University of Twente
Faculty of Electrical Engineering, Mathematics and Computer Science
Track Software Engineering

Master of Science Thesis

An investigation in qualities of software variability

by

Albert Brand

Supervisors:
ir. Arjan de Roo
Dr. ir. Lodewijk Bergmans
ir. Jos Jans

Venlo, 2009

Abstract

Océ Technologies provides a range of wide format printing systems. Several of these printers run similar software to provide a remote control web interface. One of the modules of this software is a job submitter, which lets customers quickly submit print jobs.

This range of software is created using a basic software product line setup. A product line separates the common and variant parts of software products. During product derivation, a specific variant of the product is chosen and the variant and common parts are combined again. Océ uses one out of several techniques to separate the variant parts from the common parts of the product.

These so-called variability realization techniques might have consequences on development efficiency and maintenance and on the efficiency of the final products. However, it is not yet clear how each technique affects these qualities.

Therefore, this thesis focuses on six variability realization techniques (VRTs). We collected these VRTs from literature and reformulated them in a design pattern style. We performed the DESMET feature analysis survey to assess the quality of the description of the techniques, the impact on efficiency of the development process and the efficiency of the final application, and the impact on the maintenance process in an industrial setting. Several software experts at Océ have taken part in this survey. Based on the outcome of this assessment, two VRTs have been selected and implemented in a prototype software product.

This thesis contributes:

- The six variability realization techniques in a design pattern style;
- The setup, execution and results of the feature analysis survey;
- The implementation description of two of the six techniques;
- Discussion on the consequences of each technique, relating the results of the survey, consequences from literature and implementation experience.

This thesis concludes with answers on how each VRT supports the introduction of variability, extensibility with respect to a new external component,

adaptation of an invariant part and preventive maintenance on a variant part, based on the qualities that each VRT affects. These answers can be used to select a technique with desired features.

Acknowledgments

Many thanks to my mentors Jos Jans, Arjan de Roo and Lodewijk Bergmans for giving valuable advice and providing feedback on the thesis; the colleagues at Océ for the interesting discussions on my topic; and the participants of the survey for taking the time to participate.

Special thanks to Astrid for her unlimited kindness and support.

Contents

Contents **i**

1 Introduction **1**
1.1 Océ . . . 1
1.1.1 Wide Format Printing Systems . . . 1
1.1.2 Web Application . . . 2
1.1.3 Job Submitter . . . 2
1.1.4 Layered Architecture . . . 3
1.1.5 Job Submitter and Layers . . . 4
1.1.6 Océ Goals . . . 5
1.1.7 Web Application Development Process . . . 5
1.1.8 Version Management . . . 6
1.2 Variability . . . 7
1.2.1 Identification . . . 7
1.2.2 Deciding on Extent . . . 7
1.2.3 Implementation . . . 8
1.2.4 Managing Variability . . . 8
1.2.5 A Model for Variability . . . 8
1.3 Structure of Thesis . . . 9

2 Research Problem **11**
2.1 Océ Problems . . . 11
2.1.1 Inefficient Extensibility of the Web Application . . . 11
2.1.2 Limited Adaptability of the Web Application . . . 12
2.1.3 Inefficient Maintainability of Final Products . . . 12
2.2 Problem Scope . . . 12
2.2.1 Problem Statements . . . 13
2.3 Research Questions . . . 14
2.4 Motivation . . . 15

3 Literature Review **17**
3.1 Previous Work Related to Variability 17
3.1.1 Variability Models 17
3.2 Previous Work Related to Research Problems 20
3.2.1 Adding an Extension Mechanism and New Extensions . . . 20
3.2.2 Creating an Adaptable Application 20
3.2.3 Maintaining an Adaptable, Extensible Application 21
3.3 Previous Work Related to Methodologies 23
3.3.1 Evaluating Impact on Software Qualities 23
3.3.2 Feature Abstraction of Design 24

4 Research Method **27**
4.1 Feature Analysis Survey 27
4.1.1 Data Collection 28
4.1.2 Methodological Problems 32
4.1.3 Methodological Limitations 32
4.2 Survey Conduction 32
4.2.1 Preparation 33
4.2.2 Execution 33
4.2.3 Data Validation 33
4.3 Data Collected 33
4.3.1 Survey Answers 33
4.3.2 Description Quality 34
4.3.3 Efficiency 34
4.3.4 Maintainability 37
4.3.5 All Features Combined 37

5 Implementation of Techniques **39**
5.1 Prototype Job Submitter 39
5.2 Introducing Variability in the Job Submitter 40
5.2.1 Identification of Variability 40
5.2.2 Deciding on Extent 42
5.2.3 Implementation of Variability 42
5.3 Variant Class Specialization Implementation 42
5.3.1 Model Layer 42
5.3.2 Controller Layer 44
5.3.3 View Layer 46
5.3.4 Domain Layer 46
5.3.5 Variant B 48
5.3.6 Evaluation 48
5.4 Variant Component Generation 49
5.4.1 Model Layer 49

5.4.2 Controller and View Layer 52
5.4.3 Domain Layer 53
5.4.4 Variant B 55
5.4.5 Evaluation 56

6 Discussion 57
6.1 Variant Class Specialization 57
6.1.1 Consequences From Survey 57
6.1.2 Consequences From Literature 58
6.1.3 Consequences From Implementation 59
6.1.4 Connecting Together 60
6.2 Inner Entity Variant 60
6.2.1 Consequences From Survey 60
6.2.2 Consequences From Literature 61
6.2.3 Connecting Together 62
6.3 Variant Code Adaptation 62
6.3.1 Consequences From Survey 62
6.3.2 Consequences From Literature 63
6.3.3 Connecting Together 64
6.4 Variant Template Instantiation 64
6.4.1 Consequences From Survey 64
6.4.2 Consequences From Literature 65
6.4.3 Connecting Together 66
6.5 Variant Architecture Specialization 66
6.5.1 Consequences From Survey 66
6.5.2 Consequences From Literature 67
6.5.3 Connecting Together 68
6.6 Variant Component Generation 68
6.6.1 Consequences From Survey 68
6.6.2 Consequences From Literature 69
6.6.3 Consequences From Implementation 70
6.6.4 Connecting Together 71
6.7 Final Thoughts 72
6.7.1 Limitations of Research 72
6.8 Summing Up 73

7 Conclusions 75
7.1 Research Results 75
7.1.1 Transition 75
7.1.2 Extending Variants 76
7.1.3 Adaptation of Invariant Parts 77
7.1.4 Maintenance of Single Variant 77

7.1.5 Common Consequences 78
7.2 Contributions of Research 79
7.3 Future Work 80

Bibliography **81**

A Variability Realization Techniques **89**
A.1 Variant Class Specialization 89
A.2 Inner Entity Variant 92
A.3 Variant Code Adaptation 94
A.4 Variant Template Instantiation 96
A.5 Variant Architecture Specialization 98
A.6 Variant component generation 100

B Survey Results **103**
B.1 Questions in Survey 103
B.2 Results 104
B.2.1 Variant Class Specialization 105
B.2.2 Inner Entity Variant 105
B.2.3 Variant Code Adaptation 106
B.2.4 Variant Template Instantiation 106
B.2.5 Variant Architecture Specialization 107
B.2.6 Variant Component Generation 107

List of Symbols and Abbreviations **109**

List of Figures **110**

List of Tables **111**

Chapter 1

Introduction

This chapter gives an introduction to Océ Technologies, variability in software and structure of this thesis.

1.1 Océ

Océ Technologies is an international supplier of printing systems, software and services for professional users. The company has departments in over 30 countries and employs about 23000 people worldwide. At the Research and Development department located in Venlo, the Netherlands, Océ develops new technologies and the majority of their product concepts.

Océ provides, next to other systems and services, a range of wide format printing systems (WFPS). These systems are available in black-and-white and color editions for various production and working group environments.

1.1.1 Wide Format Printing Systems

Though a typical WFPS is much larger than a conventional printer, their functionality is pretty similar. Both a conventional printer and a WFPS are controlled from a graphical display attached to the printer. Both have the notion of print jobs: a set of documents with specific media or formatting settings that are sent to the printer. Both provide means to manipulate these print jobs.

Also, similar to conventional printers, a WFPS can be connected to a network to let client computers remotely control the printing system. Remote control was previously only possible by installing specialized software drivers or applications on the client computer. Océ has taken a step forward by implementing a web based, zero install remote control web interface for a subset of their wide format systems.

1.1.2 Web Application

Océ has developed the web interface software in-house. We will refer to this software package as the Web Application (WA). The WA software provides remote control for a set of wide format systems and consists of a set of tightly integrated web applications that run on the embedded controller of the WFPS. Using these web applications, users can view information, submit jobs, manipulate queues and settings, and perform software upgrades.

The high-level goals that form the basis of the current WA software are:

Zero installation for clients
By providing a web application accessible through the client's browser, zero installation is realized. This leads to less (service) costs and more ease of use.

Ease of use
People with different levels of (technical) knowledge, trained and untrained, need to be able to use the WA user interface. Also, WA adheres to Océ's graphical design standards. This makes the user interface of the product recognizable.

Support multiple hardware architectures
Each printer has its own unique printer hardware architecture. WA is designed to run on multiple system architectures.

Support multiple platforms
Due to different printer hardware, different operating systems need to be supported. WA is designed to run on multiple platforms.

Support multiple projects
WA currently supports three different wide format printing system projects. Each project has a specific subset of functionality that needs to be supported.

1.1.3 Job Submitter

A part of WA is the Job Submitter (JS). The main function of JS is to allow users to quickly submit print jobs via a web browser. A print job consists of one or more files in a supported graphical format and a job configuration that describes the desirable media and formatting settings. Such a job configuration is called a job ticket.

JS offers the possibility to select files from the client user computer, to change media, formatting and workflow settings and to select presets. When a user submits a new print job to JS, JS creates a job ticket and sends the job ticket

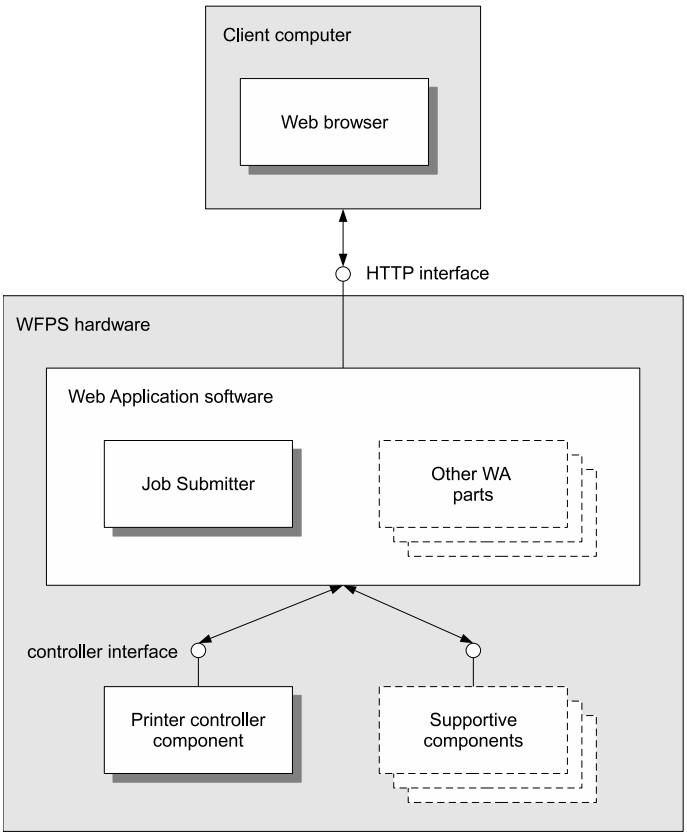

Figure 1.1: Overview of the Web Application architecture

and file data to the printer controller. When jobs are queued by the printer controller, the job details and status can be viewed by JS.

Figure 1.1 gives an overview of the WA architecture that is shared between the different projects. At the top is the client computer with a web browser. The computer communicates with the WFPS hardware through the network. The WA software consists of multiple parts. JS is shown in the context of the other parts. JS communicates with the printer controller and supportive components by using their respective interfaces. The supportive components consist of for instance a printer settings cache.

1.1.4 Layered Architecture

The WA software is based on a layered architecture. The idea behind this layering is separation of concerns, which makes the development of the system better controllable. There are three different layers in WA:

- The Access layer, which contains everything related to presentation
- The Model layer, which comprises the state of the applications

- The Boundary layer, which consists of the bi-directional interfaces to components outside of WA

At first glance, the separation in layers has similarities with the Model-View-Controller (MVC) design pattern [GHJV95]. However, there is a subtle difference. We describe these layers more in detail.

Access layer

This layer handles the client sessions, user interaction and visual response. This conforms roughly to a combination of the View and Controller layer of the MVC pattern. The system where actions are performed upon is actually the underlying Model layer. This is not modeled explicitly in this architecture.

Model layer

This layer manages the application-wide, persistent data. The layer consists of several manager classes. These classes read all long-term object data from the relevant data sources on application initialization. They also provide interfaces to read and manipulate these objects. These objects generally are unique objects that purely encapsulate data, as described in the Singleton and Data Access Object pattern [GHJV95].

Boundary layer

This layer abstracts the communication with the components outside of WA. This makes WA less dependent on the exact implementation of protocols. Also, the Access and Model layers can be reused for multiple hardware architectures by implementing alternative Boundary layers.

It is not the goal of the Boundary layer to provide one uniform interface for any printer where WA is embedded in. Instead, it supports several features, which all have a generic interface and multiple printer-specific implementations.

1.1.5 Job Submitter and Layers

JS is also separated in an Access, Model and Boundary layer.

Access layer

This layer consists of the user interface that has a printer-specific part. This makes it possible to select a specific user interface part for each printer-specific build.

Model layer

This layer contains different printer-specific status objects that adhere to a generic interface. It also has a job delivery interface that generates a

printer-specific job ticket for the printer controller, given the submitted data from the client. The job delivery interface would fit better in the Boundary layer, as the Boundary layer is meant for encapsulation of the implementation details of communicating with the printer controller.

Boundary layer
This layer provides an interface for sending job tickets. The interface has multiple implementations, which are not meant to support different printers, but to support different communication mechanisms. The actual printer-specific configuration is expressed in the job ticket.

1.1.6 Océ Goals

Océ has set the following goals for the continuing development of WA:

Lower cost per release
Océ takes part in a very competitive market, with competitors like HP and Xerox that offer similar products. To stay in this competitive race, cost efficiency is key. There is a constant pressure to lower costs for every printing system developed, and that also includes the development of the WA software. Partly, this can be realized by outsourcing parts of the work to lower cost departments. But, Océ believes that a lot of cost reduction can be realized by optimizing the development process.

Run WA on more printing systems
There is a high demand to run WA software on more different printing systems. Other department units that produce printing systems have their own remote control applications, which are very similar to the WA software. These units could profit from the effort that already has gone into the WA software, if WA is adaptable enough to support the other printing systems' requirements.

React efficiently to changing requirements
And finally, Océ must be able to adjust its WA software quickly to react efficiently to changing requirements. The process of creating a suitable WA release is running parallel with the ongoing development of other parts of the printing system. This development can constantly influence the requirements of WA.

1.1.7 Web Application Development Process

The WA developers have set up a Development and Maintenance Environment (DME) for delivering printer-specific systems. This environment consists of source

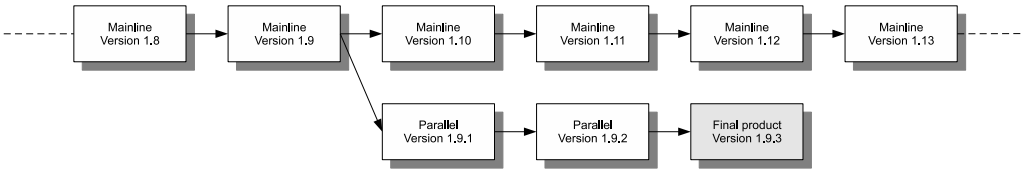

Figure 1.2: The mainline and a branch of the Web Application

code and a set of applications and tools that are necessary to build the printer-specific products.

Each release that is built reuses many common components. The different, or variable parts of each printer-specific release are mixed in using a choice of techniques. For instance, for each build, the build framework copies the relevant compiled common and printer-specific classes to the target build directory. Using this environment, a developer can produce a build, which is a compiled, printer-specific end product.

1.1.8 Version Management

All changes to the WA software are committed to a version management repository. This makes it possible to retrieve older versions of the software and compare differences between two versions of the software. Also, the version management system provides release management support. There is a repository that contains the WA-specific mainline. The mainline consists of all the WA-related software artifacts, organized in a directory structure.

At a certain point, the development of the WA software is regarded finished with respect to a set of requirements. The WA software is then ready to be integrated with the other components of the WFPS. After the integration, the WFPS will be thoroughly tested. This integration and testing process can easily take two weeks. During this process, the integration code may not be altered. Also, it is undesirable to have to wait until the process is finished. This is solved by creating a branch from the mainline, using the release management software. This process is clarified by Figure 1.2.

A branch contains all the software artifacts from the mainline, but is isolated from changes on the mainline. Vice versa, changes to artifacts in the branch do not affect the mainline either. Mainline development can continue immediately, while the branch can receive final touches. If bugs are found that affect the mainline and branched versions, fixes need to be incorporated on the mainline and all affected branches. This is done using an automatic process called merging. However, it is sometimes not possible to do this automatically, and requires manual developer labor.

