analysis such as interview transcriptions, field notes taken during meetings and field visits, and other artefacts provided by the company. I also verified the findings and interpretations with a number of people in the company to check for any misunderstandings.

10.9 Comparison with the Literature

In this section, I discuss literature relevant to the findings of the study presented in this chapter, and I compare my findings with those of previous studies and highlight differences if any. The literature is abundant on success stories of adopting a software platform strategy (e.g. [Cusumano1995], [Romberg2007]). But in this section, I only focus on literature discussing challenges.

In the broader context of the topic, there is a large body of research on platform development in fields other than software engineering. Muffatto [Muffatto1999] analyzed the introduction of a platform strategy in the automobile industry and identified a raft of issues. Some issues are related to the organizational structure, namely: the need for an effective communication structure among platform teams as well as between platform teams and product innovation teams, and the issue of the collocation of platform teams. This goes hand in hand with my findings under the organizational challenges category. Another identified challenge was in regard to the derivation process of platforms from existing products which I also visited in multiple occasions in this chapter. In the manufacturing context, Sundgren [Sundgren1998] points out that the architectural reconfiguration of elements in platform development imposes a real challenge.

Moving closer to the software field, Lynex et al. [Lynex1998] identified nontechnical inhibitors of reuse adoption and suggested possible solutions. The authors mention issues such as competition amongst business units, unwillingness to share, overlapping responsibilities, quality of components, and other issues. The authors also suggested some solutions such as introducing central coordination of development. The findings of my study, however, clearly show that centralization by itself is not an effective solution, especially in flat organizations. Halman et al. [Halman2003] discussed some risks and challenges in platform-driven development, namely: integrating existing assets into the platform, the challenge of meeting the needs of all target markets, added complexity in the development process, the need to have a good understanding of the market, and the issue of the flexibility in responding to the market needs versus platform stability. While the findings of the study presented in this chapter are consistent with all of these challenges, the author argues that product families (which are based on platforms) make communication easier. In my findings, on the other hand, communication was found to be a major challenge introduced by platform thinking. This is also confirmed by Jiao et al. [Jiao2007] who highlights communication among the different players in platform development as an issue to be addressed. In the context of Hewlett-Packard, Jandourek [Jandourek1996] mentions that one of the key factors in the platform development practice is an organizational structure that supports interdependencies between platform teams and product teams. The author also addresses concerns similar to ours regarding quality criteria and test procedures, and regarding common development environments and processes which I discussed under standardization. Mili et al. [Mili1995] visits the issue of team structure in the organization and asserts that a combination of feature and component teams may be necessary. Similar to our findings regarding the downside of highly autonomous teams, Whitworth et al. [Whitworth2007] asserts that agile teams tend to become overly differentiated or isolated from the rest of the organization.

In the discussion of component-based software engineering, Crnkovic [Crnkovic2001] addresses the issue of the sensitivity of platforms to changes. Cusumano et al. [Cusumano1999] offers a thorough discussion on issues pertaining to cross-platform code such as: synchronizing code bases, keeping track of all variations, and exhaustively testing all versions. Moreover, Greenfield et al. [Greenfield2003] addresses issues such as standardization and automation in production processes. Barnes et al. [Barnes1988] provided an economic foundation for software reuse in which they mention two source control model: a pure producer-consumer model, and an open source model. In the context of my study, both models were part of the discussion under the code contribution subsection.

Some of the challenges found in the literature that I did not stumble upon include: finding a balance between the goal of maximizing reuse at one end and the goal of delivering distinctly unique products to the market to drive innovation at the other end [Sundgren1998], customer integration in platform development, and economic justification of platforms [Jiao2007].

10.10 Chapter Summary

This chapter provided a comprehensive taxonomy of the challenges that arise when organizations decide to adopt software platforms as a SPL technique. The study also reveals how new trends in software engineering - especially as agile methods, distributed development, and flat management structures interplay with the platform strategy. I used an ethnographic approach to collect data by spending time at a medium-scale company in Scandinavia. The collected data was analyzed using Grounded Theory. The findings identify four classes of challenges, namely: business challenges, organizational challenges, technical challenges, and people challenges. To the best of my knowledge, the work presented in this chapter provides the most comprehensive list of issues and challenges, both technical and nontechnical within the context of software engineering in general and modern software development in particular.

CHAPTER ELEVEN: SUMMARY AND CONCLUSIONS

11.1 Summary

This dissertation presented a research inquiry on whether it is feasible to treat variability management in a reactive as opposed to proactive manner in order to lower the adoption barrier to SPLs in agile environments. The main goal of the dissertation was to construct a framework for agile organizations to enable systematic variability management for similar software products. Towards achieving this goal, six different studies were conducted to examine the different areas of variability management and how they can be incorporated within an ASD practice. Five studies spanned over a range of variability management aspects, namely: variability elicitation for business logic requirements, variability elicitation for presentation and portability requirements, variability modeling, variability realization and product derivation. The sixth study aimed to provide an understanding of the challenges surrounding transferring the framework to an industrial context. The main contributions of this dissertation are summarized in Table 12.

Table 12 - Summary of Contributions

Contribution	**Areas**	**Method**	**Tool Support**	**Publication**
Literature Review	APLE, variability management, feature modeling, traceability	Literature Survey	NA	Distributed in different publications based on the topic [Ghanam2008], [Ghanam2008b], [Ghanam2011], [Ghanam2010], [Ghanam2010c], [Ghanam2010b], [Ghanam2009],

				[Ghanam2011b] – under review
Variability elicitation & evolution in business logic requirements	Elicitation, evolution	Exploratory study	-	[Ghanam2011]
Variability elicitation & evolution in presentation and portability requirements	Elicitation, evolution	Action research	-	[Ghanam2010]
Variability modeling	Feature modeling, traceability	Comparative evaluation, running example	Yes	[Ghanam2010c]
Variability realization	Implementation	Proof-of-concept, case study	Yes	[Ghanam2010b]
Product derivation	Extraction, instantiation	Self-evaluation	Yes	[Ghanam2009]
Transferability	Adoption issues, technical challenges, non-technical challenges	Ethnography, grounded theory	NA	[Ghanam2011b] – under review

11.2 Conclusions

The research presented throughout this dissertation provides an in-depth understanding of the feasibility of an agile framework for SPL engineering. The findings of this research show that ATs can play a valuable role in the variability elicitation process. Nonetheless, ATs alone may not be sufficient to deduce implicit constraints from requirements. This issue is addressed further in my work on leveraging EATs to extend feature models and support traceability between the model and the implementation. Using this traceability approach, hidden constraints and dependencies can be exposed. Moreover, EATs, assisted by proper tool support, enable the evolution of variability by providing instantaneous feedback on the impact of adding or removing features or variants.

Furthermore, the dissertation demonstrated examples of non-functional aspects that can be treated in a reactive rather than proactive manner. The significance of this kind of treatment lies in the ability to cope with the volatility of a given market or the instability associated with emerging technologies. The example I showed tackled the digital tabletop market which is still emerging and unpredictable.

At the implementation level, the results of my research show that realizing variability can occur in a reactive manner provided that proper refactoring and testing practices are followed. The results also illustrate how the process can be made more systematic by using tests as a common starting point to inject variability on-demand. The efficiency of the process can be improved by providing automated tool support. Once variability has been realized in the system, the dissertation shows that individual products can be built using the derivation technique or the instantiation technique.

The last study in the dissertation provides important findings on what challenges to expect when adopting the new framework in an industrial context. The findings show that there is a number of technical challenges to address, and there is also a great deal of non-technical issues related to the business needs, the organizational context, and a raft of human factors.

11.3 Future Work

A key finding I have encountered throughout this dissertation is that the need for automated tool support cannot be overstated if we were to implement a successful agile product line practice. Such tool support is needed in the different stages of the proposed framework to improve the efficiency of repetitive tasks, provide automated measures to support error-prone activities, and to help preserve and communicate the necessary knowledge to the different stakeholders. Moreover, building a reliable economic model is essential in order to evaluate the added economic value of the proposed framework in relation to other frameworks. Future research also includes taking the proposed framework to practice to conduct a longitudinal holistic evaluation in order to determine how the different components of the framework interplay. Furthermore, the transferability study provides grounds for numerous research questions to be examined more thoroughly.

References

[Agile2001]	Agile Manifesto, 2001, http://www.agilemanifesto.org, last accessed February 22, 2010.
[Agile2011]	Agile Hallmarks, VersionOne, available at: http://www.versionone.com/Agile101/Agile_Hallmarks.asp, last accessed on July 6, 2011.
[Ambler2008]	Ambler, S., Agile Adoption Rate Survey Results, 2008, http://www.ambysoft.com/surveys/agileFebruary2008.html, last accessed on March 2, 2011.
[Anastasopoulos2009]	Anastasopoulos, M., Muthig, D., An Evaluation of Aspect-Oriented Programming as a Product Line Implementation Technology, 141-156, Proceedings of the 8th International Conference, ICSR 2004, Madrid, Spain, July 5-9, 2009.
[André2010]	André, F., http://featuremodeldsl.codeplex.com/. *Feature Model DSL Homepage*, 2009. Accessed February 10, 2010.
[Andreychuk2010]	Andreychuk, D., Ghanam, Y., and Maurer, F., Adapting Existing Applications to Support New Interaction Technologies - Technical and Usability Issues. The ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS) - Late Breaking Results, Germany, 2010.
[Antón1997]	Antón, A., Goal Identification and Refinement in the Specification of Information Systems, 1997, PhD Thesis, Georgia Institute of Technology.
[Antoniol2002]	Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., Merlo, E., Recovering traceability links between code and documentation, *IEEE Transactions on Software Engineering*, vol.28, no.10, pp. 970- 983, Oct 2002.
[Apple2011]	Apple iPad, http://www.apple.com/ipad/, last accessed July 6, 2011.
[Atkinson2000]	Atkinson, C., Bayer, J., and Muthig, D. "Component-based product line development: the KobrA approach," in 1st Int'l Software Product Line Conference Denver, Colorado, United States: Kluwer Academic Publishers, 2000, pp. 289-309.

[Babar2009]	Babar, M., Ihme, T., and Pikkarainen, M., An industrial case of exploiting product line architectures in agile software development, 2009, proceedings of the 13th International Software Product Line Conference (SPLC '09). Carnegie Mellon University, Pittsburgh, PA, USA, 171-179.
[Barnes1988]	Barnes, B., Durek, T., Gaffney, J., and Pyster, A., A framework and economic foundation for software reuse, Software Reuse: Emerging Technology (1988), 77-88.
[Beck2003]	Beck, K., Test-Driven Development: By Example. Addison Wesley, 2003.
[Beck2004]	Beck, K., and Andres, C. (2004) Extreme Programming Explained: Embrace Change (2nd Edition). Addison-Wesley Professional.
[Berg2005]	Berg, K., Bishop, J., and Muthig, D., "Tracing Software Product Line Variability — From Problem to Solution Space," presented at *2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*, White River, South Africa, 2005.
[Bergey2004]	Bergey, J., Cohen, S., Jones, L., Smith, D., Software Product Lines: Experiences from the Sixth DoD Software Product Line Workshop. Technical report CMU/SEI-2004-TN-011, 2004.
[Beuche2011]	Beuche, D., What's the difference? A Closer Look at Configuration Management for Product Lines, 2010. Available at http://productlines.wordpress.com/2010/03/13/whats-the-difference-a-closer-look-at-configuration-management-for-product-lines/, last accessed on September 8, 2011.
[Bieman1995]	Bieman, J., and Kang, B., Cohesion and reuse in an object-oriented system. *SIGSOFT Software Engineering Notes,* 20, 259-262, August, 1995.
[Bittner2006]	Bittner, K., and Spence, I. (2006) "Managing Iterative Software Development Projects," Addison Wesley Professional.

[Boehm1976] Boehm, B., Brown, J., and Lipow, M., 1976. Quantitative evaluation of software quality. In Proceedings of the 2nd international conference on Software engineering (ICSE '76). IEEE Computer Society Press, Los Alamitos, CA, USA, 592-605.

[Bosch2000] Bosch, J., Hogstrom, M., 2000. Product instantiation in software product lines: a case study. In: Second International Symposium on Generative and Component-Based Software Engineering, pp. 147–162.

[Brownsword1996] Brownsword, L., Clements, P., A Case Study in Successful Product Line Development. Technical report CMU/SEI-96-TR-016, 1996.

[Buddi2011] Buddi, http://sourceforge.net/projects/buddi, last accessed July 6, 2011.

[Bühne2004] Bühne, S., Lauenroth, K., and Pohl, K., "Why is it not Sufficient to Model Requirements Variability with Feature Models?" Proceedings of AURE`04, 2004, Japan, pp. 5-12.

[Buhrdorf2003] Buhrdorf, R., Churchett, D., and Krueger, C., Salion's Experience with a Reactive Software Product Line Approach, Revised Papers of the 5th International Workshop, PFE 2003, Siena, Italy, November 4-6, 2003.

[CaliberRM2010] CaliberRM, http://www.borland.com/us/products/caliber/index.html, accessed March 1, 2010.

[Carbon2006] Carbon, R., Lindvall, M., Muthig, D., Costa, P.: Integrating product line engineering and agile methods: Flexible design up-front vs. incremental design. In: APLE '06: 1st International Workshop on Agile Product Line Engineering (In conjunction with SPLC 2006).

[Carbon2008] Carbon, R., Knodel, J., Muthig, D., Meier, G.: Providing feedback from application to family engineering – the product line planning game at the Testo AG. In: SPLC '08: Proceedings of the 12th International Software Product Line Conference, IEEE Computer Society (2008) 180-189.

[Chen2009]	Chen, L., M. A. Babar, et al. (2009). Variability Management in Software Product Lines: A Systematic Review. 13th International Software Product Line Conference, San Francisco, CA, USA.
[Chen2009b]	Chen, L., Ali Babar, M.: A Survey of Scalability Aspects of Variability Modeling Approaches. In: Workshop on Scalable Modeling Techniques for Software Product Lines at SPLC 2009, San Francisco, CA, USA (2009).
[Cho2008]	Cho, H., Lee, K., and Kang, K. C. 2008. Feature Relation and Dependency Management: An Aspect-Oriented Approach. *Proceedings of the 2008 12th international Software Product Line Conference* (2008). IEEE Computer Society, Washington, DC, 3-11.
[Chung1999]	Chung, L., Nixon, B.A., Yu, E., Mylopoulos, J.: Non-Functional Requirements in Software Engineering. International Series in Software Engineering, vol. 5, p. 476. Springer, Heidelberg (1999).
[Chung2009]	Chung, L., and do Prado Leite, J., On Non-Functional Requirements in Software Engineering, Conceptual Modeling: Foundations and Applications (LNCS), 2009, pp. 363-379.
[Clegg2002]	Clegg, K., Kelly, T., and McDermid, J., Incremental Product-Line Development, International Workshop on Product Line Engineering, Seattle, 2002.
[Cleland-Huang2004]	Cleland-Huang, J., Zemont, G., and Lukasik, W. 2004. A Heterogeneous Solution for Improving the Return on Investment of Requirements Traceability. In *Proceedings of the Requirements Engineering Conference, 12th IEEE international*(September 06 - 10, 2004). RE. IEEE Computer Society, Washington, DC, 230-239.
[Clements2001]	Clements, P., and Northrop, L., Software Product Lines: Practice and Patterns, Addison-Wesley, US, 2001.
[Clements2002]	Clements, P. and Northrop, L. (2002) Salion, inc.: A software product line case study. Software Engineering Institute, Carnegie Mellon University.
[Cohn2004]	Cohn, M., User Stories Applied: For Agile Software Development: Addison-Wesley, 2004.