After the release of the final WA build for a specific printer, the branch is not kept up to date. When the specific printer needs a new software release, the

developers prefer to branch the mainline again instead of continuing to work on an old branch. They need to merge any finishing fixes from the previous branch, which is again not always possible automatically.

1.2 Variability

The description of the WA development process in section 1.1.7 mentions different release variations. The process of supporting different variations of a software product is closely related to the concept of variability.

Software variability is the ability to vary parts of the software system at some point in the software lifecycle. Such a part is called a software artifact. This can be the software architecture design, the detailed design, components, classes, lines of source code, executable binaries and so on. By making artifacts variable in a product, products can be produced with varying characteristics. To make artifacts variable, several techniques exist to make this possible. Such a technique is called a variability realization technique (VRT) [SvGB05].

Several steps can be identified in the process of introducing variability in a product.

1.2.1 Identification

The first step is to identify where variability is needed. Often, the product is modeled as a set of features. Features separate the application in useful high-level parts. By modeling the product using features, it is easier to identify which features of the system need to be variable. A set of features from which only one can appear in the final product is called a variant feature; a single feature from this set is called a variant.

1.2.2 Deciding on Extent

The next step is identification of the required extent of the variability. The variant feature needs flexibility to suit the current and future needs of the system in a cost-effective way. To determine the extent of the variability, there are three important processes that need consideration: the introduction, population and binding of the variant feature.

Introduction

To introduce a variant feature, variation points need to be added to the system. These variation points are the mechanisms that provide the selection of a variant from a variant feature. Each VRT has its specific variation point mechanism.

Population

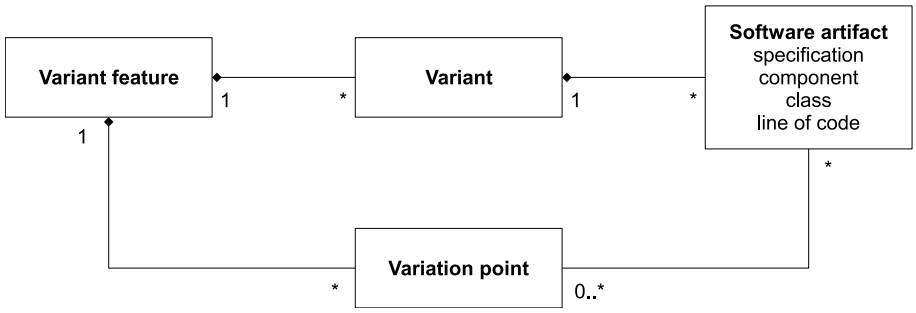

Figure 1.3: A model for variability

When the variation points are in place, the variants of the variant feature are added to the system. Each VRT has a certain way of adding a variant.

Binding

After population of the variant feature, one of the variants of the variant feature is selected. The selected variant will contribute to certain characteristics of the software system. Each VRT has a certain way of selecting a variant.

These processes take place during a certain moment in the software lifecycle. Each VRT support different moments of introducing, populating and binding the variant feature. Deciding on the required extent has to take these characteristics of VRTs in consideration.

1.2.3 Implementation

The following step of introducing variability in a product is the actual implementation. After selecting the most suitable VRT, the variant feature is introduced in the product and the variants are expressed. By selecting variants from variant features, different products can be produced.

1.2.4 Managing Variability

The last step is to manage the variability. Variant features can be populated with new variants and no longer used variants can be removed. New variant features can be introduced and some variant features need to be removed altogether, as the product requirements change.

1.2.5 A Model for Variability

The relation between variant features, variants, variation points and software entities in the software product are modeled in Figure 1.3 [SvGB05].

Each variant and variation point connects to one variant feature. Each variant can consist of multiple software artifacts: a variant can be expressed in many ways, for example as a specification in a domain-specific language, a set of classes, a single class, a few lines of code and so on. Also, each variation point can consist of multiple software artifacts, as it an implementation of the variant mechanism. Variation points do not have to appear in the software product.

1.3 Structure of Thesis

First, we present the research problem, research questions and the motivation for these questions in chapter 2. We give an overview of relevant previous work related to these problems and the methodologies that are used to investigate these problems in chapter 3.

We present the research in chapter 4. This chapter contains details on how the collection of data is performed and analyzed. Also, the limitations of the methodology are discussed. Finally, the data results and the analysis are presented.

Interpretation of the results will follow in chapter 6, where we discuss the various aspects of the results. In chapter 7 the conclusions are drawn, and future and related work is discussed.

Chapter 2

Research Problem

This chapter gives an identification of the problems with WA, the scope of the research questions, the formulation of the research questions and the motivation for attempting to answer these questions.

2.1 Océ Problems

Océ feels that there are many areas in their software production process that can be improved. Although the DME supports the production process for the current set of printers adequately, several problems have been identified. The following sections explain each problem thoroughly.

2.1.1 Inefficient Extensibility of the Web Application

Even though WA has been designed to support multiple projects (see section 1.1.2), the process of adding support for a new printer is not regarded cost-efficient. From previous experiences, Océ knows that it takes about 30 to 40 hours of development time to implement changes in the software to support a new printer product. Océ believes that this process can be much more optimized and envision a reduction in developer hours of 50% or more. However, they have not researched how this can be realized.

At Océ, we have found almost no documentation on this implementation process; development is done by experts on the WA architecture and intrinsics of the software. In the WA software, the variable parts are very coarsely separated throughout the code tree. An observation is that the printer-specific parts are actually spread through each layer (as described in section 1.1.4), which implies that the layered approach does not help to separate the printer-specific parts from the common parts.

2.1.2 Limited Adaptability of the Web Application

The WA software is not designed to handle change other than the different architectures, platforms and projects as described in section 1.1.2. Agile programming techniques are employed to quickly react to changing requirements of the common and variable software parts. However, as this separation is not clearly realized, implementing changes can cause conflicts between the common and variant parts of different product variants. Océ has not yet researched this limitation of their development process.

There is a long-term vision for several larger change scenarios. Due to uncertainty and the dependence on other units at Océ, the choice of which scenarios to support is delayed and no preliminary redesign work is performed. It is unclear if the current variability mechanisms are sufficient to support larger change scenarios and delayed choice.

2.1.3 Inefficient Maintainability of Final Products

Océ has chosen for a traditional release management strategy that separates development of final products from baseline development. This separation leads to limited maintenance efficiency, as extra developer effort has to be spent to perform maintenance on each separated final product. Océ has not researched alternative release management possibilities using different variability mechanisms.

2.2 Problem Scope

The problems identified at Océ are good examples of real-life software engineering problems. Extensibility, adaptability and maintainability of software has been thoroughly researched in the last decades. Many different opinions exist on how to create extensible, adaptable and maintainable software. It seems that there is no silver bullet; a software developer must be happy when, through the use of different techniques, the problems are partially and often sub-optimally dealt with.

Based on the problems identified at Océ, we broaden our scope of the research area. It is not our goal to find solutions that are catered for the specific problems of Océ. Instead, we want to focus our research on a more generic software application that needs to support a variant component outside of the application. This software application uses a variability realization technique to implement the variable part of the application. This is a broader scope that is applicable to many application development.

We will focus on development of a prototype application. We use the informal Job Submitter specifications as a starting point for prototype development. The Job Submitter application is a straightforward and isolated application that

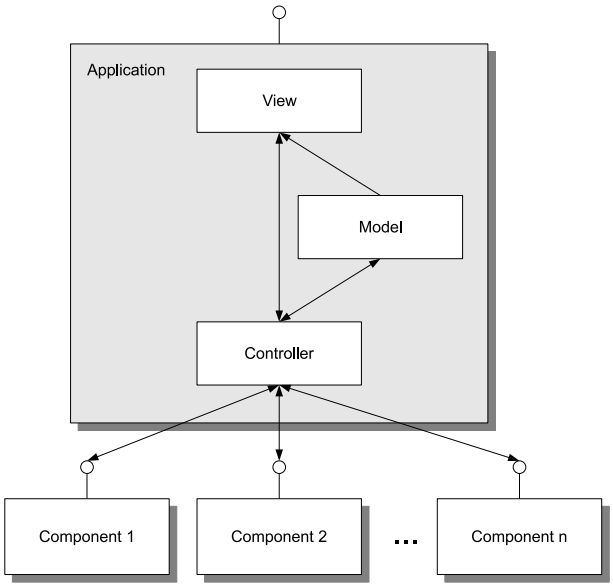

Figure 2.1: Architecture of the Base Application

contains a single variation point, which makes the design a good candidate for prototype development. We refer to the prototype application as the Base Application (BA). Figure 2.1 gives an architecture overview of this application.

BA is separated in three parts: a Model, View and Controller part. The View part retrieves values from the Model and provides a user interface for the user interaction. The Controller processes the user interaction, updates the Model and provides the communication with external components. The Model defines the representation of the application data.

The BA supports a family of external components. These components are similar in functionality. For instance, at Océ the components that are supported by the Job Submitter are printer controllers. Which variability realization technique the BA implements to support these external components is left open in the design.

2.2.1 Problem Statements

The problem statements that we will address are:

Efficiently transform an existing application that supports a single external component into a BA

An existing application that consists of a Model-View-Controller separation and supports a single external component needs to support multiple components. The application is then transformed into a BA. A variability realization technique is implemented, which has an impact on different parts of the application.

For Océ, this is a situation that already has occurred. We like to revisit this situation to give a complete overview of the qualities affected.

Efficiently add support to a BA for an external component that is similar to the components already supported

Support for a new external component is added to a BA. This has an impact on different parts of the application. The application might need to represent the data for this component differently. It might need a different model to store this data. It might need to communicate differently with the component to fetch and send data. These changes need to be performed as efficiently as possible.

For Océ, this is a situation that is described in section 2.1.1.

Adapt a BA with the least effort to meet new requirements

Apart from supporting a new external component, other functional requirements for a BA need to be implemented. Implementing these functional requirements has an impact on different parts of the application. Invariant parts of the application might need to be added, changed or removed. These changes need to be performed as effortless as possible.

For Océ, this situation is described in section 2.1.2.

Reduce preventive maintenance effort for a BA

After a BA has been released for a specific external component, the specific release needs to be maintained for a certain period. However, the extending and adapting process of the application continues. Preventive maintenance on the variant part of the application needs to be supported as efficient as possible.

For Océ, section 2.1.3 describes this situation.

2.3 Research Questions

During our literature study, we will identify different variability realization techniques and different qualities that might be influenced by the techniques. Then we investigate the perceived impact in these qualities, using experts at Océ. Based on the investigation of these qualities, the literature study and the implementation of several VRTs as part of the BA, we try to give a conclusive answer on the consequences of each VRT.

This gives us the following research questions to address.

R1 What is the perceived impact on system qualities of different variability realization techniques?

R2 How does each VRT support the transformation of an existing application into a BA?

R3 How does each VRT support extensibility of a BA with respect to a new external component?

R4 How does each VRT support adaptation of an invariant part of the BA to meet new functional requirements?

R5 How does each VRT support preventive maintenance on the variant part of a BA?

2.4 Motivation

A part of the motivation for these research questions can be contributed to Océ. For Océ, it is important that steps are undertaken to solve the problems they have identified. Océ believes that in the long term, if development effort can be significantly reduced, this will lead to cost reductions. By investigating the different possibilities of handling variability, a more efficient alternative for their situation might be uncovered.

The motivation to broaden the scope and investigate generalized problems comes from the thought that the situation at Océ is not unique. There are many applications similar to the base application. By researching the generalized problems, we hope to find generic answers that cover the whole spectrum of such applications.

Chapter 3

Literature Review

This chapter gives an extensive overview of literature on previous work related to variability, the research problems stated in chapter 2 and the relevant research methodologies.

3.1 Previous Work Related to Variability

Software variability has gained attention as a subject of research in the past decade. A short introduction on the notion of variability is given in section 1.2. Variability is a system quality. It is the ability of a software system to be efficiently extended, changed, customized or configured for use in a particular context [SvGB05].

3.1.1 Variability Models

Variability can be modeled in many different ways. One of the models of variability is described in section 1.2.5. This model describes a possible separation of the software entities that are involved in the variability realization.

However, many more models are proposed that model the details of variability. Some models are specifically designed for an existing software product line [JB02]. Some incorporate the process around the introduction and managing of variability into the model [BLP04], [SDNB04]. Some incorporate other qualities of software into the model [EV05].

3.1.1.1 Variability Realization Techniques

Just as there are many different models for variability, the actual realization of variability in an application can be implemented in different ways. Many different

variability realization techniques exist. Bosch et al. were the first to explicitly describe several VRTs [Bos00].

Svahnberg et al. provide a non-exhaustive taxonomy of VRTs, where they describe, in a design pattern-like style, the relevant factors to determine which technique is most appropriate for implementing variability [SvGB05].

Several other techniques can be found in literature. A technique that is similar to what Svahnberg describes as code fragment superimposition is called change set modeling [HvdH07].

Figueiredo et al. describe an aspect-oriented technique that is similar to code fragment superimposition. Preliminary results of their research are that this technique leads, in certain scenarios, to better stability and modularity of products [FCS+08].

Van Gurp et al. describe a technique to realize variability by using a set of web service related technologies to integrate products constructed from multiple, independently developed product lines [vGS06].

Becker et al. use a domain specific language (DSL) to model the specification and realization of variability [BVG+02]. This idea is later reused by Weber et al. They describe a prototype system where application behavior is described in a DSL, which according to Weber, makes modification fast and easy and makes software components more flexible and reusable [WCD+05].

For even more flexibility, Dolstra et al. introduce the concept of timeline variability, which is a method to vary the moment of variability binding [DFV03]. Also, Myll et al. propose to make the quality attributes of a system explicitly variable, to create systems that make an explicit trade-off between quality values [MMR06].

3.1.1.2 Effects of Variability Realization Techniques

As a variability realization technique is part of the system, the technique might have an effect on one or more quality attributes of the system. Quality attributes are the non-functional attributes of a system, as they relate to the desired properties of the system that are not explicitly related to functionality [BKLW95].

Svahnberg et al. provide a list of variability realization techniques and its effects on software. However, even though some of the effects of introducing variability are described extensively in the taxonomy, the affected quality attributes of the system are not explicitly structured or compared. This makes the consideration for a certain technique with respect to certain quality attributes still a very subjective process.

Harrison discusses the current limitations of architectural patterns on their impact on quality attributes, and proposes integrating this information in order to increase the usefulness of architectural patterns [HA07].

3.1.1.3 Qualities affected by Variability Realization Techniques

We want to know which qualities are affected by VRTs. To give insight in affected quality attributes of a system, the software can be quantitatively evaluated. Quantitative evaluation of software quality has been done for decades. Boehm et al. described over 30 years ago a framework for quantitatively assessing the quality of software [BBL76]. They define certain measurements based on the structure of the code.

One of the aspects Boehm et al. introduce is the quality utility tree. This tree structures the different qualities of software and shows the relation between very generic qualities (such as maintainability) and its underlying qualities (such as modifiability and testability).

However, it is very hard to actually measure the impact of a certain variability realization based on code measurements of the final product. A variability realization consists of the mechanism to handle different variants and the variants itself. This mechanism can be fully outside the final product code. For the code measurements, it would appear as if the mechanism did not exist. Also, the variants can be expressed in a different language or in a specialized format. Code measurements on these variants is non-trivial.

As we see, quantitative evaluation of the application has its limitations. Alternative evaluation methods have been proposed. For instance, Briand presents a methodology to combine two software product evaluation methods: measurement of structural design properties and evaluation of change scenarios. The results of these methods can be combined. However, it is clear that it is not the definitive answer [BW01].

3.1.1.4 Variability Problems

When introducing variability, other factors need to be taken into account. Variability is not limited to a single technique; its realization can be done by several techniques. However, each technique has its limitations. By creating a mixture of techniques, there may be unforeseen limitations on variability. Also, the mixture introduces complexities that might limit the development of the application even more. Deelstra et al. point at the limitations of mixing variability mechanisms as the main reason to model variability in a software system [DS08]. They believe that modeling variability makes variability more predictable and manageable.

A closely related problem that is addressed in the research community is the notion that, using different variability techniques, only a subset of correctly functioning applications can be derived from all the possible combinations of variants. Tekinerdogan et al. call this subset the design space model [TA04]. The variability model has to take this into account.

It is considered a problem if the variant binding moment is not exactly determined beforehand. To support variants that have a variable binding moment

introduces overhead and complexity in the process [DFV03]. However, such supportive methods have been investigated. Tooling has been designed to support variability that has a variable moment of binding [vdH04].

A concern for variability management is the scoping of its extents. Variability models need to be prepared for concrete projects before they can be effectively utilized in the process [RGD07].

Another problem that needs to be addressed is a configuration problem. If the variability configuration needs to be performed by people who are not software developers, this can lead to configuration misunderstandings and resulting bad software quality [BM04b].

3.2 Previous Work Related to Research Problems

For each problem, an overview of work found in literature related to the problem is given.

3.2.1 Adding an Extension Mechanism and New Extensions

Extensibility is a system design principle where a software implementation takes into consideration future growth. If a system is not prepared for certain future extensions, this will naturally lead to higher costs during the extending of the product.