[Cooper2006]	Cooper, K., and Franch, X., APLE: 1st International Workshop on Agile Product Line Engineering, International Conference on Software Product Line Engineering, 2006.
[Coplien1999]	Coplien, J., Hoffman, D., and Weiss, D., “Commonality and Variability in Software Engineering,” IEEE Software, 1999, pp. 37-45.
[Crnkovic2001]	Crnkovic, I., Component-based software engineering – new challenges in software development, Software Focus (2001), 2(4), 127- 133.
[Cunnigham2011]	Cunnigham, W. FIT: Framework for Integrated Test. Available at: http://fit.c2.com. Last accessed on June 13, 2011.
[Cusumano1995]	Cusumano, M., Selby, R., Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People, New York: Free Press, 1995.
[Cusumano1999]	Cusumano, M., Yoffie, D., What Netscape learned from cross-platform software development. Communications of the ACM (1999), 42(10), 72-78.
[Deelstra2005]	Deelstra, S., Sinnema, M., and Bosch, J., 2005, Product derivation in software product families: a case study, The Journal of Systems and Software, 74, pp. 173–194.
[Dittrich2007]	Dittrich, Y., John, M., Singer, J., and Tessem, B., For the Special issue on Qualitative Software Engineering Research, Information and Software Technology (2007), 49 (6), 531-539.
[DOORS2010]	DOORS, http://www-01.ibm.com/software/awdtools/doors/, accessed March 1, 2010.
[DViT2009]	DViT Technology, available at: http://smarttech.com/DViT, last accessed June 18, 2009
[Engstrom2011]	Engstrom, E., Runeson, P., Software product line testing - A systematic mapping study, Information and Software Technology (2011), 53(1), 2-13.
[Eriksson2005]	Eriksson, M., Börstler, J., and Borg, K., The PLUSS Approach - Domain Modeling with Features, Use Cases and Use Case Realizations, *In Proceedings of the International Conference on Software Product Lines*, 33-44, 2005.

[Feng2007] Feng, K., Lempert, M., Tang, Y., Tian, K., Cooper, K., Franch, X.: Developing a survey to collect expertise in agile product line requirements engineering. In: RWASE'07: International Research-in-Progress Workshop on Agile Software Engineering.

[Fey2002] Fey, D., Fajta, R., and Boros, A., Feature Modeling: A Meta-Model to Enhance Usability and Usefulness. In *Software Product Lines (SPLC2)*: Spri*nger, 2002, pp. 198-21.*

[Filho2002] Filho, I., Oliveira, T., Lucena, C., 2002. A proposal for the incorporation of the features model into the UML language. Proceedings *of the 4th International Conference on Enterprise Information Systems* (ICEIS2002), Ciudad Real, Spain.

[FIT2010] FIT, http://fit.c2.com, accessed Nov, accessed March 1, 2010.

[FitNesse2011] FitNesse, http://www.fitnesse.org/, last accessed July 6, 2011.

[Fowler2004] Fowler, M., and Beck, K. (2004). Refactoring: improving the design of existing code. Addison Wesley.

[Frakes1994] Frakes, W., and Pole, T., An empirical study of representation methods for reusable software components, *IEEE Transactions on Software Engineering*, 20(8), 617-630, 1994.

[Frakes1995] Frakes, W., Fox, C., Sixteen Questions about Software Reuse, Communications of the ACM, 38(6), 75-87, 1995.

[Fuchs1992] Fuchs, N., Specifications are (Preferably) Executable. IEE/BCS Software Engineering Journal 7(5) (1992) 323–334.

[Fuchs1992] Fuchs, N., Specifications are (Preferably) Executable. IEE/BCS Software Engineering Journal 7(5) (1992) 323–334.

[Gacek2001] Gacek, C. and Anastasopoules, M., 2001. Implementing product line variabilities. SIGSOFT Software Engineering Notes 26, 3 (May. 2001), 109-117.

[Gaffney1989] Gaffney, J., and Durek, T., Software reuse-key to enhanced productivity: some quantitative models, *Information and Software Technology*, 31(5), 258–267, 1989.

[Gamma1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995) "Design Patterns: Elements of Reusable Object-Oriented Software," Addison-Wesley.

[Ghanam2008] Ghanam, Y., and Maurer, F., An Iterative Model for Agile Product Line Engineering. The SPLC Doctoral Symposium, 2008 - in conjunction with the 12th International Software Product Line Conference (SPLC 2008), Limerick, Ireland.

[Ghanam2008b] Ghanam, Y., Park, S., and Maurer, F., A Test-Driven Approach to Establishing & Managing Agile Product Lines. The 5th Software Product Lines Testing Workshop (SPLiT 2008) in conjunction with SPLC 2008, Limerick, Ireland.

[Ghanam2009] Ghanam, Y., and Maurer, F., Extreme Product Line Engineering: Managing Variability & Traceability via Executable Specifications. Agile Conference 2009, Chicago.

[Ghanam2009b] Ghanam, Y., Maurer, F., Abrahamsson, P., and Cooper, K., A Report on the XP Workshop on Agile Product Line Engineering. SIGSOFT Software Engineering Notes 34, 5 (Oct. 2009), 25-27.

[Ghanam2010] Ghanam, Y., Andreychuk, D., and Maurer, F., Reactive Variability Management Using Agile Software Development. Proc. of the international conference on Agile methods in software development (Agile 2010), Orlando, USA, 2010.

[Ghanam2010b] Ghanam, Y., and Maurer, F., Extreme Product Line Engineering – Refactoring for Variability: A Test-Driven Approach. The 11th International Conference on Agile Processes and eXtreme Programming (XP 2010), Trondheim, Norway, 2010.

[Ghanam2010c] Ghanam, Y., and Maurer, F., Linking Feature Models to Code Artifacts using Executable Specifications. Proc. of the 14th International Software Product Line Conference (SPLC 2010), Jeju Island, South Korea, 2010.

[Ghanam2010d] Ghanam, Y., Cooper, K., and Maurer, F., The Second XP Workshop on Agile Product Line Engineering. In conjunction with the 11th International Conference on Agile Processes and eXtreme Programming (XP 2010), Trondheim, Norway.

[Ghanam2011] Ghanam, Y., and Maurer, F., Using Acceptance Tests for Incremental Elicitation of Variability in Requirements: An Observational Study. Proc. of the international conference on Agile methods in software development (Agile 2011), Salt Lake City.

[Ghanam2011b]	Ghanam, Y., Maurer, F., and Abrahamsson, P., Making the leap to software platforms. Journal submission under review.
[Glinz2007]	Glinz, M., On Non-Functional Requirements, 15th IEEE International Requirements Engineering Conference, 2007, pp.21-26.
[Gotel1994]	Gotel, O., and Finklestein, A., An analysis of the requirements traceability problem, Proceedings of ICRE94, 1st International Conference on Requirements Engineering, Colorado Springs, Co, IEEE CS Press, 1994.
[Gotel1994]	Gotel, O., and Finkelstein, A., An Analysis of the Requirements Traceability Problem, *1st International Conference on Requirements Eng*., 1994, pp. 94-101.
[Greenfield2003]	Greenfield, J., Short, K., Software factories: assembling applications with patterns, models, frameworks and tools, Companion of OOPSLA (2003).
[GreenPepper2010]	GreenPepper, http://www.greenpeppersoftware.com, accessed March 1, 2010.
[Gurp2006]	Gurp, J., and Prehofer, C., 2006, Version management tools as a basis for integrating product derivation and software product families, Proceedings of the Workshop on Variability Management at SPLC 2006, pp. 48–58.
[Halman2003]	Halman, J., Hofer, A., Van Vuuren, W., Platform-driven development of product families: Linking theory with practice, Journal of Product Innovation Management (2003), 20(2), 149-162.
[Halmans2003]	Halmans, G., and Pohl, K., Communicating the variability of a software-product family to customers, Software System Model, 2003, vol. 2, pp. 15–36.
[Hammersley1983]	Hammersley, M., Atkinson, P., Ethnography: principles in practice, Tavistock - London, 1983.
[Hanssen2008]	Hanssen, G., Faegri, T., Process fusion: An industrial case study on agile software product line engineering, Journal of Systems and Software, Volume 81, Issue 6, Agile Product Line Engineering, June 2008, Pages 843-854.

[Heineman2001] Heineman, G., and Councill, W., *Component-based Software Engineering, Putting the Pieces Together*. Addison-Wesley, 2001.

[Henninger1997] Henninger, S., An evolutionary approach to constructing effective software reuse repositories. *ACM Transaction on Software Engineering,* 6(2), 111-140, 1997.

[Highsmith2001] Highsmith, J., Cockburn, A., Agile Software Development: The Business of Innovation, Computer, vol. 34, no. 9, pp. 120-127, Sept. 2001.

[HP2009] HP TouchSmart IQ770 PC datasheet, available at: http://www.hp.com/hpinfo/newsroom/press_kits/2007/ces/ds_pc_touchsmart.pdf, last accessed June 18, 2009.

[Huang2006] Huang, J., Just enough requirement traceability, *Proceedings of the 30th Annual International Computer Software and Applications*, Chicago, September 2006, pp. 41– 42.

[Jacobson1997] Jacobson, I., Griss, M., and Johnson, P., 1997, Software Reuse: Architecture, Process and Organization for Business success, Addison-Wesley.

[Jandourek1996] Jandourek, E., A model for platform development - HP's software development strategy - Company Operations, Hewlett-Packard Journal (1996), available at: http://www.hpl.hp.com/hpjournal/96aug/aug96a6.pdf, last accessed on March 2, 2011.

[Jandourek1996] Jandourek, E., A model for platform development - HP's software development strategy - Company Operations, Hewlett-Packard Journal (1996), available at: http://www.hpl.hp.com/hpjournal/96aug/aug96a6.pdf, last accessed on March 2, 2011.

[Jianhong2006] Jianhong, M., Runhua, T., Handling Variability in Mass Customization of Software Family Product, Knowledge Enterprise: Intelligent Strategies in Product Design, Manufacturing, and Management (2006), 207, 996-1001.

[Jiao2007] Jiao, J., Simpson, T., and Siddique, Z., Product family design and platform-based product development: a state-of-the-art review, Journal of Intelligent Manufacturing (2007), 18(1), 5-29.

[Joyce1988] Joyce, E., Reusable software: passage to productivity, *Datamation,* 34 (18), 97–102, 1988.

[Kakarontzas2008] Kakarontzas, G., Stamelos, I., and Katsaros, P.: Product line variability with elastic components and test-driven development. In: CIMCA ’08: Proceedings of the International Conference on Computational Intelligence for Modelling Control & Automation, IEEE Computer Society (2008) 146-151.

[Kaner2003] Kaner, C., Cem Kaner on Scenario Testing: The Power of ‘What-If…’ and Nine Ways to Fuel Your Imagination, *Better Software*, *5(5)*:16–22, 2003.

[Kang1990] Kang, K., Cohen, S., Hess, J., Novak, W., and Peterson, A., Feature-Oriented Domain Analysis (FODA) Feasibility Study, SEI Technical Report 1990.

[Kang1998] Kang, K., Kim, S., Lee, J., Kim, K., Shin, E., and Huh, M., FORM: A feature-oriented reuse method with domain specific reference architectures, *Annals of Software Engineering,* vol. 5, pp. 143-168, 1998.

[Kerievsky2010] Kerievsky, J., Storytesting, http://industrialxp.org/ storytesting.html, accessed March 1, 2010.

[Kruchten2004] Kruchten, P., 2004, Going Over the Waterfall with the RUP, available at: http://www.ibm.com/developerworks/rational/library/4626.html#N100AF

[Kruger2002] Kruger, C., Easing the Transition to Software Mass Customization, in Proceedings of the 4th International Workshop on Product Family Engineering, Germany, 2002.

[Kurmann2006] Kurmann, R., Agile SPL-SCM Agile Software Product Line Configuration and Release Management. In: APLE ’06: 1st International Workshop on Agile Product Line Engineering (In conjunction with SPLC, 2006).

[Larman2003] Larman, C., Basili, V., Iterative and Incremental Development: A Brief History, IEEE Software, 2003.

[Larman2009]	Larman, C., Vodde, B., Scaling Lean & Agile Development: Thinking and Organizational Tools for Large-Scale Scrum, Addison-Wesley, 2009.
[Lau2004]	Lau, K., 2004, Component-based software development: case studies, World Scientific Publishing.
[Leffingwell2007]	Leffingwell, D., Scaling Software Agility: Best Practices for Large Enterprises, Addison-Wesley Professional, 2007.
[Li2006]	Li, Z., Lu, Myagmar, S., Zhou, Y., CP-Miner: finding copy-paste and related bugs in large-scale software code. IEEE Transactions on Software Engineering (2006), 32(3), 176-192.
[Linden2007]	Linden, F., Schmid, K., Rommes, E., Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering, Springer-Verlag New York, Inc., Secaucus, NJ, USA (2007).
[Lynex1998]	Lynex, A., Layzell, P., Organisational considerations for software reuse. Annals of Software Engineering (1998), 5, 105-124.
[MacManus2007]	MacManus, R., Facebook Grows Up - An Analysis of Today's News, 2007, http://www.readwriteweb.com/archives/facebook_grows_up.php, last accessed July 6, 2011.
[Marchetto2010]	Marchetto, A., http://selab.fbk.eu/swat/slide/2_Fitnesse.ppt, accessed March 10, 2010.
[Martin2002]	Martin, R., Agile Software Development, Principles, Patterns and Practices, Prentice Hall, 2002.
[McGrath1995]	McGrath, M., Product Strategy for High-Technology Companies, IL: Irwin, 1995.
[McGregor2008]	McGregor, J., Agile software product lines - a working session. In: SPLC ’08: Proceedings of 12th International Software Product Line Conference, IEEE Computer Society (2008) 364-364.
[Melnik2004]	Melnik, G., Read, K., and Maurer, F., Suitability of FIT User Acceptance Tests for Specifying Functional Requirements: Developer Perspective, Extreme Programming and Agile Methods - XP/Agile Universe, 2004, pp. 638-663.

[Melnik2006]	Melnik, G., Maurer, F., and Chiasson, M., "Executable Acceptance Tests for Communicating Business Requirements: Customer Perspective," *Proc. Agile 2006 Conf.,* IEEE CS Press, 2006, pp. 35–46.
[Melnik2007]	Melnik, G., Jeffries, R., Test-Driven Development – The Art of Fearless Programming, IEEE Software, 24(3): 24-30, 2007.
[Melnik2007b]	Melnik, G., Empirical Analysis of Executable Acceptance Test Driven Development, Ph.D. Thesis, University of Calgary, Department of Computer Science, Aug 2007.
[Microsoft2011]	Microsoft Surface, http://www.microsoft.com/surface, last accessed July 6, 2011.
[Mili1995]	Mili, H., Mili, F., and Mili, A., Reusing software: issues and research directions. IEEE Transactions on Software Engineering, 21(6), 1995, pp. 528–562.
[Mohagheghi2004]	Mohagheghi, P., The Impact of Software Reuse and Incremental Development on the Quality of Large Systems, *Doctoral Thesis*, Department of Computer and Information Science, Norwegian University of Science and Technology, 2004.
[Mohan2010]	Mohan, K., Ramesh, B., and Sugumaran, V. (2010). Integrating software product line engineering and agile development. IEEE Software, 27(3), 48–55.
[Morisio2002]	Morisio, M., and Torchiano, M., Definition and Classification of COTS: A Proposal, *In Proceedings of the First International Conference on COTS-Based Software Systems*, 165-175, 2002.
[Muffatto1999]	Muffatto, M., Introducing a platform strategy in product development. International Journal of Production Economics (1999), 145-153.
[Navarrete2006]	Navarrete, F., Botella, P., and Franch, X. (2006). An approach to reconcile the agile and cmmi contexts in product line development. In APLE ’06: Procceedings of the 1st International Workshop on Agile Product Line Engineering in conjunction with SPLC 2006, Baltimore, Maryland, USA.

[Noor2008]	Noor, M., Rabiser, R., Grünbacher, P., Agile product line planning: A collaborative approach and a case study, Journal of Systems and Software, Volume 81, Issue 6, Agile Product Line Engineering, June 2008, Pages 868-882.
[O'Brien1998]	O'Brien, R, An Overview of the Methodological Approach of Action Research, 1998, available at: http://www.web.net/ ~robrien/papers/arfinal.html, last accessed on July 18, 2011.
[O'Leary2007]	O'Leary, P., Babar, M., Thiel, S., and Richardson, I., Product Derivation Process and Agile Approaches: Exploring the Integration Potential, Proceedings of 2nd IFIP Central and East European Conference on Software Engineering Techniques, Poznań, Poland, 2007, pp. P. 166--171.
[O'Leary2010]	O'Leary, P. McCaffery, F., Thiel, S., and Richardson, I. (2010). An Agile process model for product derivation in software product line engineering. Journal of Software Maintenance and Evolution: Research and Practice. John Wiley & Sons, Ltd.
[Paige2006]	Paige, R., Xiaochen, W., Stephenson, Z., and Phillip J., "Towards an Agile Process for Building Software Product Lines", LNCS: XP 2006, pp. 198 – 199.
[Park2008]	Park, S.S., Maurer, F.: The benefits and challenges of executable acceptance testing. In: APOS 2008: Proceedings of the 2008 international workshop on Scrutinizing agile practices or shoot-out at the agile corral, pp. 19–22 (2008).
[Parnas1976]	Parnas, D., On the Design and Development of Program Families, *IEEE Trans. Software Engineering*, 2(1), 1-9, 1976.
[Parra2009]	Parra, C., Blanc, X., Duchien, L.: Context Awareness for Dynamic Service-Oriented Product Lines. *Proceedings of 13th International Software Product Line Conference* (SPLC), San Francisco, CA, USA (2009).
[Pashov2004]	Pashov, I., Feature Based Method for Supporting Architecture Refactoring and Maintenance of Long-Life Software Systems. *PhD Thesis*, Technical University Ilmenau, 2004.
[Perry2000]	Perry, W. *Effective Methods for Software Testing, 2/e*, John Wiley & Sons: New York, NY, 2000.