To realize extensibility, the system architecture needs to be prepared for it. However, little research has been done on how a software architecture can incorporate such a quality. Architecture designers typically rely on their intuition and experience [KBI94]. Often, design patterns are used to solve design issues [GHJV95]. Design patterns generally make it easier to design an architecture. However, there is a downside to using design patterns, as it can lead to an increase in code complexity.

3.2.2 Creating an Adaptable Application

In literature, a gap between software requirement adjustments and mechanisms to implement these adjustments is identified. This gap causes mechanisms that are not appropriate for the context of the adjustment to be chosen to evolve the software. This is called the software evolution problem. Ciraci et al. provide a taxonomy of contexts and the appropriate mechanisms to evolve the software without deteriorating quality attributes [CvdBA07]. However, the approach they use is not validated in a practical context.

Others have identified problems with changing requirements. The fundamental problem is that many changes actually required are those that designers just cannot think of in advance. Van Gurp et al. introduce the concept of design

erosion. Design decisions accumulate and become invalid because of new requirements. Van Gurp states that there is no strategy that leads to an optimal design because of unforeseen requirement changes. These changes invalidate design decisions that once were optimal [vGB02]. This is also the conclusion of [EGK+01], although there is no evidence for extreme design decay that cannot be overcome.

3.2.2.1 Change Scenarios

Even though not all changes can be predicted exactly, it is still desirable to prepare an architecture for probable change. To analyze the qualities that an architecture possesses with respect to future change, different methodologies have been proposed that use a set of change scenarios as predictor of change.

The Software Architecture Analysis Method (SAAM) analyzes modifiability, flexibility, maintainability and other quality attributes [KBAW94]. The Architecture Trade-Off Analysis Method is a revised version of SAAM, and analyzes similar quality attributes [KKC00]. The Architecture-Level Modifiability Analysis is also based on SAAM and focuses specifically on modifiability [BLBvV04].

Some remarks can be made on the use of scenario based analysis. The outcomes of the methods are all highly company culture dependent; what works in one company may not be a solution in another [IHO04]. All methods lack means of validating estimates and completeness of the analysis. Some scenarios will be very unrealistic, and there will always be alternative scenarios that have not been considered. Also, prioritization of scenarios by voting might not lead to desirable outcome [IHO04].

3.2.2.2 Incorporating Change through Variability

America et al. have proposed a scenario based analysis method that takes the software and its variability in account [AHI+05]. Deelstra et al. go a step further and propose an analysis method that considers variability as a first-class citizen of product line development and explores multiple solutions for change [DNBS04].

3.2.3 Maintaining an Adaptable, Extensible Application

Software maintenance is the act of keeping software in an existing state of efficiency, validity and preserve it from failure or decline. For this, a constant process of change is necessary [LB85].

According to the ISO standard on software maintenance, there are four different types of maintenance [ISO98]:

- Adaptive maintenance, which makes the program become suited to different conditions;
- Corrective maintenance, which removes defects from the program;

- Perfective maintenance, which improves certain qualities of the software; and
- Preventive maintenance, which reverses the software's deterioration.

We cover preventive maintenance problems in this section. Adaptive maintenance is covered by the extensibility and adaptability of the application. We will not include corrective and perfective maintenance in this thesis.

Harrison et al. state that, to keep software maintainable, flexibility needs to be introduced that does not deteriorate the quality of the software [HA07].

Land et al. approach preventive maintenance formally and introduce a model that incorporates maintainability and software deterioration. They discuss the problem areas in this model. However, the model has not yet been validated [Lan02].

3.2.3.1 Software Configuration Management

To create different versions of products, a software configuration management (SCM) system can be used. Conradi et al. provide an overview and classification of different versioning paradigms [CW98]. Although newer SCM systems have been introduced since this publication, the paradigms have not dramatically changed. The techniques that are used by a SCM system to manage the evolution of large and complex systems have not changed in the last decade.

One of the reasons to use a versioning system is better support for software reuse. Software reuse can improve the software development processes, decrease time-to-market and costs, and improve product quality. However, effective reuse requires much more than just code and library technology: careful consideration must be given to people, process and technology [Gri93].

3.2.3.2 Software Product Lines

One approach to systematic reuse is the concept of a software product line (SPL). An SPL is a set of software systems sharing a common, managed set of features that satisfy the specific needs of a particular market and that are developed from a common set of core assets in a prescribed way [CN02].

One of the development techniques that can be employed is generative software development. The technique focuses on automating the creation of system-family members: a given system can be automatically generated from a specification written in a textual or graphical domain-specific language [Cza04].

However, the SPL method has its problems. Deelstra et al. identify a collection of product line development problems that cause unwanted extra costs. They elaborate that these problems stem from the underlying complexity of variability and will eventually arise in any organization that uses a software product line.

They provide possible solutions and research issues for their identified problems [DSB04].

3.2.3.3 Towards a Software Product Line

It is not necessary to adopt all parts of the SPL architecture to create variant products. Staples et al. describe an approach of adopting product line development for a pre-existing product set without the use of a product line architecture. This approach relies on a more simple configuration management infrastructure [SH04].

Deelstra et al. describe how the concept of model driven architecture can be used as an approach to develop products in a software product line, and how variability in the product line benefits from such a technique [DSGB03].

3.3 Previous Work Related to Methodologies

An overview of relevant methodologies in literature is given.

3.3.1 Evaluating Impact on Software Qualities

How can the impact of a certain method on software be evaluated? The DESMET methodology describes nine different techniques for evaluating software engineering methods and tools [KLL97]. One of the specific objects that can be evaluated is a generic paradigm for some aspect of software development. This makes the methodology relevant for this thesis.

The DESMET methodology makes a distinction in quantitative and qualitative methodologies. In the context of this thesis, the feature analysis survey is relevant. It is a qualitative evaluation, with input from participants who have studied the method but might not have used the method before.

Another methodology that is proposed is evidence-based software engineering. The methodology aims to improve decision making related to software development and maintenance by integrating current best evidence from research with practical experience and human values [DKJ05].

Other, more specialized methods to evaluate software impact can be found in literature. Especially for the evaluation of variability realization techniques, Fritsch et al. present a method to identify, document and evaluate these techniques [FLS02]. However, the authors provide little examples and do not give an extensive validation for their method.

When looking at quantitative methods to measure the impact on software, the Goal-Question-Metric (GQM) technique is a viable technique. GQM defines a certain goal, refines this goal in certain questions and defines metrics that should provide the information to answer these questions [BCR94].

When prioritizing different methods, a generic multi-criteria decision making method (MCDM) can be used. One of the most familiar MCDM methods is the Analytic Hierarchic Process (AHP). This process provides a framework for structuring a problem, for representing and quantifying its elements, for relating those elements to overall goals, and for evaluating alternative solutions [Saa80]. Svahnberg et al. propose a methodology to find out which quality attributes are supported or obstructed by different architecture approaches by applying AHP [SW05].

3.3.2 Feature Abstraction of Design

Kang et al. introduced the Feature-Oriented Domain Analysis method. The method identifies distinctive user-visible characteristics of software systems in a domain. These characteristics are called features. Features of software define both common aspects of the domain as well as differences between related systems in the domain [KCH+90].

Lee et al. clarified the concept of features and the goals of feature modeling, and provide practical guidelines for successful product line software engineering [LKL02].

Beuche et al. present a set of tools that use an extended feature model to provide user changeable customization of the software artifacts to be managed [BPSP04].

3.3.2.1 Decomposing Development

Component-based development (CBD) is a decomposition technique. CBD is the process of decomposing the system into functional or logical components. If this decomposition is properly performed, components can be reused to create new applications [BS97].

Nierstrasz et al. have been one of the first to realize that building applications from components can have many benefits such as effort reduction. However, they also identified several problems such as lack of standardization of the components [NGT92].

Another decomposition technique is separation of concerns (SoC). SoC is the process of decomposing the system into distinct concerns that overlap in functionality as little as possible. A concern is any piece of interest or focus in a program, which typically is synonymous with a feature or a behavior.

Dividing an application into several layers is an example of SoC. The Model-View-Controller design pattern is based on this concept [GHJV95]. If used correctly, it can lead to effort reduction. However, dependencies between the concerns may have a strong influence on modifiability of software architectures [Ber06].

Using a supportive application framework helps creating an application with the advantages of decomposition. Several web application frameworks exist that promote decomposition, for instance Wicket, Tapestry and JavaServer Faces [wic], [tap], [jsf]. Also, in literature, alternative prototypes for application development can be found [Zdu06].

Chapter 4

Research Method

This chapter describes the chosen methodology and contains details on how the collection of data is performed and analyzed. Also, the limitations of the methodology are discussed.

4.1 Feature Analysis Survey

In the literature review we described the DESMET methodology. The DESMET methodology consists of nine different evaluation methods to assess the qualities of a method or tool. One of these methods is a formalized, qualitative evaluation methodology to assess certain features of a method or tool. This method is called a feature analysis survey [KLL97].

We will perform a feature analysis survey to assess the qualities of certain VRTs in the context of Océ. In this chapter, we will use the DESMET terminology: features for quality attributes and methods for variability realization techniques.

The reason we decided to perform a feature analysis survey is that there are no process metrics available at Océ to undertake quantitative evaluations on the effects of using such methods.

The objective of the evaluation is to provide input for a decision about which method out of several is to be used under given circumstances. The results of the evaluation provide information about the suitability of the method for its purpose, the other advantages of the method and the drawbacks of the method.

Each method has certain features. A feature is a set of distinguishable characteristics of a method. These characteristics represent certain functional or quality properties of the method. Based on these features, the methods are analyzed and ranked. To retrieve the data for this analysis, different experts at Océ are asked to fill out a questionnaire.

4.1.1 Data Collection

The feature analysis survey can be split in two parts: the setup of the survey and the survey conduction part. The following section covers the steps undertaken to setup the survey. The actual survey conduction is covered in section 4.2.

4.1.1.1 Selection of Candidate Methods

The first step is to identify which methods are assessed by the feature analysis. These candidate methods are described in Appendix A. We choose to select methods that can bind the variability at build time.

The following variability realization techniques are chosen as candidate methods:

- Variant class specialization
- Inner entity variant
- Variant code adaptation
- Variant template instantiation
- Variant architecture specialization
- Variant component generation

The candidate methods have been based on:

- several techniques from the variability mechanism taxonomy [SvGB05];
- the variant architecture specialization technique from the software architecture guide [Bos00]; and
- the domain specific language technique [BVG+02].

We have rewritten the methods in a unified design pattern style, with concrete examples of the implementation of the method. This step is motivated by the fact that not all original method authors used the design pattern style, which could influence the outcome of the survey. Also, several methods were described quite abstract. Thus, the rewrite was performed to remove implementation ambiguities, too.

In the description of the methods, consequences on the system are mentioned. These consequences are partially based on the outcome of the survey, and have not been part of the participants reading material. This has been done to guarantee that the participants were not influenced in their judgment prior to the survey.

4.1.1.2 Decide Features to Evaluate

The next step is to identify the features of the methods that are evaluated. These features can be ordered in a hierarchy, where a feature can have subfeatures. This is analogous to the concept of the quality tree [BBL76].

The feature analysis method proposes several features to be evaluated. The proposed features focus on generic methods and tools. We elaborate on what a quality attribute (the feature) actually means in the context of a variability realization technique (the method).

According to Harrison & Avgeriou, quality attributes are characteristics of the system that are non-functional in nature. They characterize what the system is, rather than what it does. For example, typical quality attributes include reliability, usability, and security [HA07].

A quality attribute of a VRT is, in our opinion, no different than a quality attribute of a system: quality attributes of a VRT characterize what a VRT is, not what it does (it is extensible, usable, etcetera). However, a VRT and the variability it addresses are part of a system. A distinction can be made between:

- quality attributes of the description of the VRT;
- quality attributes of the variant system part the VRT addresses; and
- quality attributes of the whole system the VRT affects.

The method evaluation focuses on the first and the third type of quality attributes. The following feature tree is created to assess the different methods:

1. Description quality: a feature category for description-related features.
    - 1.1. Recognition: the familiarity of the method.
    - 1.2. Document readability: the readability of the documentation of the method.
    - 1.3. Memorization: the easiness of recalling the method in detail.
2. Efficiency: a feature category for efficiency-related features.
    - 2.1. Initial implementation effort: the amount of effort needed to implement the method and the initial variants as a part of the system.
    - 2.2. New variant effort: the amount of effort needed to implement a new variant as a part of the system.
    - 2.3. Documenting effort: the amount of effort needed to document the system architecture and code.

   2.4. Runtime memory usage: the runtime consumption of memory, based on a baseline system usage test.
   2.5. Runtime processor usage: the runtime usage of processor capacity, based on a baseline system usage test.

3. Maintainability: a feature category for maintainability-related features.
   3.1. Modification effort: the amount of effort needed to change an invariant part of the system to a variant part.
   3.2. Debugging effort: the amount of effort needed to debug a variant part of the system.
   3.3. Refactoring effort: the amount of effort needed to refactor a variant part of the system.
   3.4. Code structure: the structuredness of the system code.
   3.5. Code readability: the readability of the system code.
   3.6. Code complexity: the complexity of the system code.

4.1.1.3 Prioritize Features

According to the description of the feature analysis survey, the features that are most important for Océ can be prioritized. We chose not to prioritize the features, as we want every feature to weigh in equally. This has two reasons: we wish to compare the methods in an objective manner; and the weighing can be performed afterwards by Océ if necessary.

4.1.1.4 Decide Judgment Scale

Feature analysis is based on the concept of a feature being fully, partially or not supported. This turns out to be difficult to translate into a scoring scale for the features that are selected. As an alternative solution, we ask the participants comparison questions, in which the method is compared to a baseline implementation. We use a five-point Likert-scale [Lik32] to retrieve the perceived positive or negative impact of the method on the feature, compared to the baseline implementation.

4.1.1.5 Decide Level of Confidence

When performing a feature analysis, there is always a risk that the evaluation results in an incorrect conclusion. A beneficial method might be rejected or a method that is not beneficial might be accepted. The extent of the risk of a feature analysis survey is rated 'medium' by Kitchenham et al. This means that such a survey is for instance less risky than a feature analysis case study and more risky than a feature analysis experiment.

4.1.1.6 Agree on a Ranking System

The answers that are given on the Likert-scale will be translated to three possible categories. Each method can have a similar, stronger or weaker effect on a certain feature compared to the baseline implementation. The Likert-scale however covers a spectrum of five answers: much less, less, similar, more and much more. We will treat much less and less as one category and more and much more as another, as this makes interpretation more simple.

This translation into three categories is debatable. As stated by McCall, one must think twice before collapsing multiple Likert scale categories into a single category, as any form of summing or scaling of Likert scores would be influenced [McC01]. However, as we do not sum or scale the Likert scores, we expect the translation to be harmless for the results.

We propose to use the median of the different outcomes to see in which category the answers can be placed. The median is the number that separates the higher half of a set of outcomes from the lower half. Expressed differently, if more than half of the answers fall into one of the categories, the median points to that category. If no category contains more than half of the answers, the median points to the middle category, in this case the similar category. If one category contains exactly half of the answers (in our case possible, because there are an even number of participants), we will favor that category.

We can rank the different methods in two dimensions:

- Most features with a positive effect on the quality affected
- Least features with a negative effect on the quality affected

4.1.1.7 Decide Type of Survey

We propose to use a questionnaire to retrieve the answers to the comparison questions. To get a high response rate on the survey, the questions are asked during an interview with the participants.

4.1.1.8 Create Survey Documentation

The survey is created. Documentation on software variability is included.

4.1.1.9 Allocate Responsibilities

We selected eight employees of Océ to participate in the feature analysis survey. Most of these participants have a background in software development.

4.1.2 Methodological Problems

Feature analysis has several significant problems. In this section we discuss these problems.

4.1.2.1 Subjectivity

Feature analysis is based on judging methods against evaluation criteria which are identified subjectively. Such evaluations are likely to be biased and context dependent.

4.1.2.2 Too Few or Too Many Features

It is possible to identify hundreds of features that can be assessed. Assessing too many features of a specific method is very time consuming. Assessing too few features can produce a too shallow assessment. A balanced amount of features must be selected.

4.1.2.3 Inconsistency in Scoring

Different participants might have different degrees of familiarity with and understanding of the method. In addition, they might interpret the scale for a particular feature in different ways.

4.1.2.4 Collating Scores

Producing a single number from the individual scores may be misleading because many different combinations of numbers can produce the same aggregate score. Furthermore, certain features might attract higher scores than others because participants understand them better and are more able to recognize support in the method.

4.1.3 Methodological Limitations

A feature analysis survey is only viable if the participants have experience with the methods that are under evaluation. Also, it is not possible to control the experience, background and capability of the participants in the same way that you can in a formal experiment.

4.2 Survey Conduction

This section describes the execution of the survey.

4.2.1 Preparation

Each of the eight participants received an introductory document on variability, similar to section 1.2 and a copy of Appendix A. This was handed out to ensure that the participants got familiar with the survey topic and the variability techniques discussed. We left out the consequences-part of each VRT, as this would influence the answers.

4.2.2 Execution

We used a personal interview style to perform the survey. We made an appointment with each participant to take the survey. We estimated that answering each question would take approximately 30 seconds. There are six VRTs and 20 questions for each VRT, which results in an approximate survey duration of one hour.