[PLD2011] PLD, https://fitclipse.svn.sourceforge.net/svnroot/fitclipse /trunk/ProductLineDesigner, last accessed July 6, 2011.

[Pohl2005] Pohl, K., Böckle, G., and Linden, F., Software Product Line Engineering: Foundations, Principles and Techniques, Springer, Germany, 2005.

[Prieto-Díaz1996] Prieto-Díaz, R., Reuse as a New Paradigm for Software Development, In *Proceedings of the International Workshop on Systematic Reuse*, London, 1996.

[Pure::Systems2010] Pure::Systems, http://www.puresystems.com/ DOORS.102+M54a708de802.0.html, accessed March 1, 2010.

[Raatikainen2008] Raatikainen, M., Rautiainen, K., Myllärniemi, V., Männistö, T.: Integrating product family modeling with development management in agile methods. In: SDG '08: Proceedings of the 1st international workshop on Software Development Governance, (2008) 17-20.

[Ramesh2001] Ramesh, B., Jarke, M., Toward Reference Models for Requirements Traceability. *IEEE Transactions on Software Engineering*, vol. 27, Issue 1 (January 2001) pp. 58 – 93.

[Reppert2004] Reppert, T., "Don't Just Break Software, Make Software: How Story-Test-Driven-Development is Changing the Way QA, Customers, and Developers Work", Better Software, 6(6): 18–23, 2004.

[Riebisch2003] Riebisch, M., Towards a more precise definition of feature models. Position Paper, in: M. Riebisch, J.O. Coplien, D, Streitferdt (Eds.), Modelling Variability for Object-Oriented Product Lines, 2003.

[Riebisch2004] Riebisch, M., Supporting Evolutionary Development by Feature Models and Traceability Links. *Proceedings of the 11th IEEE international Conference and Workshop on Engineering of Computer-Based Systems* (May 24 - 27, 2004). ECBS. IEEE Computer Society, Washington, DC, 370.

[Riegger2010] Riegger, F., Test-based Feature Management for Agile Product Lines, Diploma Thesis (conducted at ASE Group), HS Mannheim, Feb 2010.

[Romberg2007]	Romberg, T., Software platforms – How to win the peace, The 40th Annual Hawaii International Conference on System Sciences (2007).
[Roschelle1996]	Roschelle, J., and Kaput, J., Educational software architecture and systemic impact: The promise of component software, *Journal of Educational Computing Research*, 14(3), 217-228, 1996.
[Royce1970]	Royce, W., "Managing the Development of Large Software System." Proceedings of IEEE WESCON (August 1970), pp.1-9.
[Salbinger2010]	Salbinger, S., Product Line Designer – A Refactoring Tool for Extreme Product Lines, 2010, available at: http://ase.cpsc.ucalgary.ca/uploads/Publications/PLD.pdf
[Schmid2002]	Schmid, K., Verlage, M., "The Economic Impact of Product Line Adoption and Evolution," IEEE Software, vol. 19, no. 4, pp. 50-57, July/August, 2002.
[Schwaber2004]	Schwaber, K., Agile Project Management with Scrum, Microsoft Press - Redmond, 2004.
[Shalloway2009]	Shalloway, A., Beaver, G., and Trott, J., Lean-agile software development: achieving enterprise agility. Addison-Wesley, Upper Saddle River, NJ, 2009.
[Sharp2000]	Sharp, D., Donohoe, P., Reducing Avionics Software Cost Through Component Based Product Line Development," Proceedings SPLC1, Kluwer Academic Publishers, 2000.
[Sharp2000b]	Sharp, D., "Containing and Facilitating Change Via Object Oriented Tailoring Techniques," to appear in Proceedings of The First Software Product Line Conference Denver, Colorado, August, 2000.
[Sillito2007]	Sillito, J., and Wynn, E., The Social Context of Software Maintenance, In Proceedings of the International Conference on Software Maintenance, 2007.
[SMART2009]	SMART Table datasheet, available at: www2.smarttech.com/st/en-US/Products/SMART+Table, last accessed June 12, 2009.

[SMART2011]	SMART Table, http://smarttech.com/ca/Solutions/Education+Solutions/-Products+for+education/Complementary+hardware+products/SMART+Table, last accessed July 6, 2011.
[Sommerville1985]	Sommerville, I., *Software Engineering*, Addison-Wesley, 1985.
[Strauss1997]	Strauss, A., Corbin, J., Grounded Theory in Practice, London, 1997.
[Sundgren1998]	Sundgren, N., Product Platform Development, Managerial Issues in Manufacturing Firms (1998), Chalmers University of Technology.
[Susman1983]	Susman, G., Action Research: A Sociotechnical Systems Perspective, Ed. G. Morgan. London: Sage Publications, 1983, pp. 95-113.
[Taborda2004]	Taborda, L., Generalized Release Planning for Product-Line Architecture. In: Proceedings of Software Product Line Conference. Volume 3054 of LNCS, Springer-Verlag (2004) 238-254.
[Thao2008]	Thao, C., Munson, E., and Nguyen, T., Software Configuration Management for Product Derivation in Software Product Families, the 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems, 2008, pp. 265-274.
[Tracz1987]	Tracz, W., 1987. Software reuse: motivators and inhibitors, *Proceedings of the Thirty Second IEEE Computer Society International Conference*, 358–363, 1987.
[Trinidad2008]	Trinidad, P., Benavides, D., Durán A. Ruiz-Cortés, A., and Toro, M., Automated error analysis for the agilization of feature modeling. Journal of Systems and Software, 81(6):883–896, 2008.
[Tseng2001]	Tseng, M., Jiao, J., 2001, Mass Customization, in: Handbook of Industrial Engineering, Technology and Operation Management (3rd ed.). New York, NY: Wiley. ISBN 0-471-33057-4.

[Tun2009]	Tun, T., Boucher, Q., Classen, A., Hubaux, A., Heymans, P., 2009, Relating Requirements and Feature Configurations: A Systematic Approach, *International Software Product Line Conference* (SPLC 2009).
[vanGurp2001]	van Gurp, J., Bosch, J., Svahnberg, M., On the Notion of Variability in Software Product Lines, The Working IEEE/IFIP Conference on Software Architecture (2001), 45–54.
[Vartiainen2002]	Vartiainen, P., On the Principles of Comparative Evaluation, *Evaluation*, July 2002, vol. 8, no. 3, pp. 359-371.
[VersionOne2011]	VersionOne, http://www.versionone.com/Resources/FeatureEstimation.asp, last accessed July 6, 2011.
[Whitworth2007]	Whitworth, E., Biddle, R., The Social Nature of Agile Teams, Agile 2007, pp.26-36.

APPENDIX A. ETHICS BOARD CERTIFICATE

APPENDIX B. CO-AUTHOR PERMISSIONS

APPENDIX C. EXPLORATORY STUDY

Setup

Step 1. Pre-questionnaire

Pre-Questionnaire: Please provide the following information.

1. Program of study:
2. Area of research (if any):
3. How many years did you spend in industry?
4. What area(s) did you work on in industry? (ignore if your answer to 3 was “0”)
5. How do you describe your background in Software Engineering?

Expert/Professional	Advanced	Basic	Novice	I know nothing

6. How familiar are you with Agile Software Engineering (ASE)?

I have been heavily involved in ASE research or practice	I practiced ASE once or twice	I have a good theoretical understanding of ASE but never used it	I have a general idea of what ASE is about	What is that?

7. How familiar are you with Acceptance Testing (AT)?

I have been heavily involved in AT research or practice	I used it once or twice	I have a good theoretical understanding of AT but never used it	I have a general idea of what AT is about	What is that?