In practice, the interviews took a little longer. Often, the participants had several questions about the techniques and the difficulty of the abstraction level of the assignment. One of the participants was not keen on taking part in the proposed survey and dropped out. The results for this participant have been removed from the survey.

4.2.3 Data Validation

It is difficult to validate the data of the feature analysis survey. It seems unreasonable to disqualify participants because their answers are not in conformance to the others. During the interview, participants were constantly asked to give their reasoning behind the answers. We did not notice any faulty reasoning, so it seems that the differences in answers can be mostly attributed to the participant's weighing of each reason.

4.3 Data Collected

This section presents the collected results of the survey.

4.3.1 Survey Answers

We want to present the data in an intuitive way. To make this possible, we transformed the answers of the participants. We did not change the answer values on recognition, document readability, memorization, code structure and code readability features, as a higher value for these features indicates a 'better' quality. However, higher values for the other features indicate a 'worse' quality. We inverted the values of these features, to make all higher values indicate a better quality and lower values a worse quality.

For each VRT, we plotted a scatter chart of the participant's response on each feature. These scatter charts are presented in Figure 4.1. We added a median line to give an indication which features score better.

Also, we created a graph for each VRT that gives the spread of the answers in three categories as proposed in section 4.1.1.6. These graphs are presented in Figure 4.2. Each part of each bar in the graph represents the number of answers in that category. The middle line is used to find the median value.

4.3.2 Description Quality

Based on the proposed scoring method in section 4.1.1.6 and the median in in Figure 4.2, the scores of the description quality features are calculated and combined in Table 4.1.

Method	Scored positively	Scored negatively
Variant class specialization	3	0
Inner entity variant	3	0
Variant code adaptation	1	1
Variant template instantiation	3	0
Variant architecture specialization	3	0
Variant component generation	3	0

Table 4.1: Description quality scores

4.3.3 Efficiency

The scores of the efficiency features are calculated and combined in Table 4.2.

Method	Scored positively	Scored negatively
Variant class specialization	1	1
Inner entity variant	0	1
Variant code adaptation	0	2
Variant template instantiation	1	2
Variant architecture specialization	1	2
Variant component generation	2	3

Table 4.2: Efficiency scores

Figure 4.1: Scatter of response on features

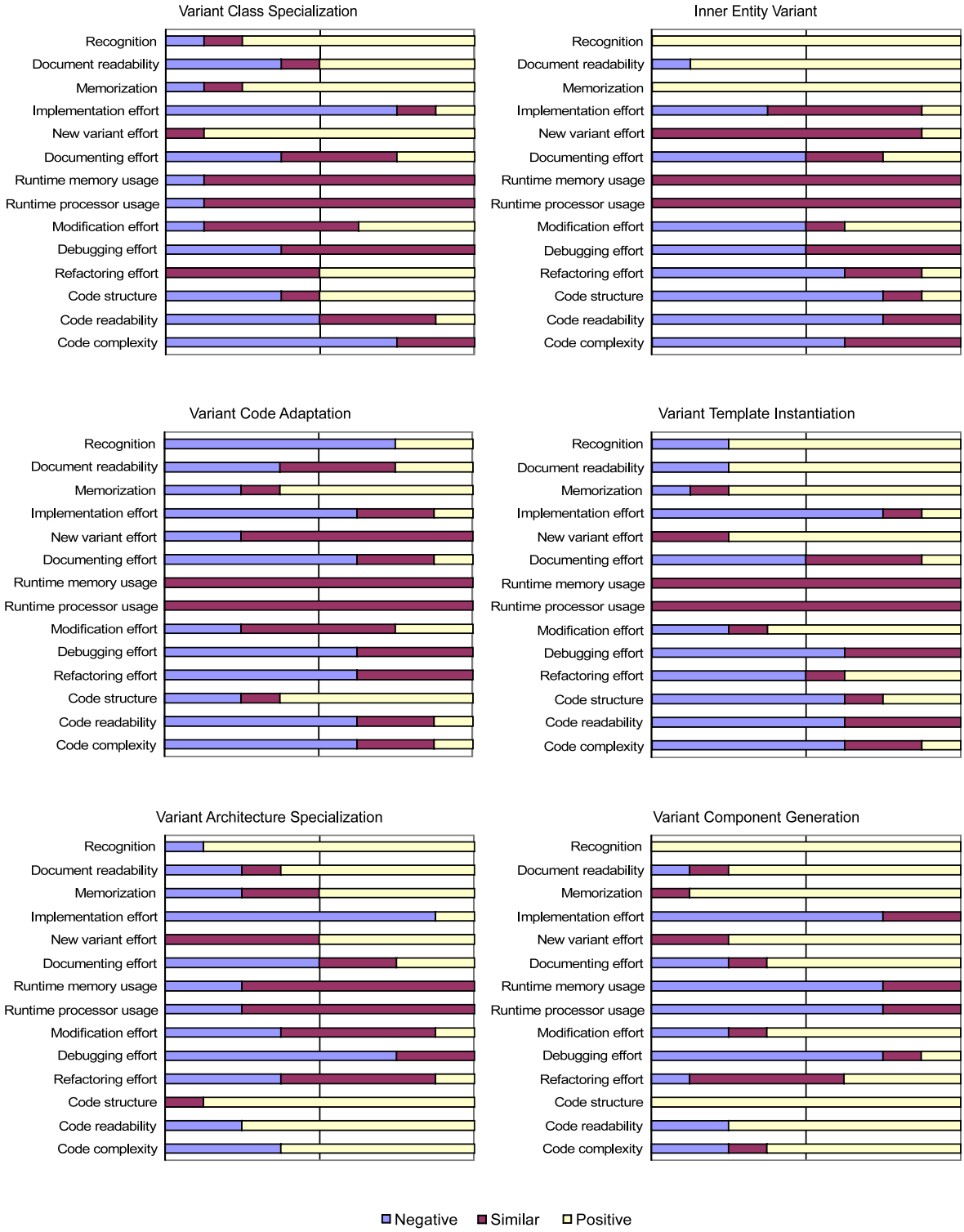

Figure 4.2: Spread of response on features

4.3.4 Maintainability

The scores of the maintainability features are calculated and combined in Table 4.3.

Method	Scored positively	Scored negatively
Variant class specialization	2	2
Inner entity variant	0	6
Variant code adaptation	1	4
Variant template instantiation	1	5
Variant architecture specialization	3	1
Variant component generation	4	1

Table 4.3: Maintainability scores

4.3.5 All Features Combined

The scores of the description quality, efficiency and maintainability are summed in Table 4.4. We used the proposed ranking from section 4.1.1.6 to determine how the methods are ranked in the two dimensions. The ordering is shown in Table 4.5 and Table 4.6.

Method	Scored positively	Scored negatively
Variant class specialization	6	3
Inner entity variant	3	7
Variant code adaptation	2	7
Variant template instantiation	5	7
Variant architecture specialization	7	3
Variant component generation	9	4

Table 4.4: Combined description quality, efficiency and maintainability scores

Method	Scored positively	Scored negatively
Variant component generation	9	4
Variant architecture specialization	7	3
Variant class specialization	6	3
Variant template instantiation	5	7
Inner entity variant	3	7
Variant code adaptation	2	7

Table 4.5: Methods sorted by most positive features

Method	Scored positively	Scored negatively
Variant architecture specialization	7	3
Variant class specialization	6	3
Variant component generation	9	4
Variant template instantiation	5	7
Inner entity variant	3	7
Variant code adaptation	2	7

Table 4.6: Methods sorted by least negative features

Chapter 5

Implementation of Techniques

We have selected two techniques to implement on top of a prototype Job Submitter: the variant class specialization and the variant component generation technique. In Table 4.5 and Table 4.6, these two techniques and the variant architecture specialization technique come out as the top three techniques. As architecture specialization is similar to class specialization, we decided to implement the more basic technique.

The following sections give an overview of the implementation of the prototype Job Submitter and the introduction of the two variability realization techniques.

5.1 Prototype Job Submitter

The prototype job submitter that is implemented is based on the original design of the job submitter that is currently produced by Océ. The Océ application already has several techniques in place to support variability. To gather objective results, we want to make sure that we start with an application which does not have any variability methods. See section 1.1.3 for an informal specification of the submitter.

The prototype is implemented using the Java Wicket-framework [wic]. We did not perform a thorough comparison of the different frameworks that exist, but merely chose a framework that we felt comfortable programming with. The Wicket framework promotes a component-based construction of user interfaces, similar to the well-known Swing library. A detailed description of the component structure is given in Figure 5.1.

Each user interface component has a separation in its client and server part. The view layer of the client part of the component consist of widgets and their layout that become part of the user interface. These widgets and layout are

specified by a HTML description that resides in the view layer of the server component part.

The controller layer of the client part consists of clientside behavior specified in the JavaScript language. This specification can be found in the server component part, next to the serverside behavior implemented in Java. The model layer of the client part consists of the value of the clientside widgets and JavaScript variables that are used by the clientside behavior. The server part model layer consists of the persistent `TicketModel` object that contains all data relevant to the job ticket.

The framework takes care of the communication between the server and client part of each component. Also, it provides default implementations of the clientside and serverside behavior, to ease the roundtrip of data between the clientside and serverside model layer.

The server side part of the application also contains a `JobTicketBuilder` that can construct a job ticket specification from the `TicketModel` data. When the user presses the print button in the user interface, the user input is validated. When the data is valid, the `TicketModel` is updated, a job ticket is generated and sent to the LPR interface of the printer.

5.2 Introducing Variability in the Job Submitter

We will introduce variability in the prototype job submitter, by the process laid out in section 1.2.

5.2.1 Identification of Variability

The variant feature of the job submitter application is the support for a set of printers. We started with a bottom-up approach to identify the need for variability.

In the boundary layer of the prototype application, the `LPR` class for accessing the printer is used for all printers that have an LPR interface. This LPR interface is an established standard interface for communication with many Océ printer systems. For this prototype implementation, we did not make this interface variable.

However, the `JobTicketBuilder` needs to be able to produce different tickets for each printer. Even though there is a job ticket standard, the printers require job tickets to be expressed in a printer specific version of the standard. And also, the printers differ in functionality; for instance, in printable formats and folding capabilities.

The `TicketModel` also needs to be variable. As the `TicketModel` is the data model of the actual capabilities of the printer, differing capabilities have influence

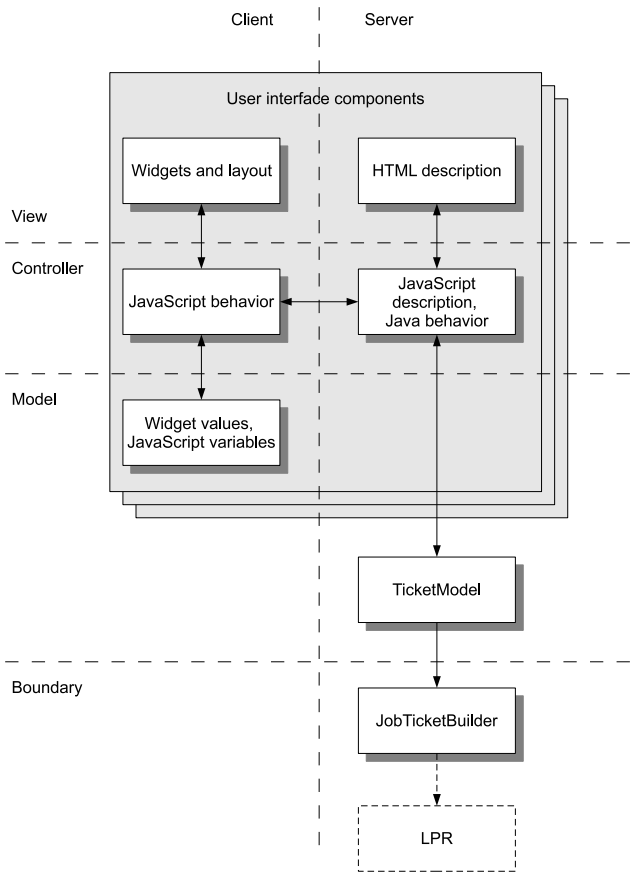

Figure 5.1: Component architecture of Job Submitter

on the data model. For instance, if the printer is capable of folding, then one or more model properties need to be present to store the folding settings.

The `TicketModel` consists of a `TicketSettings` object. This object contains all the settings that are relevant to this printer as properties, each with a getter and setter. These properties are of type `String`, `int` or `double`. Also, this object contains a list of `FileSettings` objects. This list contains the filenames and file data of the files submitted to the printer.

The user interface components that contain the widgets that are bound to the global settings need to vary too. Again, if a printer has a unique capacity, this needs to be reflected in the user interface. For this, the HTML description of the components needs to be varied. Also, the Java behavior of the widgets in the user interface needs to be varied.

Some of the widgets in the user interface are limited to a selection from a list of options. These lists need to vary too; for instance, to represent different printable

formats. We have modeled these lists of options as part of the `TicketModel`.

5.2.2 Deciding on Extent

We want to manipulate the variability before we build a product. Also, we want the possibility of building a variant product that does not contain other variant leftovers. For this, we require VRTs that can introduce and populate the variants during development and can bind the variant before runtime. The six VRTs all qualify for these conditions.

5.2.3 Implementation of Variability

For the implementation, we selected the variant class specialization and component generation technique. The implementation details for these techniques can be found in Appendix A. We implemented two variants of the prototype job submitter, based on two different printers.

5.3 Variant Class Specialization Implementation

We started with the prototype application that is fully functional for use with a single printer. We added an abstract class `VariantFeature` to the application that is implemented for each variant.

Our two variant classes are named `VariantA` and `VariantB`. Also, we added a mechanism to the persistent application object to instantiate and get the bound variant object. The selection of the variant object is done by adding a class name to a configuration file and instantiating the object at startup. The build process is altered to only compile and package the variant class file specified in the configuration file.

5.3.1 Model Layer

We started to add a variation point to the `Settings` class that contains several static maps of string pairs. Each pair consists of a widget value and its textual representation in the user interface.

One of the maps is a list of print modes. The following listing shows a shortened `Settings` class. The `printmodes` map is a class property and is initialized using a static code block.

Listing 5.1: Settings.java

```
public class Settings {
  public static Options printmodes = new Options();
  ...

  static {
```

```
    printmodes.put("auto", "Oce Print Assistant");
    printmodes.put("draft", "Economy");
    printmodes.put("normal", "Production");
    printmodes.put("enhanced", "Presentation");
    ...
  }
}
```

We want to make this map of print modes variant in each product. The static code path is changed by adding a method to the abstract class, overriding the method in the variant class and moving the code path to this method. The following listing reflects the changes.

Listing 5.2: Settings.java with variation points

```
public class Settings {
  public static Options printmodes = new Options();
  ...

  static {
    JobSubmitterApplication.getVariantFeature().static_Settings();
    ...
  }
}

public abstract class VariantFeature {
  public void static_Settings() {}
  ...
}

public class VariantA extends VariantFeature {
  public void static_Settings() {
    Settings.printmodes.put("auto", "Oce Print Assistant");
    Settings.printmodes.put("draft", "Economy");
    Settings.printmodes.put("normal", "Production");
    Settings.printmodes.put("enhanced", "Presentation");
  }
  ...
}
```

Another map we want to make variant is the map of color modes. The map of color modes is only relevant to `VariantA`, as this printer supports printing in color. `VariantB` only supports black-and-white, and thus does not need the property.

We have noticed a limitation of the variant class specialization technique that is not mentioned in literature. It is not possible to make the presence of class properties variant. Java does not have a language construct for dynamically adding properties, as far as we know. Java supports dynamic manipulation of the value of a property through reflection, but it is not possible to add a property to a class.

A solution is to create a map of maps that represents the properties of this class and can be dynamically adjusted to contain any set of maps necessary by the variant. However, this would reduce the code understandability. Also, less static code checks could be performed.

An alternative solution is to move the variant settings properties to the variant class. This would require changes to the places in the application code that refer to these specific variant settings properties, as they would need to be accessed via the variant class.

Another solution is to make the whole `Settings` class variant. This would require even more changes in the application code. All places where the `Settings` are accessed directly via the static properties would need to be refactored to use the variation point code.

We chose to add the superset of all variant properties to the invariant class, even though this means that some properties will not be used and the invariant class becomes cluttered with properties.

In the `TicketSettings` class, the same limitation applies. The following listing shows a shortened `TicketSettings` class.

Listing 5.3: TicketSettings.java

```
public class TicketSettings {
  private String colormode;
  ...

  public String getColormode() {
    return colormode;
  }
  public void setColormode(String colormode) {
    this.colormode = colormode;
  }
  ...
}
```

In this class the `colormode` property and its getter and setter methods would be a candidate for making variant. However, as we have stated above, this is not possible using this technique. Therefore, we chose to use the same workaround and add the superset of all variant properties and methods to the invariant `TicketSettings` and `FileSettings` classes.

5.3.2 Controller Layer

We continued implementing variation points in the Java classes that contain the component hierarchy of the application. There are two classes that need to be partially variant, namely the `SettingsCategory` and `WorkflowCategory` class.

These classes are coupled with the HTML description by the framework to generate the web page. The framework utilizes the composite pattern to cre-

ate the component hierarchy [GHJV95]. The following listing gives a shortened `SettingsCategory` class.

Listing 5.4: SettingsCategory.java

```
public class SettingsCategory extends Panel {
  public SettingsCategory(String id, IModel model) {
    super(id, model);

    add(new CatRadioChoice("colormode", bind("colormode"),
      Settings.colorModes));
    ...
  }
}
```

The class extends the `Panel` framework class to indicate that it needs to be coupled with a HTML description. The constructor arguments `id` and `model` are passed through to the framework super class. The `id` string is used to couple this component with a HTML element, the `model` is a wrapper that points to our `TicketSettings` object.

In the constructor code, a new object is added to the component hierarchy. The `CatRadioChoice` component is a custom component, based on the default framework components, and contains a row with a label and a radio choice widget to select a certain option. The `bind` method couples the widget value of the radio choice to the `colormode` model property. The possible radio choice options are retrieved from the static `Settings` class.

We want to make this `CatRadioChoice` component variant. We again added an abstract method to the `VariantFeature` class, implemented the method in the `VariantA` class and moved the code to this new method. The following listing reflects these changes.

Listing 5.5: SettingsCategory.java with variation points

```
public class SettingsCategory extends Panel {
  public SettingsCategory(String id, IModel model) {
    super(id, model);
    JobSubmitterApplication.getVariantFeature().
      constructor_SettingsCategory(this);
    ...
  }
}

public class VariantA extends VariantFeature {
  public void constructor_SettingsCategory(SettingsCategory cat) {
    cat.add(new CatRadioChoice("colormode", cat.bind("colormode"),
      Settings.colorModes));
  }
  ...
}
```

We used the method parameter to pass a reference to the `SettingsCategory` component. This makes it possible to manipulate the component hierarchy and access the model in the variant code. In a similar fashion, we treated the `WorkflowCategory` class.

5.3.3 View Layer

We continued looking for variation points and found several in the HTML description of the web page. The following listing gives a shortened HTML description that is coupled to the `SettingsCategory` component.

Listing 5.6: SettingsCategory.html

```
<wicket:extend>
  ...
  <span wicket:id="colormode"></span>
  ...
</wicket:extend>
```

The framework uses the `wicket:id` attribute to couple any HTML element to its Java behavior. The custom `CatRadioChoice` component produces HTML output that will be added as a child to this `span` tag.

The variant class specialization technique does not have a solution for variant non-code artifacts, such as text fragments. As the prototype has several HTML fragments that need to be varied, a solution would be to generate the HTML at runtime. However, the framework explicitly uses static HTML fragments to describe the HTML contribution of each component.

To make these HTML fragments dynamic would oppose the methodology of the framework and reduce understandability. A solution might be to use another VRT that is able to vary non-code, such as variant template instantiation. However, we chose not to introduce another VRT.

We chose to add the superset of all variant HTML fragments to the invariant HTML description, even though this means that some HTML will not be used and the description becomes cluttered with fragments. As each component represents only a single tag in the invariant HTML description, we think that the amount of clutter is minimal.

5.3.4 Domain Layer

Finally, we added variation points to the `JobTicketBuilder` class. The class contains a `getOutput` method that accepts the `TicketSettings` model object reference as a parameter. The method creates a string buffer that is used to construct the job ticket and returns the contents of the buffer as a plain string. The following listing gives a shortened overview of the class.

Listing 5.7: JobTicketBuilder.java

```
public class JobTicketBuilder {
  public String getOutput(TicketSettings ticket) {
    StringBuffer sb = new StringBuffer();
    sb.append("BeginTicket 2.5\n");
    ...
    for (FileSettings file : ticket.getFiles()) {
      ...
      sb.append("OrigName \"" + file.getFilename() + "\"\n");
      ...
    }
    ...
    return sb.toString();
  }
}
```

A solution to make this output variant is to make all of the code of the `getOutput` method variant with a single variation point. Duplicate code in each variant can then be reduced by moving it to protected methods in the abstract `VariantFeature` class.

However, we chose to only move the variant code parts to the variant class. Moving invariant parts to the variant class is an unnecessary step and more effort intensive. We identified ten variation points in the `JobTicketBuilder` class, based on the differences between the job tickets. The following listing gives a shortened overview of the implementation.

Listing 5.8: JobTicketBuilder.java with variation points

```
public class JobTicketBuilder {
  public String getOutput(TicketSettings ticket) {
    StringBuffer sb = new StringBuffer();
    sb.append("BeginTicket 2." + JobSubmitterApplication.
      getVariantFeature().beginTicket_JobTicketBuilder() + "\n");
    ...
    for (FileSettings file : ticket.getFiles()) {
      ...
      JobSubmitterApplication.getVariantFeature().
        origName_JobTicketBuilder(sb, file, ticket);
      ...
    }
    ...
    return sb.toString();
  }
}

public class VariantA extends VariantFeature {
  ...
  public String beginTicket_JobTicketBuilder() {
    return "5";
  }
```

```
  public void origName_JobTicketBuilder(StringBuffer sb,
    FileSettings file, TicketSettings ticket) {

    sb.append("OrigName \"" + file.getFilename() + "\"\n");
    ...
  }
}
```

To manipulate the output of the ticket builder from the variant parts, we sometimes passed a reference to the string buffer, as seen in the `origName_JobTicketBuilder` method. We also sometimes concatenated the return value of the method to the parameter of the string buffer, as seen in the `beginTicket_JobTicketBuilder` method.

5.3.5 Variant B

The implementation of the second variant went much quicker than the first. This can be attributed to the fact that the initial setup was already performed, many variation points were already in place and we were much more experienced with the technique.

We added the second variant's specific settings to the `Settings` class using the existing variation point. We already added the unification of all variant properties and methods to the `TicketSettings` and `FileSettings` class. We reused the variation point in the `SettingsCategory` and `WorkflowCategory` to add specific components for the second variant. The HTML fragments already contained the unification of all possible components. We used a capability of the framework to hide the components from the view layer that were not relevant for the second variant.

As we already compared the different job ticket behaviors, we had added enough variation points to the `JobTickerBuilder` class to make the behavior of the second variant possible without the addition of extra variation points. We believe that this is not usually the case in practice, as the different implementations are often not available beforehand.

5.3.6 Evaluation

We identified the limitation of not being able to make the existence of properties and methods of classes variant, as discussed in section 5.3.1. This limitation is specific for this technique and we believe that the workaround solutions can lead to less maintainable code.

As we have seen in the variant model and controller code, we needed to pass a reference to invariant objects if we want to manipulate these from variant code. This creates a coupling between the variant and invariant code and thus makes

the variant code not fully separate from the invariant code. Changes to the invariant code could influence the variant code and vice versa, making it more difficult to modify and maintain the application. We believe that this is not a specific problem of the technique used, but rather a problem with variability in general.

If full separation means never passing any data or objects between invariant and variant code, then a full separation is only possible when variant code can only access and manipulate variant data. However, in this implementation we have chosen to use an invariant superset of data properties as data model that is shared with the variant code. This choice automatically leads to coupling. It is impractical, perhaps even impossible to reach perfect separation.

We have noticed that, especially when there are many variation points closely located to each other, the code gets less comprehensible using this technique, as the flow of code is interrupted by the variation points and harder to understand. The straightforward process of appending strings in the ticket builder is more difficult to follow, even with the help of a proper IDE. And if several variant features are introduced that would crosscut this and each other's variant code, we expect the code to become even more incomprehensible.

5.4 Variant Component Generation

We started again with the prototype application that supports a single printer. We already identified the classes that contain variant code. We set up the openArchitectureWare (oAW) tool to generate two different variations of the prototype application [ope]. Each variation is based on a description written in a custom, domain specific language. We changed the build process to copy and package the invariant artifacts and generate the variant artifacts from the DSL description that is specified in a configuration file. At this point, nothing is generated yet, as we have not defined our DSL.

5.4.1 Model Layer

We want to generate the whole `Settings` class using our custom DSL. oAW has its own specification language to define a DSL, which is similar to the ANTLR syntax for describing languages. In the following listing, the relevant specification for our DSL is given.

Listing 5.9: DSL specification for option lists

```
Model:
  "model" "{" (jsmodels+=JSModel)* "}";

JSModel:
  Options | ...;
```

```
Options:
  "options" "{" (options+=Option)* "}";

Option:
  name=ID "{" (pair+=KeyValuePair)* "}";

KeyValuePair:
  key=STRING ":" value=STRING ";";
...
```

The specification links the DSL code representation to the abstract syntax tree nodes that are created. Our DSL syntax tree starts with a single `Model` node which is represented by the `model` string with brackets. Between the brackets, representations of multiple `JSModel` nodes are allowed. These nodes will become children of the `Model` node. A `JSModel` node is subclassed by an `Options` node and others that are left out for brevity.

An `Options` node is represented by the `options` statement with brackets and can have multiple `Option` child nodes. An `Option` node is represented by a name, which becomes the `ID` attribute of the node, and by a representation of multiple `KeyValuePair` nodes. And a `KeyValuePair` node is represented by a string, which becomes the `key` attribute, followed by a colon, followed by a string, which becomes the `value` attribute, followed by a semicolon.

oAW generates an Eclipse editor plugin based on this specification. The editor plugin can be added to Eclipse, which makes it possible to edit code written in the DSL we just specified, including syntax highlighting and code hints. We created the following DSL code in the editor.

Listing 5.10: DSL code for option lists

```
model {
  options {
    printmodes {
      "auto" : "Oce Print Assistant";
      "draft" : "Economy";
      "normal" : "Production";
      "enhanced" : "Presentation";
    }
    ...
  }
  ...
}
```

This enabled us to generate the variant code for the `Settings` class. oAW contains a generator tool that accepts code written in a custom DSL, creates an abstract syntax tree model from the code based on the DSL and uses a oAW-specific template language to transform the tree model into actual code. The following listing gives the relevant template design.

Listing 5.11: Template for Settings.java

```
«DEFINE main FOR Model»
   «EXPAND model FOREACH entities.typeSelect(Options)»
«ENDDEFINE»

«DEFINE model FOR Options»
«FILE "com/oce/jobsubmitter/model/Settings.java"»
...
public class Settings {
  «EXPAND option FOREACH options»

  static {
    «EXPAND staticOption FOREACH options»
  }
}
«ENDFILE»
«ENDDEFINE»

«DEFINE option FOR Option»
public static Options «name» = new Options();
«ENDDEFINE»

«DEFINE staticOption FOR Option»
«EXPAND pair(name) FOREACH pair»
«ENDDEFINE»

«DEFINE pair(String name) FOR KeyValuePair»
«name».put("«key»", "«value»");
«ENDDEFINE»

...
```

The generator starts at the top of the syntax tree, which contains the `Model` node. Using the `typeSelect` method, all child `Options` nodes are selected. The `FOREACH` operator iterates over each `Options` node. The `EXPAND` statement forwards the node to a template that matches the `Options` type with the `model` identifier.

The `Options` node is passed to the second template. Here, the `FILE` statement creates a new file in the specified path. The text contents of the template is written to the file. Both `EXPAND` statements are replaced by their parsed contents, and so on. Based on these templates, the generator produces a `Settings` class identical to the code in Listing 5.1.

We also want the properties and methods in the `TicketSettings` class to be variant. Unlike the variant class specialization technique, we are not limited to make only method contents variant. The following listing gives shortened DSL code that is used to generate the `TicketSettings` class.

Listing 5.12: DSL code for settings model

```
model {
  global {
    colormode string "color";
    ...
  }
}
```

The `global` string is accepted as part of the `model` block. Each line between the `global` brackets separated with a semicolon represents a property of the `TicketSettings` class. The `colormode` string is used as a property name in the template that transforms the syntax tree to the generated code. This template is left out for brevity. The `colormode` property will be of type `String` and has the default value `color`. The generated code is identical to the code in Listing 5.3. We used the same approach for the `FileSettings` class.

5.4.2 Controller and View Layer

We continued to extend our custom DSL to generate the view HTML description and controller code. The following listing gives the shortened DSL code for these layers.

Listing 5.13: DSL code for settings controller and user interface

```
view {
  settings {
    ...
    radio[colorModes] >> colormode;
    ...
  }
  ...
}
```

We added a top-level string `view` to represent the controller and view model. The `settings` block contains the configuration of the `SettingsCategory` class and HTML. Each line in the `settings` block represents a specific component in the user interface. The order of these lines matters; the components will appear in the same order in the user interface.

The line in the above listing represents a component that allows a selection via a radio button group. The component is parameterized by a list of choices. These are put between square brackets. The `colorModes` list must refer to a list created in the model layer, as was done in Listing 5.10. This constraint is expressed in the oAW constraint language, which will automatically be statically checked in the generated DSL editor. The component will put the valid input of the user in the `colormode` property of the `TicketSettings` object. The property must exist in the settings model. This is also expressed as a constraint.

Based on the above DSL code, the `SettingsCategory` class file and HTML description can both be generated. The resulting files are identical to the code in Listing 5.4 and Listing 5.6.

5.4.3 Domain Layer

For the generation of the `JobTicketBuilder` class, we examined the common types of code and operations in this class. Most of the operations are additions of a line to the `StringBuffer`. These strings often consist of a static part concatenated with values from the properties of the settings model. A special case is the file list in the settings model: for each file, a set of lines are added to the ticket.

Different cases of output are handled by using if-else statements that accept boolean expressions. The model properties can also be manipulated through these expressions. Several operators are used, for instance a `contains` operator, which checks if a string contains a certain value, and several number manipulation operators.

Based on this subset of Java language constructs, we created a DSL specification that contains these operations. The following listing gives shortened DSL code that is used to generate the `JobTicketBuilder` class.

Listing 5.14: DSL code for job ticket builder

```
ojt {
  "BeginTicket 2.5";
  "JobName \"" + jobname + "\"";
  ...
  forEachFile {
    ...
    "Segment " + file.counter;
    "OrigName \"" + file.filename + "\"";
    ...
  }
  ...
  if scale = "NO_ZOOM" {
    "Zoom 100";
  } else {
    "AdjustZoom tofit";
  }
  ...
  if rotation startsWith "auto_mediasaving" {
    if (  flip = "yes"  &   rotation contains "extra180") |
       (!(flip = "yes") & !(rotation contains "extra180") ) {
      "AdjustRotate mediasaving";
    } else {
      "AdjustRotate mediasaving extra180";
    }
  }
```

```
  ...
}
```

Each line in the `ojt` block is an expression and contributes to the final ticket, except for the special statements `forEachFile` and `if`. We do not want the type of the referred properties to matter. The properties should be dynamically converted based on the used operands.

In the `forEachFile` block, there is a special syntax to refer to properties of the current `file` object. Also, a counter variable should be available that can be added to the ticket output, or used in an `if`-statement.

The above DSL code generates the following code for the `JobTicketBuilder` class.

Listing 5.15: Generated JobTicketBuilder.java

```
...
public class JobTemplate {
  public boolean eq(String e1, String e2) {
    if (e1 == null && e2 == null) {
      return true;
    } else if (e1 == null && e2 != null) {
      return false;
    }
    return e1.equals(e2);
  }
  public boolean startsWith(String e1, String e2) {
    if (e1 == null || e2 == null)
      return false;
    return e1.startsWith(e2);
  }
  public boolean contains(String e1, String e2) {
    if (e1 == null || e2 == null)
      return false;
    return e1.contains(e2);
  }
  ...
  public String getOutput(TicketSettings ticket) {
    StringBuffer sb = new StringBuffer();
    sb.append("BeginTicket 2.5");
    sb.append('\n');
    sb.append("JobName \"" + ticket.getJobname() + "\"");
    sb.append('\n');
    ...
    for (FileSettings file : ticket.getFiles()) {
      int count = ticket.getFiles().indexOf(file) + 1;
      sb.append("Segment " + count);
      sb.append('\n');
      sb.append("OrigName \"" + file.getFilename() + "\"");
      sb.append('\n');
      ...
```

```
    }
    if (eq(ticket.getScale(), "NO_ZOOM")) {
      sb.append("Zoom 100");
      sb.append('\n');
    } else {
      sb.append("AdjustZoom tofit");
      sb.append('\n');
    }
    ...
    if (startsWith(ticket.getRotation(), "auto_mediasaving")) {
      if ((eq(ticket.getFlip(), "yes") &&
          contains(ticket.getRotation(), "extra180")) ||
          (!(eq(ticket.getFlip(), "yes")) &&
          !(contains(ticket.getRotation(), "extra180")))) {
        sb.append("AdjustRotate mediasaving");
        sb.append('\n');
      } else {
        sb.append("AdjustRotate mediasaving extra180");
        sb.append('\n');
      }
    }
    ...
  }
}
```

In the generated code, several problems needed to be solved. For instance, the dynamic type conversion of the properties is not straightforward. The conversion code is left out for brevity, but has a significant impact on the amount of code for this class.

5.4.4 Variant B

The implementation of the second variant also went much quicker than the first, just as the variant class specialization technique. This can again be attributed to the fact that the initial setup was already performed, the variation points were in place and we were much more experienced with the technique.

We started with a copy of the DSL specification. We removed the specification details that were specific for the first variant, and added the specification details for the second variant. We quickly were able to generate the `Settings`, `TicketSettings`, `FileSettings`, `SettingsCategory` and `WorkflowCategory` related files.