8. How familiar are you with Software Product Line Engineering (SPLE)?

I have been heavily involved in SPLE research or practice	I practiced SPLE once or twice	I have a good theoretical understanding of SPLE but never practiced it	I have a general idea of what SPLE is about	What is that?

Step 2. Tutorial

The objective of this tutorial is to provide you with the background required to proceed with this experiment. You can refer back to this tutorial at any point during the course of the experiment. Please do ask questions about this tutorial should you need any further clarification. Traditionally in software engineering, the requirements of a software system are described in a textual format like this one:

1.1 The system shall allow the user to add a course, given that:
1.1.1 The user has the privilege to do so (i.e. the user is a student).
1.1.2 The student has the pre-requisites for the course to be added.
1.1.3 The student is not already enrolled in the course
1.1.4 The student has not previously taken the course
1.2 The system shall allow the user to schedule an automatic addition of a course on a specific day given that all requirements in 1.1 are met.
1.3 The system shall allow the user to drop a course he is currently enrolled in as long as classes in the course have not yet started.

In Agile Software Development, user requirements are usually NOT documented in the style shown above. A rather more structured format is used to describe features. Take this as an example:

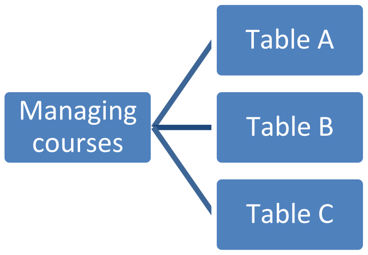
Default	Table A: Manual addition of a course				
Check	User	120356	Is registered for	CPSC433	False
Check	User	120356	Is student	True	
Check	User	120356	Has taken	CPSC433	False
Check	User	120356	Has taken	CPSC333	True
Add course	CPSC433	For user	120356		
Check	User	120356	Is registered for	CPSC433	True

Optional	Table B: Scheduled addition of a course					
Enter	User	120356	Should be registered for	CPSC433	on	23-11-2008
Check	User	120356	Is student	True		
Check	User	120356	Has taken	CPSC433	False	
Check	User	120356	Has taken	CPSC333	True	
Enter	Date	22-11-2008				
Check	User	120356	Is registered for	CPSC433	False	
Enter	Date	23-11-2008				
Check	User	120356	Is registered for	CPSC433	True	

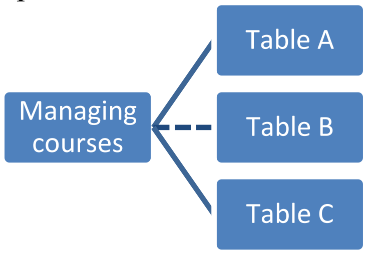
Default	Table C: Drop a course given classes have not started				
Check	User	120356	Is registered for	CPSC433	True
Check	Start date of classes in course	CPSC433	Is	> Today	True
Drop course	CPSC433	For user	120356		
Check	User	120356	Is registered for	CPSC433	False

Suppose that the two test tables shown above represent the UofC requirements of a certain feature in the system called "**Managing courses**." In this experiment, you will be given similar scenarios and asked to:

1. Draw a feature tree representing the specific instance of this feature as requested by the customer.
   This question can be answered by looking at what test tables a given customer has specified.
   In this example, we have three tables: "**Table A**", "**Table B**" and "**Table C**". Therefore, we draw:



2. Draw a feature tree representing the current state of the feature in the product line context. If required, add any [min..max] constraints.
   This question is asking you to draw the feature as it will be shown to the next customer of the same system. To answer this question, we should consider requests by all previous customers of the feature. For now, this is our first customer; this is why this drawing should look similar to the previous one. However, notice the dotted line used to convey that Table B is "**optional**" meaning that it can be removed without substantially affecting the value of the feature at hand. You can find out whether a test table is "**optional**" or "**default**" by looking at the top left corner of the table.



Now say that UofA wants the same feature as previously shown, but with the following changes:

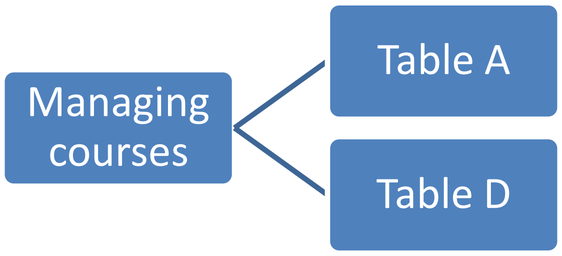
Don't select Table B: because it contradicts the bylaws of the registrar's office.
Add Table D:

Default	Table D: Drop a course even if classes have started, but with a penalty of a W grade					
Check	User	120356	Is registered for	CPSC433	True	
Drop course	CPSC433	For user	120356			
Check	Grade of	CPSC43 3	For user	120356	W	True
Check	User	120356	Is registered for	CPSC433	False	

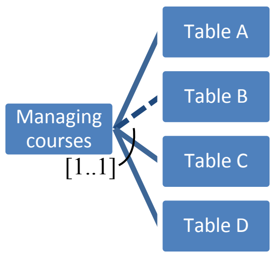
Remove Table C: as it cannot coexist with Table D.

Given the new customer request, let's answer the same two questions again.

1. Draw a feature tree representing the specific instance of this feature as requested by the customer.



2. Draw a feature tree representing the current state of the feature in the product line context. If required, add any [min..max] constraints.
   Once again, remember to include test tables requested by all customers. Again, the objective of this diagram is to show your future customers what you can offer in a certain feature.
   We start with the diagram we built in the previous scenario. Then we add to that any newly added tables. In this case, we add "**Table D**" as a "**default**" component. Other components should not change. However, to denote that Table D and Table C cannot coexist, we impose a [min..max] constraints of [1..1]. This means one and only one of these two tables has to exist in the "**Managing courses**" feature.



Suppose UofS wants the same feature with the following customization request:

Don't select Table B
Select Table D
Add Table E:

It is often sufficient to read the first row of the table to have an idea of what the table is testing (this is called user story).

Optional	Table E: Trade courses between students						
Enter	User	120356	Allows trade of	CPSC433	YES		
Enter	User	235235	Allows trade of	CPSC410	YES		
Trade	CPSC433	by user	120356	With	CPSC410	by user	235235
Check	User	120356	Is registered for	CPSC433	False		
Check	User	120356	Is registered for	CPSC410	True		
Check	User	235235	Is registered for	CPSC410	False		
Check	User	235235	Is registered for	CPSC433	True		

Try to answer these questions and check your answers with the experimenter:

1. Draw a feature tree representing the specific instance of this feature as requested by the customer.
2. Draw a feature tree representing the current state of the feature in the product line context. If required, add any [min..max] constraints.

Step 3. Exercises (three sections)

Section I

Part 1

In a feature called "Door Access Control", the customer requests that an authentication mechanism control entry to the house. This is how the initial test page looked like:

Assume: resident name: John Smith PIN: 12345

Default	Table A. Authentication through keypad input			
Enter	Resident name	John Smith	PIN	12345
Check	Door is unlocked	True		

Optional	Table B. Input locked for 2 minutes after two failed attempts			
Enter	Resident name	John Smith	PIN	11111
Check	Door is unlocked	False		
Enter	Resident name	John Smith	PIN	22222
Check	Door is unlocked	False		
Check	Input locked	True		

1. Draw a feature tree representing the specific instance of this feature as requested by the customer.

2. Draw a feature tree representing the current state of the feature in the product line context. If required, add any [min..max] constraints.

Part 2

A new customer requests the "Door Access Control" feature, but requires the following changes:

Don't select Table B: Customer does not want to restrict number of attempts.
Add Table C

Optional	Table C. A successful attempt after a failed one should prompt the user to enter his info again			
Enter	Resident name	John Smith	PIN	11111
Check	Door is unlocked	False		
Enter	Resident name	John Smith	PIN	12345
Check	Door is unlocked	False		
Check	User prompted to enter info again	True		
Enter	Resident name	John Smith	PIN	12345
Check	Door is unlocked	True		

1. Draw a feature tree representing the specific instance of this feature as requested by the customer.
2. Draw a feature tree representing the current state of the feature in the product line context. If required, add any [min..max] constraints.

Part 3

A new customer requests to have the "Door Access Control" feature, but requires the following changes:

Select Table C
Add Table D

Optional	Table D. Owner notified after three failed attempts			
Enter	Resident name	John Smith	PIN	11111
Check	Door is unlocked	False		
Enter	Resident name	John Smith	PIN	22222
Check	Door is unlocked	False		
Enter	Resident name	John Smith	PIN	33333
Check	Door is unlocked	False		
Check	Owner notified	True		

Don't select Table B: since it cannot coexist with Table D.

1. Draw a feature tree representing the specific instance of this feature as requested by the customer.

2. Draw a feature tree representing the current state of the feature in the product line context. If required, add any [min..max] constraints.

Section II

Part 1

A customer of the Intelligent Home system would like to have a security system with a new feature "Open Window Detection." According to home security standards in Canada, at least two detection mechanisms need to be supported in a burglary detection system. The customer describes his request as:

Default	Table E. Notify owner if a window is opened when nobody at home.			
Enter	No of people currently at home	1		
Check	Window number	3	Is closed	True
Force value of window sensor	3	OPEN		
Check	Window number	3	Is closed	False
Check	Alarm is off	False		
Force value of window sensor	3	CLOSED		
Enter	No of people currently at home	0		
Check	Window number	3	Is closed	True
Force value of window sensor	3	OPEN		
Check	Window number	3	Is closed	False
Check	Owner notified	True		

Default	Table F. Detect if a window is opened between 12:00 am & 6:00 am.			
Enter	System time	12:00 am		
Check	Window number	3	Is closed	True
Force value of window sensor	3	OPEN		
Check	Window number	3	Is closed	False
Check	Alarm is off	True		

1. Draw a feature tree representing the specific instance of this feature as requested by the customer.
2. Draw a feature tree representing the current state of the feature in the product line context. If required, add any [min..max] constraints.

Part 2

A new customer would like to have the "Open Window Detection" feature with the following modifications:

Add Table G:

Default	Table G. Alarm goes off if a window is opened between 12:00 am till 6:00 am only if motion sensor in the backyard has been activated.			
Enter	System time	12:00 am		
Check	Window number	3	Is closed	True
Force value of window sensor	3	OPEN		
Force value of motion sensor	3m	ACTIVE		

Check	Window number	3	Is closed	False
Check	Alarm is off	True		

Remove Table F: as it cannot coexist with Table G
Add Table H:

Optional	Table H. If more than one window is broken into, police should be notified, and camera surveillance should be activated.			
Check	Window number	2	Is closed	True
Check	Window number	3	Is closed	True
Force value of window sensor	3	OPEN		
Force value of window sensor	2	OPEN		
Check	Window number	3	Is closed	False
Check	Alarm is off	True		
Check	Window number	2	Is closed	False
Check	Police notified	True		

1. Draw a feature tree representing the specific instance of this feature as requested by the customer.
2. Draw a feature tree representing the current state of the feature in the product line context. If required, add any [min..max] constraints.

Part 3

A new customer would like to have the "Open Window Detection" feature. He does not know how to make a choice from the existing feature, but he provides the following information:

- **He is not willing to pay for motion sensors.**
- **He does not have surveillance cameras.**
- **He would like to add Table I:**

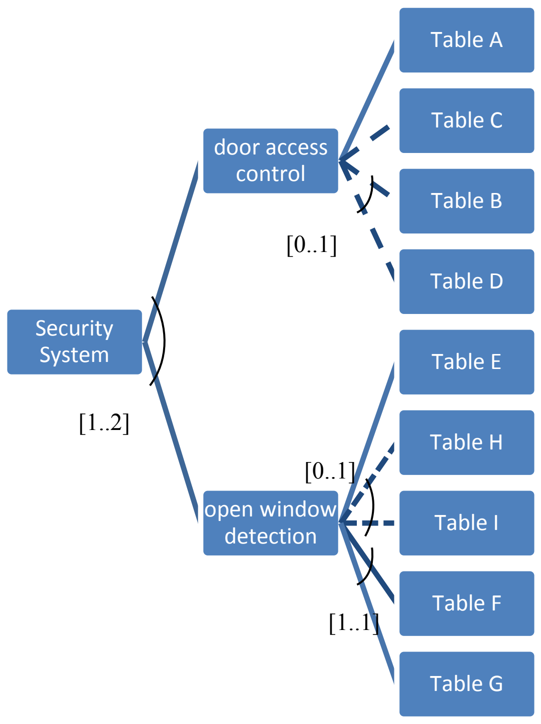
Optional	Table I. Police is notified only if at least two windows are broken into and alarm is not deactivated within 5 minutes.			
Enter	System time	1:00 am		
Check	Window number	2	Is closed	True
Check	Window number	3	Is closed	True
Force value of window sensor	3	OPEN		
Force value of window sensor	2	OPEN		
Check	Window number	3	Is closed	False
Check	Alarm is off	True		
Check	Window number	2	Is closed	False
Enter	System time	1:02 am		
Check	Alarm is off	True		
Check	Police notified	False		
Enter	System time	1:06 am		
Check	Alarm is off	True		
Check	Police notified	True		

1. Draw a feature tree representing the specific instance of this feature as requested by the customer.

2. Draw a feature tree representing the current state of the feature in the product line context. If required, add any [min..max] constraints.
3. Draw a feature tree representing both the "Door Access Control" and "Open Window Detection" features in a product line context. Both features belong to a module (feature at a higher level) called "Security System." Bear in mind that when a customer purchases the "Security System" module, he should choose at least one of the two features but can also select both.

Section III

Say, the result of Section I and Section II was the following feature tree:



1. Draw the configuration that yields the minimum cost for the customer assuming that he has to pay an extra amount of money for each added feature? There might be more than one. Any answer will do.
2. Draw the configuration that yields the maximum value to the customer assuming that he is paying the same amount of money for any given configuration? There might be more than one. Any answer will do.

Step 4. Post-questionnaire

Post-Questionnaire: In this experiment you have used a procedure based on a new model to manage variability in software product lines. Please rank each of the following statements on the given scale:

1. I found the procedure **realistic and practical**: I think the procedure can deal with real-life problems of different scales, and it does not make unrealistic assumptions.

Strongly agree	Agree	Neutral	Disagree	Strongly disagree

2. I found the procedure **easy to comprehend**: It does not take so much time and mental effort to understand how to apply the procedure on a given problem.

Strongly agree	Agree	Neutral	Disagree	Strongly disagree

3. I found the procedure **easy to apply**: when working with the procedure, it was easy to construct feature trees and understand relations between the requested changes as well as the [min..max] constraints.

Strongly agree	Agree	Neutral	Disagree	Strongly disagree

4. I found the procedure **flexible**: when executing the procedure, I could easily handle different scenarios without finding myself stuck with the semantics or/and notations of the procedure. The procedure was flexible enough to accommodate change requests of different complexities.

Strongly agree	Agree	Neutral	Disagree	Strongly disagree

5. I found it **easy to work with acceptance tests:** As a cohesive representation of the features in a software system, it was easy to understand and work with acceptance tests to produce feature trees.

Strongly agree	Agree	Neutral	Disagree	Strongly disagree

6. I found it **easy to read feature trees**: It was reasonably easy to understand what a feature tree tells about possible variations in a given feature including: the breakdown of the feature into smaller components (the hierarchical structure) and the [min..max] constraints.

Strongly agree	Agree	Neutral	Disagree	Strongly disagree

7. I found it **easy to use feature trees:** to produce different instances of the same feature based on customer requests.