For the `JobTicketBuilder` class we needed to extend the DSL. The first variant did not require the `startsWith` string operator. However, the second variant required it for a certain `if` statement, which we missed during our comparison. We added the `startsWith` operator to the DSL specification. Then, we extended the code template to generate the correct Java code. This process was well supported by the oAW tooling. After this change, we were able to generate the

second variant of the `JobTicketBuilder` that was nearly identical to the original ticket builder implementation.

5.4.5 Evaluation

We have noticed that code generation can create exactly the code necessary for a single variant. It is not necessary to use the suboptimal solution of creating a superset of properties, as we have done with the variant class specialization. This increases the readability and comprehensibility of the code. Also, the model layer DSL contains only a minimal set of statements and identifiers. This makes modification of the model classes very efficient in comparison with refactoring the classes using an IDE.

The dynamic type conversion problem that needed to be solved in the generated code is a good example of a more generic problem when incorporating a DSL. It is the goal of a DSL to be able to write a specification or code using domain specific abstractions. However, the more the DSL abstracts from the actual produced code, the more code a developer has to write to interpret the specific abstractions. It is unclear what the severity of this problem is in practice.

Also, it is questionable that this ticket builder DSL code is significantly more comprehensible than the actual code. The DSL code highly resembles the end result and only leaves out a few statements and language constructs. However, we do believe that a variant written in the DSL code is better maintainable than a variant based on the variant class specialization technique. Especially the speed of modifying a variation of the ticket builder code is a crucial argument of implementing it using this technique.

Chapter 6

Discussion

This chapter discusses the various consequences of the variability realization techniques.

6.1 Variant Class Specialization

We discuss the consequences of the variant class specialization technique based on the results of the survey, relevant literature and our experience implementing this technique.

6.1.1 Consequences From Survey

For each category of features, the consequences are discussed.

6.1.1.1 Description Quality

This technique is recognized as a method to implement variability. The initial documentation of the technique is considered too abstract and could use several examples. The technique is considered highly memorable. The well-known design patterns that are used for construction might explain this.

6.1.1.2 Efficiency

Substantial developer effort is needed for the initial implementation of this technique. However, most of the participants believe that this technique makes the creation of new variants more efficient.

This technique is considered to cost no significant extra documentation effort. It might be that the technique is considered to be well known, which lowers the need to explain the functionality of the technique in extra documentation.

The memory and processor usage of the final product is assumed not to be significantly affected, even though this technique introduces a mechanism that has a slight effect on memory and performance at runtime.

6.1.1.3 Maintainability

Modifying an invariant part to become a variant part is assumed to cost a similar amount of effort compared to modifying copied code, even though there is some overhead to modify the abstract class and its implementation classes.

Debugging the system with this technique incorporated is considered as difficult as debugging a system that does not have such a technique. The introduced variation point method calls are not considered problematic for debugging.

Refactoring the variant code of this technique is assumed to cost less effort. This might be explained by the familiar design pattern structure of the variant. Also, the variants are contained in classes which can be easily refactored by most IDEs.

It is assumed that this technique increases the structure of the code. The introduction of a set of structured classes by this technique can account to this assumption.

A system with this technique incorporated is regarded to have less readable code. Readability might be considered as how easy the flow of programming code can be followed. The flow of code is explicitly altered by this technique to move the variant code into separate classes, which might explain lower readability.

The final system is assumed to have higher code complexity. The participants might have considered the extra classes introduced to influence the complexity of the code.

6.1.2 Consequences From Literature

The variant component specialization technique [SvGB05] described by Svahnberg et al. is a little different. The technique does not use an abstract class. Instead it only uses the interface, which results in implementing all methods in each variant class instead of overriding only the necessary changed methods.

Svahnberg et al. focus on several consequences. They too regard the extra cost of processing power due to the layer of indirection minimal. They think the creation of an interface for the variant part is not always easy. This would affect the initial implementation effort negatively.

They also believe that the readability of the code is hindered when an excess of method calls is made to the variant code, especially when there are a lot of calls to variant methods. Also, they note that code duplication of each variant class happens often, which would imply that modification and refactoring of variant classes is more difficult and thus less efficient.

However, they think that the technique facilitates the creation of new variants and maintainability of the variants in general. How this technique facilitates these qualities is not stated.

6.1.3 Consequences From Implementation

Based on our implementation in section 5.3, we note several consequences for the efficiency and maintainability features.

6.1.3.1 Efficiency

We noticed that the initial implementation of this technique can be done incrementally. This eases the transition to a full implementation of this technique. Adding support for a new variation point is straightforward: create a new method in the abstract variant class, add a call to the method from the invariant code and override it as needed in each variant class.

If there are more variants that need the same method behavior, it is tempting to add the code to the method in the abstract class and override it in a specific variant case. However, this would affect the readability of the code negatively. We found out that it is better to add another indirection layer by adding protected methods to the abstract class and calling these from the variant classes. We consider this more clearly from a readability viewpoint.

From our experience, we would say that the initial implementation is indeed less efficient in terms of developer effort. We thought of several solutions to separate the variant from the invariant code and settled with a solution that did not fully separate the variant and invariant parts. We think that the less optimal implementation of the technique outweighs the consequences of fully separating the parts.

Adding a new variation was also straightforward. To create a new variant without introducing new variation points costs a similar amount of effort as copying and altering code. We found that the variant class structure helped reduce the amount of implementation considerations: there are less ways of implementing changes. This implies a substantial effort gain when many variants are implemented.

However, we found that it was often needed to add new variation points to the classes to support a specific change. We conclude that this technique does not significantly reduce the effort needed to add a new variant when compared to copying code.

During the transition and creation of new variants, we tried to minimize the amount of duplicate code in the variant classes. This was done by creating components outside the variant classes. These components are extensions of the application framework. We regard these components as part of the variant shared

code, as they were only used in the variant code. However, these could also be reused in the invariant code.

We did not look into documenting effort and runtime memory and processor usage during the prototype implementation.

6.1.3.2 Maintainability

The refactoring work to realize these components is properly supported by the IDE. However, the framework helped here too, as the framework supports creating reusable components extensively. This is done by extending existing components that override certain behavior.

We often needed to debug a certain variant user interface behavior. As it is possible to filter a part of the code during debugging, we did not have trouble debugging the variant classes. However, this can also be partly attributed to our choice of using a runtime variant selection mechanism. This made it easy to trace behavior back to its originating class file.

We found that the code complexity did not increase by utilizing this technique. The code paths were longer due to the layers of indirection, but the amount of different paths possible did not increase for a single variant. However, the readability of the code did decrease. Especially the invariant parts of the code were harder to understand, because the extra method calls to the variant class made the behavior less comprehensible. We think that the code structure was positively influenced by the separation of the variant and invariant parts.

6.1.4 Connecting Together

As we stated in our own experience with implementing a new variant, we think that there is no significant effort gain compared to duplicating and modifying code, contrary to the results of the survey and literature. At most, the variant is implemented without significant extra effort, but never with lower effort.

Also, we think that code complexity does not increase using this technique, contrary to the results of the survey. We believe that the quality might have been misinterpreted.

6.2 Inner Entity Variant

We discuss the consequences of the inner entity variant technique based on the results of the survey and relevant literature.

6.2.1 Consequences From Survey

For each category of features, the consequences are discussed.

6.2.1.1 Description Quality

This technique is recognized as one of the oldest methods to realize variability. The technique is considered highly memorable. This might be due to the simple character of the technique.

6.2.1.2 Efficiency

The initial implementation does not cost significant extra effort. Also, a new variant can be created with similar effort compared to the adaptation of copied code. This can also be explained by the simple character of the technique. It can also be attributed to the high familiarity of the participants with this technique.

This technique is considered to cost more documentation effort. This can be attributed to the extra technique details that need to be explained in the documentation.

The memory and processor usage of the final product is assumed not to be significantly affected. With this technique, any trace of the mechanism disappears and only the code of a single variant appears in the final product.

6.2.1.3 Maintainability

Modifying an invariant part to become a variant part is assumed by most participants to cost more effort for this technique.

Debugging the system with this technique incorporated is more difficult than debugging a system that does not have such a technique. The code to be debugged has been preprocessed, which makes it harder to trace back bugs to their originating code entities.

Variants of this technique are considered to require more effort to refactor. The unavailability of automatic refactoring tools might explain this.

It is assumed that this technique decreases the structure of the code. A system with this technique incorporated is regarded to have less readable code. Also, such a system is assumed to have higher code complexity.

6.2.2 Consequences From Literature

Svahnberg et al. focus on two consequences for the technique they call condition on constant [SvGB05]. First, they believe that the number of potential execution paths tends to explode when using #IFDEFs. This would make maintenance and bug fixing of the system difficult.

Secondly, Svahnberg et al. have seen that variation points often tend to scatter throughout the system. This would make it difficult to keep track of what parts of a system are actually affected by one variant.

6.2.3 Connecting Together

We disagree with only this technique suffering from a potential execution path explosion, as mentioned in literature. This problem affects all techniques. If the amount of variants increases, the amount of possible execution paths increases.

However, this technique combines the different variant implementations in the same software entity, which leads to an undesirable decrease of code readability. Thus, the problem here is not the execution paths explosion, but the variation point mechanism and its container. This technique is not well suited when many variation points are close to each other.

We question the survey result on invariant to variant modification effort. We believe that this method provides the most basic of variation mechanisms and that the effort of adding a variation point is negligible when adopting this technique.

We also question the survey result on code structure. Perhaps the participants mixed up the concept of structure and comprehensibility. A system with this technique incorporated can be considered less comprehensible as all variant code is contained in the same entity.

6.3 Variant Code Adaptation

We discuss the consequences of the variant code adaptation technique based on the results of the survey and relevant literature.

6.3.1 Consequences From Survey

For each category of features, the consequences are discussed.

6.3.1.1 Description Quality

This technique is not recognized as a method to implement variability. It might be that the lower readability influenced the recognition of the experts. However, during the interview, the participants stated that they had not seen the variant code adaptation technique before.

That the participants are not familiar with the technique can also be explained by the missing Java language background knowledge of most of the participants; the technique uses aspect oriented programming methodologies that are mainly utilized in the Java language.

The initial documentation for this technique is considered to be too abstract and would benefit from a concrete example. However, the participants believe that this method is highly memorable.

6.3.1.2 Efficiency

Most participants agree on the substantial extra developer effort needed for the initial implementation of this technique. The participants might have accounted for the learning curve and setup of AspectJ. A new variant can be created with similar effort compared to the adaptation of copied code.

This technique is considered to cost more documentation effort. This can be explained by the extra technique details that need to be explained in the documentation.

The memory and processor usage of the final product is assumed not to be significantly affected. With this technique, any trace of the mechanism disappears and only the code of a single variant appears in the final product.

6.3.1.3 Maintainability

Modifying an invariant part to become a variant part is assumed to cost a similar amount of effort compared to modifying copied code.

Debugging the system with this technique incorporated is more difficult than debugging a system that does not have such a technique. The code to be debugged has been preprocessed, which makes it harder to trace back bugs to their originating code entities. The participants might not have taken IDE debugging support for AspectJ into account.

Variants of this technique are considered to require more effort to refactor. The unavailability of automatic refactoring tools might explain this.

It is assumed that this technique increases the structure of the code. A system with this technique incorporated is regarded to have less readable code. Also, such a system is assumed to have higher code complexity.

6.3.2 Consequences From Literature

The code fragment superimposition technique [SvGB05] described by Svahnberg et al. is very similar to the variant code adaptation technique.

They state that a consequence of utilizing this technique is the full separation of variant behavior from invariant functionality. Svahnberg et al. think that this increases the readability of "what the source code is intended to do".

However, they think that by using this technique, it becomes harder to understand how the final system will work since the execution path is no longer obvious. A developer must be aware that the variant code is superimposed in the invariant part at a later stage, and that this superimposition process has limitations and can break too.

Braganca et al. state that code composition techniques are not widely accepted and used in the industry, because most of these techniques are based on recent academia research [BM04a].

6.3.3 Connecting Together

The statement of Braganca et al. on the acceptance of this technique in the industry is confirmed by the survey results, as this technique is also not well known by Océ experts.

Svahnberg et al. think that this method increases the readability of the system code. However, the results of the survey object to this claim. Most participants believe that the code of the system becomes less readable with this technique in place. Participants might have considered that the final system code can always be affected by the code adaptation, which influences the understanding and thus the readability of the code negatively.

Svahnberg et al. also state that this technique leads to full separation of the variant and invariant behavior of the system. We do not think that this technique leads to a full separation automatically. We have discussed in section 5.3.6 that full separation is impractical as no data can be shared between variant and invariant behavior. This limitation also applies when utilizing this technique. However, Svahnberg et al. do not explain what they mean by full separation of variant and invariant behavior, which makes it possible that they have a different view on such a separation.

We find the result of the survey regarding the increase of code structure debatable. As this technique moves any form of variant behavior to the aspect specifications, the invariant code does not necessarily become more structured. It might become more comprehensible. However, code structure is not the same as code comprehensibility, as we stated in section 6.2.3.

6.4 Variant Template Instantiation

We discuss the consequences of the variant template instantiation technique based on the results of the survey and relevant literature.

6.4.1 Consequences From Survey

For each category of features, the consequences are discussed.

6.4.1.1 Description Quality

This technique is recognized as a method to implement variability. The initial documentation of the technique is considered too abstract and could use several examples. However, the technique is considered highly memorable.

6.4.1.2 Efficiency

Substantial extra developer effort is needed for the initial implementation of this technique. The participants might have accounted for the setup of a custom parser for code entities.

The participants believe that a new variant can be created with less effort compared to the adaptation of copied code. We question the validity of this assumption on the same ground as we stated in section 6.1.4.

This technique is considered to cost more documentation effort. This can be explained by the extra technique details that need to be explained in the documentation.

The participants agree that the memory and processor usage of the final product is not affected by this technique. With this technique, no trace of the mechanism appears in the final product, which has no negative effect on memory and performance at runtime.

6.4.1.3 Maintainability

Modifying an invariant part to become a variant part is assumed to cost less effort compared to modifying copied code.

Debugging the system with this technique incorporated is more difficult than debugging a system that does not have such a technique. The code to debug has been preprocessed, which makes it harder to trace back bugs to their originating code entities. This might explain the perceived difficulty with debugging.

Variants of this technique are considered to require more effort to refactor. The unavailability of automatic refactoring tools might explain this.

It is assumed that this technique decreases the structure of the code. A system with this technique incorporated is regarded to have less readable code. Also, such a system is assumed to have higher code complexity.

6.4.2 Consequences From Literature

The variant template instantiation technique makes use of a pattern that is commonly known as a macro, where placeholders are added to files and a preprocessor replaces these placeholders with final values.

Van Deursen et al. think that the main advantage of using macros is simplicity. Its main disadvantage is that static checking and optimization are not done at the domain level. Consequently, generated code is error prone, and the user is provided with feedback on these errors at the level of the base language, or only at runtime [VDV00].

The use of a preprocessor for handling variability has advantages and disadvantages according to Baum and Becker. They state that preprocessors can effectively handle variability in both the structure of code and data, and the be-

havior of programs. Preprocessors support high versatility and fine granularity at which modifications can be applied.

However, Baum and Becker state that the preprocessors are limited in their use, as the modifications have to be planned in advance thoroughly and source code needs to be complemented by special markup [BB00].

6.4.3 Connecting Together

We think that modifying an invariant part to become a variant part using this technique will cost a similar amount of effort compared with modifying copied code. Even if the mechanism would need a negligible amount of extra effort to adjust, still a similar amount of effort would be needed to modify the code. Also, without this technique code modifications can be checked statically as mentioned by Van Deursen et al., which leads to more efficient development.

We also disagree on the survey outcome on code structure of the system. As discussed in section 6.2.3, the code structure might have been interpreted as comprehensibility. This technique can be considered less comprehensible as the invariant code is littered with placeholders. This would explain the lower valuation of the code structure.

6.5 Variant Architecture Specialization

We discuss the consequences of the variant architecture specialization technique based on the results of the survey and relevant literature.

6.5.1 Consequences From Survey

For each category of features, the consequences are discussed.

6.5.1.1 Description Quality

This technique is recognized as a method to implement variability. The participants pointed out that especially this technique could use a concrete example. This could explain the low memorability valuation by the participants. However, the technique is regarded as one of the most recognized techniques.

6.5.1.2 Efficiency

Substantial extra developer effort is needed for the initial implementation of this technique. The participants might have accounted for the encapsulation of variant code in components.

The participants believe that a new variant can be created with less effort compared to the adaptation of copied code. We question the validity of this assumption on the same ground as we stated in section 6.1.4.

This technique is considered to cost more documentation effort. This can be explained by the extra technique details that need to be explained in the documentation.

The participants unanimously agree that the memory and processor usage of the final product is not significantly affected by this technique, even though this technique introduces a mechanism that has a slight effect on memory and performance at runtime.

6.5.1.3 Maintainability

Modifying an invariant part to become a variant part is assumed by most participants to cost as much effort as making a modification in one of the copied code bases.

The participants agree that debugging the system with this technique incorporated is as difficult as debugging a system that does not have such a technique. The introduced variation point method calls are not considered problematic for the debugging.

This technique is considered to require as much effort as to refactor the copied system code. We think that this technique can be compared to the class specialization technique in terms of refactoring effort.

It is assumed that this technique increases the structure of the code.

A system with this technique incorporated is regarded to have better readable code. This might be explained by the decomposition of variant parts in components. This decomposition might be considered positive for code readability as a component makes the behavior of a separate, encapsulated part more clear.

The final system with this technique in place is assumed to have lower code complexity.

6.5.2 Consequences From Literature

We based the variant architecture specialization technique on the variant architecture component technique of Svahnberg et al. They describe on a high level how to support variant components with differing interfaces. They state that to implement the technique, a technique similar to variant class specialization is required. This explains the commonalities between the variant class specialization and architecture specialization technique.

Svahnberg et al. state that this technique may lead to code duplication between each variant component, code that could have been reused if the system had been designed differently.

Also, they state that the decision of which component interface to use, and how to use it, is placed in the calling components rather than when the actual variant feature is implemented [SvGB05].

6.5.3 Connecting Together

We do not agree that only the utilization of the variant architecture specialization technique leads to code duplication. During the implementation of the variant class specialization and component generation technique, duplicate code also appeared between the variant implementations.

However, especially when variant black box components are used in the architecture that do not reuse code outside of the component, there will be a significant amount of code duplication.

If the participants considered complexity as the number of independent paths through the code, then the answer should have been that the complexity is similar. The code paths are separate in the copied systems and combined in one system employing a VRT. However, the amount of independent paths does not increase.

6.6 Variant Component Generation

We discuss the consequences of the variant component generation technique based on the results of the survey, relevant literature and our experience implementing this technique.

6.6.1 Consequences From Survey

For each category of features, the consequences are discussed.

6.6.1.1 Description Quality

This technique is recognized as a method to implement variability. The technique is also considered highly memorable. However, most participants consider the description of this technique too abstract. Also, most of the participants state that they have not used component generation before. It seems that there is a reluctance for actually utilizing a component generation technique.

6.6.1.2 Efficiency

Participants believe that this technique of all techniques requires the most effort to implement, even when the parser and generator are already available. This might be explained by the idea that writing code templates and variant specifications for each variant component costs more effort than writing variants for the other techniques.

The participants believe that a new variant can be created with less effort compared to the adaptation of copied code.

This technique is regarded costing less documentation effort. This might be explained by the self-explanatory domain specific language in which the variants are written.

Most participants also assume that this technique will have a negative effect on memory and processor usage of the system. The participants agree that the transformation of the domain specific specification into actual code is probably less efficient than coding it directly, which has a negative effect on memory and processor usage.

6.6.1.3 Maintainability

Modifying an invariant part to become a variant part is assumed by most participants to cost less effort as making a modification in one of the copied code bases. The participants might have considered that such a change is simpler because it requires only a change in the DSL. However, the DSL specification and transformation templates that convert the DSL into actual code also need to be extended.

The participants agree that debugging the system with this technique incorporated is more difficult than debugging a system that does not have such a technique. The code is generated, which makes it harder to trace back bugs to their originating code templates. This might explain the perceived difficulty with debugging.

This technique is considered to require as much effort to refactor the variant code as in the copied system code. The participants might have considered tooling to be available to refactor a DSL.

It is assumed that this technique increases the structure of the code. This might be explained by the extra layer that is introduced that is a model of the system.

A system with this technique incorporated is considered to have better readable code. This might be explained by the decomposition of variant parts in components and the abstraction of these components in the DSL.

The final system with this technique in place is assumed to have lower code complexity.

6.6.2 Consequences From Literature

Van Deursen et al. state several consequences of using a DSL to describe a program. One of the consequences is that domain experts can understand, validate, modify, and often even develop programs in the domain specific language, because DSLs allow solutions to be expressed in the idiom and at the level of abstraction

of the problem domain.

They state that DSL programs are concise, self-documenting to a large extent, and can be reused for different purposes. They claim that DSLs enhance productivity, reliability, maintainability and portability.

They also state several negative consequences. A DSL has extra costs, because the language requires extra design, implementation and maintenance, and the language's users need education.

They state that it is difficult to balance between domain-specificity and general-purpose programming language constructs. Also, they think that there is a potential loss of efficiency when compared with hand-coded software [VDV00].

Baum and Becker discuss code generators in general. They believe that the use of code generators is the most powerful but typically also the most complex way to implement variant components. They think that the use of generators is especially tempting wherever the modification of pre-built components does not appear to be sufficient to cope with heavily varying and unforeseen requirements.

They question the use of generators for general system software functionality. They too think that generated code cannot be as efficient as manually optimized code. However, they believe that generators can make sense for narrow subdomains [BB00].

6.6.3 Consequences From Implementation

Based on our implementation in section 5.4, we note several consequences for the efficiency and maintainability features.

6.6.3.1 Efficiency

Just as with the variant class specialization technique, the initial implementation of this technique can be done incrementally. Actually, the creation of a domain specific language, validation code and code templates cannot be done in parallel.

A variant is created by its specification in a domain specific language, which is specified in a language for DSLs. During the specification of the DSL, we had the freedom to choose any possible model for the variant specification. We chose to stay close to the original variant code and remove several unnecessary generic language statements and operators.

If multiple variants need similar behavior, this can be solved by using code templates. A single statement in the DSL can be expanded to any code or other text, even to multiple files. We used this to create the HTML output and its Java behavior of the variant user interface part.

It required a lot of effort to create the initial implementation compared to the variant class specialization. We think that quite some effort can be attributed to our inexperience with oAW and the concepts of component generation. However,

we do not believe that the initial implementation can be done faster than other techniques. A developer has to manipulate a lot more complex processes using this technique.

When we finished the DSL and code templates, creating a new variation was very simple indeed. However, we needed to adjust the DSL a little bit to support several new variant components in the user interface. We opted to create a language for writing these components in. However, we thought that this would be unnecessary for this proof of concept. We did not find a reduction in effort for creating new variants using this technique either.

We did not look into documenting effort and runtime memory and processor usage during the prototype implementation. However, we believe that the code generated is indeed slightly less efficient, mainly because we chose to perform some runtime checks on method parameters in the final code that were not present in the code of the other techniques.

6.6.3.2 Maintainability

Debugging the generated variant code was the only recurring problematic issue. As the variant specification is written in a language that generates certain code, it is more difficult to track down a bug or a fault in code templates starting from the generated code than when using for instance the variant class specialization.

We found that the code complexity of the generated code did not increase by utilizing this technique. The code paths were definitively longer due to the extra checks and less efficient code, but the amount of different paths possible did not increase for a single variant.

The readability and structure of the variant specification was very high. The DSL removes several generic programming statements and operators. Also, a DSL gives the freedom to use for instance spacing and domain specific symbols to clarify the specification.

6.6.4 Connecting Together

We think that the code templates, language specification and transformer need extra documentation and that these were not accounted for by the participants.

We think that refactoring the variant code of the component generation technique is more difficult, as a developer needs to change code templates that transform the DSL specification into native code. There is limited tooling support for changing such templates.

6.7 Final Thoughts

We have given an overview of the consequences on different qualities, based on the feature analysis survey, on relevant literature and on our own experience implementing the techniques on top of a prototype application. We note several things when comparing these consequences.

The consequences given in literature do not fully cover the spectrum of consequences that are part of the survey. Svahnberg et al. only explicitly state the binding time as a selection criteria for all techniques they cover. We agree that this is an important factor that needs consideration when selecting a technique. However, we believe that there are many other factors that need to be considered.

For the techniques, the consequences that are described almost seem anecdotal and not very consistent. We think there are several possibilities as to why consequences are not consistently mentioned. It might be that many consequences are considered trivial by the author of the technique and do not have to be written down. It might be that the techniques are very similar to well-known concepts such as a specific design pattern and need no superfluous explanation. It might be that the author does not consider the product line domain where such techniques can be employed. And finally, it might be that some consequences are not taken seriously or regarded not affected.

6.7.1 Limitations of Research

We see that there are several discrepancies between the answers of the survey, our implementation and the relevant literature. Do these discrepancies all have similar underlying reasons? This is hard to tell from the data collected in this thesis. Several reasons can be given to explain why these discrepancies exist.

The survey results can be skewed, as the survey has been performed with only a few participants who each have differing experience levels with the subject and are likely to be biased towards a specific technique. The questions that have been asked on the qualities can have been misinterpreted, even though some care has been taken during the survey interview to disambiguate each question. Also, the scoring for each feature can be interpreted in different ways, and our trimming of the number of possible outcome categories might have some influence on the discrepancies.

Next to the outcome of the survey, the interpretation of the qualities based on our implementations of the techniques is purely subjective. It is very well possible that the interpretation of our implementations is skewed too. Especially when multiple techniques are implemented in serial, there is a good chance that the order of implementation has some influence on the interpretation, as we become more experienced with the subject.

Quality	VCS	IEV	VCA	VTI	VAS	VCG
Recognition	+	+	-	+	+	+
Document readability	+	+		+	+	+
Memorization	+	+	+	+	+	+
Initial implementation effort	-	-	-	-	-	-
New variant effort	-		-		-	-
Documenting effort	-		-		-	
Runtime memory usage						-
Runtime processor usage						-
Modification effort					-	-
Debugging effort		-	-	-		-
Refactoring effort	+	-	-	-	+	-
Code structure	+	-		-	+	+
Code readability	-	-	-	-	+	+
Code complexity						-

Table 6.1: Impact on qualities of variability realization techniques

6.8 Summing Up

Table 6.1 gives an overview of our opinion on the impact on qualities, based on the outcome of the survey, the literature and our experience implementing two techniques.

Chapter 7

Conclusions

This chapter gives conclusions on the research questions, positions the results in a broader context and discusses future work.

7.1 Research Results

This section gives a conclusion of the qualities when implementing variability using the six variability realization techniques.

7.1.1 Transition

When an application makes a transition to a product line, we have seen that the familiarity of developers with variability realization techniques plays an important role in the selection of one of the techniques. We have seen that developers do not actively search for software product line development or variability realization techniques in literature. Often, a solution is opportunistically chosen, based on the already present separation of concerns in the invariant code.

One of the selection criteria is the variant container. For variant class and architecture specialization, the container is a class file. For the other variants, the container is a specialized file that needs to be processed. The format of the container clearly plays a role in the selection of the technique employed. Any type of extra processing is regarded as adding more complexity to the development process.

Developers state that they would not choose more complicated techniques when a simple one would suffice. This seems to be based on the common belief that simpler code is easier to comprehend and maintain. If there are no requirements that would rule out the simpler variability solutions (for instance, requiring

a domain specific language) it seems that the more complex techniques are never considered.

However, we have seen that during development using the variant component generation technique the complexity of extra processing (the transformation of specification in code) and extra layers (the domain specific language, the code template language) can be made manageable with the right tooling. We are convinced that even the most complex of these techniques can be made manageable with technique-specific tooling. However, such tooling comes at a price, as the tooling has to be maintained next to the variability of the software. Luckily for component generation, there are several actively developed, mature tooling solutions for domain specific languages that can be employed.

Svahnberg et al. claim that one of the forces for developing software with high variability is delaying design decisions in the software lifecycle to the latest phase possible [SvGB05]. We have not seen evidence to support this claim. For instance, Svahnberg et al. see runtime binding time as an expression of delay of design decisions. What we have seen is that the choice for runtime binding time is motivated by clear advantages during development, for instance better debugging support. We have not found developers choose runtime binding time to accommodate for delay design decisions.

7.1.2 Extending Variants

The separation of invariant and variant parts by the introduction of a variability realization technique limits the initial variant design space. We have seen that this limitation of the variant part of the application is important. If too little is made variant, not all required variant applications can be created from the product line. If the variant parts are separated coarsely, unnecessary code duplication between variants will occur.

The variant architecture specialization and component generation techniques are specifically aimed at making components variable in the product line, implying a coarse, component-level variant code separation. However, components can be constructed in such a way that they contain very fine-grained behavior parts. Thus, when implemented correctly, these component techniques do not limit fine-grained variant parts in any way.

When implementing several variants, it is very likely that at some point two or more variants contain similar code. This is an opportunity to reduce duplicate variant code and increase the maintainability of these variants. Each technique has different ways to support duplicate code reduction between variants. For the variant class, architecture specialization and code adaptation technique, the code can be shared between specific variants by using inheritance. For the inner entity variant and template instantiation technique, code can be deferred to methods that are added to specific variants.

The component generation technique is a special case. Duplication can happen between the variant specifications written in the domain specific language. However, such duplication is often not considered as reduction-worthy. Especially as the domain specific language is optimized to serve a specific goal, possible reductions are probably limited. The actual code that is generated from the component specifications often contain duplicate code. However, as this code is generated, this is not regarded as problematic for maintainability.

7.1.3 Adaptation of Invariant Parts

At some point in the software lifecycle, it is probably necessary to make changes to the invariant parts of the product line. These changes can have an impact on the variant behavior. The more the invariant and variant code is separated, the less impact such a change can have. However, full separation of the invariant and variant parts is not possible.

Full separation would mean that there is no dependence between the variant and invariant parts. This would imply that for instance calling a variant method from an invariant part is not possible, as this would require a dependency between the parts. This would make it impossible to construct an application. As there is no solution for full separation, a good strategy might be to keep the amount of dependencies low.

However, during the creation of the application, many dependencies between the variant and invariant code parts can be unintentionally formed. For instance, think of dependencies on a very common invariant part of an application: a framework. A developer will probably opportunistically use or reuse parts of this framework in the variant parts of the product line. This can quickly lead to many dependencies between the variant and invariant parts.

None of the techniques accommodates for keeping the amount of dependencies as low as possible. We see it as the task of the developer to strive for low dependency to keep the invariant part adaptable and the application maintainable.

7.1.4 Maintenance of Single Variant

At some other point in the software lifecycle, it is probably necessary to perform maintenance changes on a single variant part of the product line. Luckily, changes to variant behavior do not affect the invariant behavior and can be performed without extra effort.

However, when variants share mutual code and the variant behavior needs to be separately changed, the code reduction has to be reversed for the variant. This will cost extra developer effort. It is paradoxical that code duplication reductions lead to more maintenance effort in some cases. A good strategy seems to be to conservatively reduce code duplication: only reuse code when it is clear that the

reused code will not change soon. A technique is required where the variants can be easily refactored afterwards, such as variant class specialization or architecture specialization. Another strategy is to generate code instead of reusing code using the variant component generation technique.

7.1.5 Common Consequences

We have seen that during the extending of the variants and the adaptation and maintenance of the invariant and variant parts of the application, there are several recurring consequences for implementing variability using the variability realization techniques.

7.1.5.1 Separation of Concerns

One consequence is constantly reconsidering the extents of the variant part of an application. Only by keeping the amount of dependencies low between the variant and invariant parts of the application and between the variants themselves, the application can be adaptable and maintainable.

7.1.5.2 Reuse Between Variants

One consequence is tackling reuse of code between variants. With the number of variants increasing, the chance increases as well that code is duplicated between two or more variants. However, as we have concluded for the maintenance of a single variant, reducing code duplication can have a counter effect and increase maintenance costs.

The tackling of reuse between variants is no different from tackling reuse in applications in general. We see that there are three strategies for handling code reuse between variants:

No reuse
: With this strategy, no effort is put in reducing the recurring code parts. This can cause a buildup of duplicate code. When faults have to be fixed or enhancements are to be made in the code that has been duplicated numerous times, extra developer effort has to be put in to trace back this code. However, this can be an acceptable solution if the amount of duplicated code is low. This often happens when each variant is very unique in nature.

Component reuse
: With this strategy, recurring code parts are moved to components and replaced by method calls to the component. To make sure that the behavior of each variant is not affected by changes to the reusable components, each variant needs to be extensively tested by a unit test.

This can be a solution if each variant is decomposable in one or more components and overlapping components between variant decompositions exist. This often happens when there is an evident composition structure in each variant, for instance in a user interface built from components.

Reuse by copy

With this strategy, recurring code is copied to arbitrary places in the variant code when necessary. This can be done using code templates, as seen in the variant component generation technique.

Just as with component or method reuse, the only way to test that the behavior of each variant is not affected by changes to the reusable components or methods, each variant needs to be extensively tested by a unit test.

This can be a solution if each variant does not have an evident composition structure but does have recurring code. This often happens when variants are mainly imperative in style.

We believe that the selected reuse strategy has a significant effect on efficiency and maintainability.

7.2 Contributions of Research

This thesis contributes to the ongoing research on variability realization techniques, its implications on development and runtime efficiency and maintainability of software products and its limitations and pitfalls.

The main contribution of this thesis is a detailed description of several well-known and lesser-known variability realization techniques, written in a design pattern style, with extensive documentation on the consequences of implementing these techniques. For each technique, the impact on a selection of qualities of the final software product is given. This overview can be used to select a technique with desired features.

We believe that such a detailed description has not been published until now. The taxonomy of techniques that Svahnberg et al. have published focuses on other aspects of variability and does not have as much concrete examples as our description.

This thesis describes a methodology to gather information on qualities of VRTs, to rate VRTs according to the outcome of the methodology and to rank the VRTs in several dimensions. The execution of the methodology within an industrial setting is described and the outcome is presented in this thesis. The methodology can be reused in follow-up research to extend or refine the qualities that were subject of this thesis.

The results of the methodology are connected to the relevant literature for these techniques. Also, for two techniques, the hands-on experience implementing

them in an industry setting is described. This leads to a final conclusion on the impact of several qualities of the system, which can also be used to select a technique with desired features.