Strongly agree	Agree	Neutral	Disagree	Strongly disagree

Please add any comments:

Detailed Results

Table 13 – Participants' background information

ID	**Background info** **Line 1 - Department: Specialization** **Line 2 - Experience in Industry** **Line 3 - Level of involvement in the following topics:** **Software Engineering: Agile Methods: Acceptance Testing: Product Line Engineering (where 1-> no background at all, and 5 -> heavily involved)**
1	ECE: SE Worked 1 year in Business Process Analysis 4:3:2:2
2	CS: SE No experience 4:5:4:2
3	CS:SE 2.5 years developing web applications 4:4:4:2
4	EE: Wireless Comm. No experience 3:1:1:1
5	CS:DB 2 years in network admin and software development 3:2:3:1
6	CS:SE No experience 3:3:4:2
7	CS:SE No experience 3:2:2:2
8	CS:SE 1 year in SE 4:2:3:2
9	CS:SE No experience 4:5:5:2
10	ECE:SE 1 year in SE 4:2:4:2
11	GE: LIS 2 years in networking and programming 2:1:1:1

ID	Background info Line 1 - Department: Specialization Line 2 - Experience in Industry Line 3 - Level of involvement in the following topics: Software Engineering: Agile Methods: Acceptance Testing: Product Line Engineering (where 1-> no background at all, and 5 -> heavily involved)
12	CS:BI No Experience 3:2:1:2
13	CE:IP 1-2 years in software development 3:3:2:3
14	CS:SE 3 years in Web applications and plugin development for VS 4:4:4:2
15	EE:BES 3 years in firmware 3:5:4:1
16	CS:DM 2 to 3 years in networking and database 3:2:2:1

Table 14 - Time and Scores

	Time					Scores		
ID	**TUT**	**Ex1**	**Ex2**	**Ex3**	**Total**	**Ex1**	**Ex2**	**Ex3**
1	12	4	10	5	31	95	86	100
2	14	5	10	2	31	80	93	100
3	14	8	10	2	34	70	89	100
4	20	12	30	5	67	100	86	100
5	19	6	18	4	47	95	64	100
6	9	6	13	4	32	70	86	100
7	8	6	6	2	22	100	100	100
8	8	5	13	3	29	85	96	100
9	9	9	15	3	36	95	96	100
10	15	12	18	3	48	70	96	100
11	17	9	23	8	57	95	96	100
12	15	8	13	6	42	95	71	100
13	9	9	17	5	40	100	100	100
14	21	12	16	6	55	85	96	100
15	12	6	15	2	35	85	89	100
16	14	13	25	7	59	95	86	*
Min	8	4	6	2	22	70	64	100
Max	21	13	30	8	67	100	100	100
AVG	13.5	8.1	15.8	4.2	41.6	88.4	89.5	100
STD	4.3	2.9	6.2	1.9	15.2	10.9	10.0	0.0

Table 15 - Questionnaire results

ID	**Q1**	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**
1	3	4	3	3	3	4	3
2	4	5	5	5	5	5	5
3	4	5	3	4	4	4	5
4	5	5	5	5	5	5	5
5	4	5	3	3	2	3	4
6	4	5	5	4	3	4	5
7	4	5	5	5	5	4	4
8	4	4	4	4	3	4	4
9	4	5	5	5	5	4	5
10	4	3	4	4	3	3	3
11	4	5	4	4	3	5	4
12	4	4	4	4	3	5	5
13	4	4	5	4	3	5	4

ID	Q1	Q2	Q3	Q4	Q5	Q6	Q7
14	5	5	4	4	5	5	4
15	4	4	4	3	4	4	4
16	5	4	4	5	5	5	5

Comments & Observations

ID	Participant's comments	Observations made by investigator
1	- answered post-questionnaire based on the feeling that I did not do well. - ATs didn't seem to be very relevant in the experiment.	- Didn't know how to use 0 as min for optional features. - didn't include one feature - Didn't realize the implicit constraint on H & I
2	- the model is easy to handle problems with not many conflicts, but I am not sure whether it would be still as easy when many constraints exist.	- participant didn't deduce the [0..1] constraint on H and I. - participant chose to group most of the optional features using [min..max] constraints.
3	- I think dealing with constraints is tricky and might cause confusion.	- participant assumed more constraints than necessary. - participant did not know how to impose constraints on higher level components (Sec III-part 3 – 3rd question).
4		- participant missed implied constraints on H & I. He confused a default feature with an optional one.
5	- Does this work well with more complex structures? For example, say you have a table that cannot coexist with another, but is needed with another table.	- Didn't know how to use 0 as min for optional features. - participant missed the addition of one feature. Also missed implied constraints on H & I.
6	- Scalability. This may or may not already be accounted for but after features/tables are always allocated together, larger trees may be useful to represent common sub trees as a single entity to reduce clutter.	- participant assumed more constraints than necessary. - participant missed implied constraints on H & I.

ID	Participant's comments	Observations made by investigator
7	- Feature trees can get very complex in a larger system.	- participant kept asking about what is to be grouped and what is not. That is, what defines a meaningful grouping constrain and what does not.
8	- The feature tree do not encode other constraints that might play a role within the feature. Also, I wonder if this would scale well for large systems.	- participant constrained features unnecessarily. - interestingly, participant used [1..1] instead of [0..1] rationalizing that the optionality is implied in the dotted line and thus [1..1] will do. - participant did not know how to specify the maximum for the higher level grouping.
9	- I found dealing with AT very easy.	- participant used [1..1] instead of [0..1] rationalizing that the optionality is implied in the dotted line and thus [1..1] will do.
10	- There is nothing mentioned in the tree about the priority of excuting tasks or procedures.	- participant ignored an explicit constraint between optional components. - participant used [1..1] instead of [0..1] rationalizing that the optionality is implied in the dotted line and thus [1..1] will do.
11		- participant used [1..1] instead of [0..1] rationalizing that the optionality is implied in the dotted line and thus [1..1] will do. - participant did not know how to specify the maximum for the higher level grouping.
12	- How will you deal with features of different weights. I.e. Both tables a and b are default, but the customer prefers a.	- participant used [1..1] instead of [0..1] rationalizing that the optionality is implied in the dotted line and thus [1..1] will do. - participant mistook a default component with an optional one. - Didn't realize the implicit constraint on H & I
13		

ID	Participant's comments	Observations made by investigator
14		- participant used [1..1] instead of [0..1] rationalizing that the optionality is implied in the dotted line and thus [1..1] will do. - participant constrained features unnecessarily.
15		- participant used [1..1] instead of [0..1] rationalizing that the optionality is implied in the dotted line and thus [1..1] will do. - participant constrained features unnecessarily [twice].
16		- participant used [1..1] instead of [0..1] rationalizing that the optionality is implied in the dotted line and thus [1..1] will do. - participant could not interpret explicit requirements. - Didn't realize the implicit constraint on H & I - didn't know how to apply [min..max] constraints on higher level abstraction. - didn't understand section 3 of the experiment.

Expected Model

The final model according to the intended interpretation

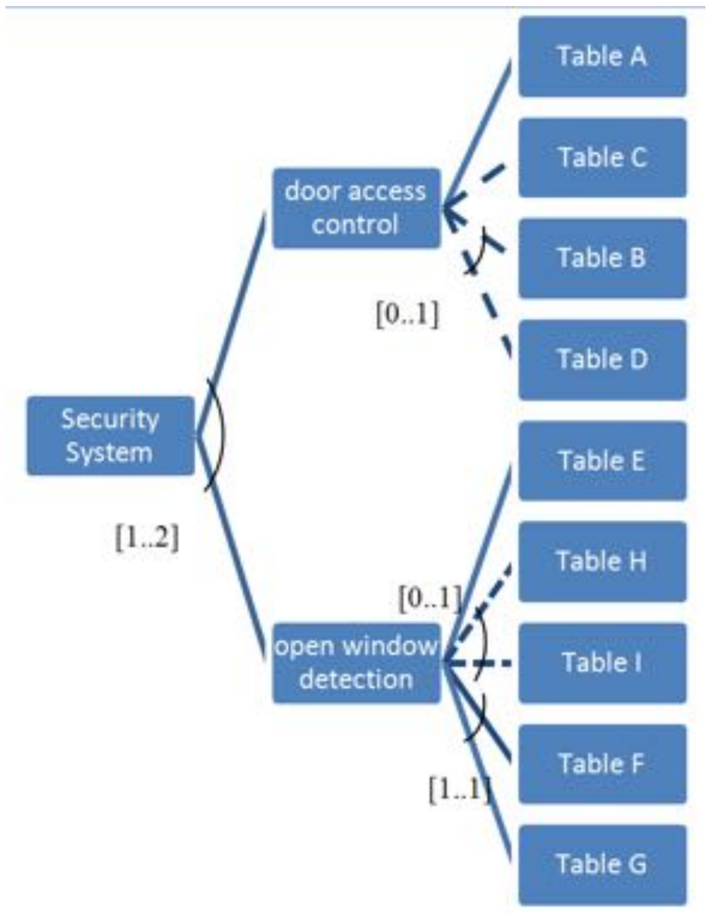


Table A
Table C
Table B
Table D
Table E
Table H
Table I
Table F
Table G
door access control
open window detection
Security System
[0..1]
[1..2]
[0..1]
[1..1]