7.3 Future Work

In future work, several other techniques can be described in detail, as the techniques proposed in this thesis do not cover every used technique in the industry. Also, the current techniques can be extended with even more examples from practice and from other domains.

For each technique, there are many more qualities that can be selected for evaluation. The qualities that have been evaluated might be too generic or too specific for use in another domain.

The feature analysis answers can be validated quantitatively by performing a quantitative analysis. For instance, this can be done by measuring certain code aspects of systems that have a technique incorporated. Or, development progress can be measured during the implementation of a software product line using one or more VRTs.

An important subject that needs future research is the consequence of mixing different techniques together in a single system. When is it worthwhile to mix techniques? How do these techniques work together and what are the pain points? What are obvious and less obvious limitations that need to be accounted for?

Bibliography

[AHI+05] P. America, D. Hammer, M.T. Ionita, H. Obbink, and E. Rommes. Scenario-Based Decision Making for Architectural Variability in Product Families. *Software Process Improvement and Practice*, 10(2):171–187, 2005.

[BB00] L. Baum and M. Becker. Generic components to foster reuse. In *Technology of Object-Oriented Languages and Systems, 2000. TOOLS-Pacific 2000. Proceedings. 37th International Conference on*, pages 266–277, 2000.

[BBL76] B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, pages 592–605, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[BCR94] V.R. Basili, G. Caldiera, and H.D. Rombach. The Goal Question Metric Approach. *Encyclopedia of Software Engineering*, 1:528–532, 1994.

[Ber06] K. Berg. Change Impact Analysis of Crosscutting in Software Architectural Design. In *Workshop on Architecture-Centric Evolution (ACE 2006) at ECOOP July*, pages 3–7, 2006.

[BKLW95] M. Barbacci, M. Klein, T. Longstaff, and C. Weinstock. Quality Attributes (CMU/SEI-95-TR-21, ADA307888). Pittsburgh, PA: Software Engineering Institute, 1995.

[BLBvV04] P.O. Bengtsson, N. Lassing, J. Bosch, and H. van Vliet. Architecture-level modifiability analysis (ALMA). *The Journal of Systems & Software*, 69(1-2):129–147, 2004.

[BLP04] S. Bühne, K. Lauenroth, and K. Pohl. Why is it not Sufficient to Model Requirements Variability with Feature Models? In *Proceedings Workshop on Automotive Requirements Engineering (AURE04). Nanzan University, Nagoya, Japan*, 2004.

[BM04a] A. Bragança and R.J. Machado. A Methodological Approach to Domain Engineering for Software Variability Enhancement. In *OOPSLA'2004 Second Workshop on Method Engineering for Object-Oriented and Component-Based Development, Vancouver, Canada*, 2004.

[BM04b] A. Bragança and R.J. Machado. Run-time Variability Issues in Software Product Lines. In *ICSR8 Workshop on Implementation of Software Product Lines and Reusable Components, Madrid*, 2004.

[Bos00] J. Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison-Wesley, 2000.

[BPSP04] D. Beuche, H. Papajewski, and W. Schröder-Preikschat. Variability management with feature models. *Science of Computer Programming*, 53(3):333–352, 2004.

[BS97] A.W. Brown and K. Short. On Components and Objects: The Foundations of Component-Based Development. In *Proceedings of the 5th International Symposium on Assessment of Software Tools. IEEE Computer Society Press. Jun*, 1997.

[BVG+02] M. Becker, S. Voget, L. Geyer, A. Gilbert, K. Becker, L. Baum, M. Becker, L. Geyer, A. Gilbert, G. Molter, et al. Mapping Variabilities onto Product Family Assets. In *Proceedings of the International Colloquium of the Sonderforschungsbereich*, volume 501, 2002.

[BW01] Lionel C. Briand and Jörgen Wüst. Integrating scenario-based and measurement-based software product assessment. *Journal of Systems and Software*, 59(1):3–22, 2001.

[CN02] P. Clements and L. Northrop. *Software product lines*. Addison-Wesley Boston, 2002.

[CvdBA07] S. Ciraci, P. van den Broek, and M. Aksit. A Taxonomy for a Constructive Approach to Software Evolution. *JOURNAL OF SOFTWARE*, 2(2), 2007.

[CW98] R. CONRADI and B. WESTFECHTEL. Version Models for Software Configuration Management. *ACM Computing Surveys*, 30(2), 1998.

[Cza04] Krzysztof Czarnecki. Overview of generative software development. In *In Proceedings of Unconventional Programming Paradigms (UPP) 2004, 15-17 September, Mont Saint-Michel, France, Revised Papers*, pages 313–328. Springer-Verlag, 2004.

[DFV03] E. Dolstra, G. Florijn, and E. Visser. Timeline Variability: The Variability of Binding Time of Variation Points. In *Proceedings of the Workshop on Software Variability Management*, pages 119–122, 2003.

[DKJ05] T. Dyba, BA Kitchenham, and M. Jorgensen. Evidence-based software engineering for practitioners. *Software, IEEE*, 22(1):58–65, 2005.

[DNBS04] S. Deelstra, J. Nijhuis, J. Bosch, and M. Sinnema. The COVAMOF Software Variability Assessment Method (COSVAM). In *Proceedings of the 2nd Groningen Workshop on Software Variability Management (SVMG 2004)*, 2004.

[DS08] S.K. Deelstra and M. Sinnema. Managing the Complexity of Variability in Software Product Families. 2008.

[DSB04] Sybren Deelstra, Marco Sinnema, and Jan Bosch. Experiences in software product families: Problems and issues during product derivation. In Robert L. Nord, editor, *SPLC*, volume 3154 of *Lecture Notes in Computer Science*, pages 165–182. Springer, 2004.

[DSGB03] Sybren Deelstra, Marco Sinnema, Jilles Van Gurp, and Jan Bosch. Model driven architecture as approach to manage variability in software product families. In *University of Twente*, pages 109–114, 2003.

[EGK+01] S.G. Eick, T.L. Graves, A.F. Karr, JS Marron, and A. Mockus. Does Code Decay? Assessing the Evidence from Change Management Data. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, pages 1–12, 2001.

[EV05] J. Estublier and G. Vega. Reuse and variability in large software applications. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 316–325. ACM New York, NY, USA, 2005.

[FCS+08] Eduardo Figueiredo, Nelio Cacho, Claudio Sant'Anna, Mario Monteiro, Uira Kulesza, Alessandro Garcia, Sergio Soares, Fabiano Ferrari, Safoora Khan, Fernando Castor Filho, and Francisco Dantas. Evolving software product lines with aspects: an empirical study on design stability. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 261–270, New York, NY, USA, 2008. ACM.

[FLS02] C. Fritsch, A. Lehn, and T. Strohm. Evaluating Variability Implementation Mechanisms. In *International Workshop on Product Line Engineering: The Early Steps: Planning, Modeling, and Managing*, 2002.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Massachusetts, January 1995.

[Gri93] Martin L. Griss. Software reuse: from library to factory. *IBM Syst. J.*, 32(4):548–566, 1993.

[HA07] Neil B. Harrison and Paris Avgeriou. Leveraging architecture patterns to satisfy quality attributes. In Flávio Oquendo, editor, *ECSA*, volume 4758 of *Lecture Notes in Computer Science*, pages 263–270. Springer, 2007.

[HvdH07] Scott A. Hendrickson and Andre van der Hoek. Modeling product line architectures through change sets and relationships. *Software Engineering, International Conference on*, 0:189–198, 2007.

[IHO04] M. Ionita, D. Hammer, and H. Obbink. Scenario Based Software Architecture Evaluation Methods: An Overview. *Analysis*, 5:6, 2004.

[ISO98] ISO. ISO/IEC FIS 14764. software engineering - software maintenance, 1998.

[JB02] M. Jaring and J. Bosch. Representing Variability in Software Product Lines: A Case Study. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 15–36, 2002.

[jsf] Sun JavaServer Faces. `http://java.sun.com/javaee/javaserverfaces`.

[KBAW94] R. Kazman, L. Bass, G. Abowd, and M. Webb. SAAM: A Method for Analyzing the Properties of Software Architectures. In *INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING*, volume 16, pages 81–81. IEEE COMPUTER SOCIETY, 1994.

[KBI94] R. Kazman, L. Bass, and CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST. *Toward Deriving Software Architectures from Quality Attributes*. Carnegie Mellon University, Software Engineering Institute, 1994.

[KCH+90] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[KKC00] R. Kazman, M. Klein, and P. Clements. ATAM: Method for Architecture Evaluation. 2000.

[KLL97] B. Kitchenham, S. Linkman, and D. Law. DESMET: a methodology for evaluating software engineering methodsand tools. *Computing & Control Engineering Journal*, 8(3):120–126, 1997.

[Lan02] Rikard Land. Software deterioration and maintainability – a model proposal. In *In Proceedings of Second Conference on Software Engineering Research and Practise in Sweden (SERPS), Blekinge Institute of Technology Research Report 2002:10*, 2002.

[LB85] MM Lehman and LA Belady. *Program evolution: processes of software change*. 1985.

[Lik32] R. Likert. *A technique for the measurement of attitudes*. sn, 1932.

[LKL02] K. Lee, K.C. Kang, and J. Lee. Concepts and Guidelines of Feature Modeling for Product Line Software Engineering. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 62–77, 2002.

[McC01] C.H. McCall. An empirical examination of the Likert scale: some assumptions, development, and cautions. In *80th Annual CERA Conference, South Lake Tahoe, CA, November*, pages 15–16, 2001.

[MMR06] Varvana Myllärniemi, Tomi Männistö, and Mikko Raatikainen. Quality attribute variability within a software product family architecture. In *In Short paper in Conference on the Quality of Software Architectures (QoSA*, 2006.

[NGT92] O. Nierstrasz, S. Gibbs, and D. Tsichritzis. Component-oriented software development. *Communications of the ACM*, 35(9):160–165, 1992.

[ope] openArchitectureWare. `http://www.openarchitectureware.org`.

[RGD07] R. Rabiser, P. Grunbacher, and D. Dhungana. Supporting Product Derivation by Adapting and Augmenting Variability Models. In *Software Product Line Conference, 2007. SPLC 2007. 11th International*, pages 141–150, 2007.

[Saa80] T.L. Saaty. The Analytical Hierarchical Process, 1980.

[SDNB04] M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch. COVAMOF: A Framework for Modeling Variability in Software Product Families. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 197–213, 2004.

[SH04] Mark Staples and Derrick Hill. Experiences adopting software product line development without a product line architecture. In *APSEC*, pages 176–183. IEEE Computer Society, 2004.

[SvGB05] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. A taxonomy of variability realization techniques: Research articles. *Softw. Pract. Exper.*, 35(8):705–754, 2005.

[SW05] M. Svahnberg and C. Wohlin. An Investigation of a Method for Identifying a Software Architecture Candidate with Respect to Quality Attributes. *Empirical Software Engineering*, 10(2):149–181, 2005.

[TA04] B. Tekinerdogan and M. Aksit. Managing variability in product line scoping using design space models. In *Impact of Software Process on Quality Workshop*, Izmir, 2004. Ege University.

[tap] Apache Tapestry. `http://tapestry.apache.org`.

[vdH04] A. van der Hoek. Design-time product line architectures for any-time variability. *Science of Computer Programming*, 53(3):285–304, 2004.

[VDV00] A. Van Deursen and J. Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.

[vGB02] J. van Gurp and J. Bosch. Design erosion: problems and causes. *The Journal of Systems & Software*, 61(2):105–119, 2002.

[vGS06] J. van Gurp and J. Savolainen. Service Grid Variability Realization. In *Proceedings of the 10th International on Software Product Line Conference*, pages 85–94. IEEE Computer Society Washington, DC, USA, 2006.

[WCD+05] Sam Weber, Hoi Chan, Lou Degenaro, Judah Diament, Achille Fokoue-Nkoutche, and Isabelle Rouvellou. Fusion: A system for business users to manage program variability. *IEEE Trans. Software Eng.*, 31(7):570–587, 2005.

[wic] Apache Wicket. `http://wicket.apache.org`.

[Zdu06] U. Zdun. Tailorable language for behavioral composition and configuration of software components. *Computer Languages, Systems & Structures*, 32(1):56–82, 2006.

Appendices

Appendix A

Variability Realization Techniques

The following chapter describes six different variability realization techniques (VRTs). The VRTs will be described in a design pattern format [GHJV95].

A.1 Variant Class Specialization

Intent

Move cross-cutting code of a variant feature in a class per variant.

Also Known As

- Variant component specialization [SvGB05]
- Runtime variant component specialization [SvGB05]
- Variant component implementation [SvGB05]

Related Patterns

- Variant code adaptation (section A.3)
- Template Method [GHJV95]

Forces

For some variant features, variation points in many components are necessary to support the implementation of that feature. Such code is called crosscutting. It might be desirable to move the code for each variant into a single class.

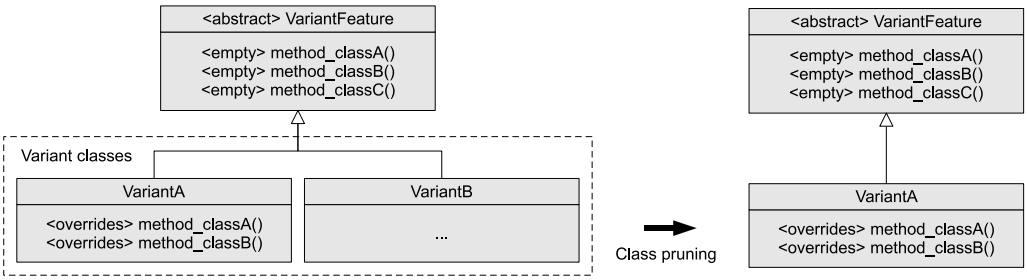

Figure A.1: Example variant selection process (VCS technique)

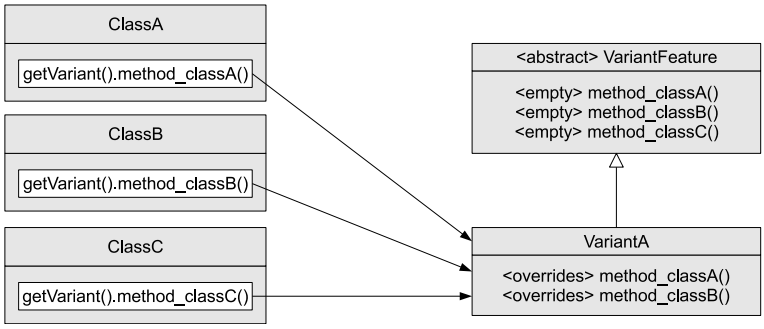

Figure A.2: Example interaction between variant and invariant code (VCS technique)

Applicability

When it is desirable to move code of crosscutting variants to a single class, and it is not possible or feasible to use variant code adaptation (see section A.3).

Structure

Figure A.1 shows the structure of the variant classes and the abstract class.

VariantFeature
: An abstract class that contains empty variant method implementations. These methods can be overridden by the variant subclasses.

VariantA, -B
: The classes that extend the abstract class and contain the code of one single variant of the variant feature.

Figure A.2 shows the structure of the invariant classes that call the variant class methods.

ClassA, -B, -C
: Invariant classes that contain method calls to the selected variant class.

Implementation

Create an abstract class with empty methods for each variation point. Add method calls in the invariant code at places where variant behavior needs to be performed if necessary. This is illustrated by Figure A.2.

Use unique names for each variant point. Practically, the method name should refer to the location of the variant point to avoid name clashes and increase developer understanding.

If certain objects should be manipulated by the variant code, add them as a method argument. Each variant should be able to be implemented by extending the abstract class and overriding variation point methods where needed. If this is not possible, not all variation points have been added to the class, or not all necessary objects are added as method arguments.

At build time, select one of the variant classes that will be packaged in the final product and leave out all other variant classes. This is illustrated by the class pruning in Figure A.1, where VariantA is selected.

Consequences

Moving variant code to a separate class introduces a layer of indirection that causes a method call per variant point. Also, the variant object costs some heap memory to instantiate. This might be problematic if many variant features are implemented using this pattern. A solution can be to implement all methods as statics.

Variant objects can be stateful if the instantiated variant class is not discarded when a method has been called, but is kept in memory. However, statefulness can lead to developer confusion, as each method call can produce different results. We advise to avoid the use of statefulness in variant classes.

Common code that spans multiple variants in a variant feature can be added as protected methods to the abstract `VariantFeature` class. This can reduce code duplication, but decreases comprehensibility of the code.

Also, many variation points in a class will lead to many methods called, which reduces comprehensibility of the code. Code browsing facilities of an Integrated Development Environment (IDE) will be very useful to browse to the variant code and back during development.

Adding a new variation point requires adding a new method to the abstract class. Adding a new variant requires creating a new variant class that extends the abstract `VariantFeature` class. Both cases are often supported by an IDE using refactoring to keep developer effort to a minimum.

Similar Approaches

Instead of using an abstract class, make `VariantFeature` an interface, analogous to the Strategy pattern [GHJV95]. Then, each variant can implement the interface directly. However, this leads to undesirable developer overhead, as all methods from the interface need to be implemented in each variant class. Alternatively, a default implementation of the interface can be made, just as the abstract class with empty methods. This default implementation can then be subclasses by each variant implementation. However, this requires the use of an extra class. Using an abstract class is advised.

A.2 Inner Entity Variant

Intent

Support several variant implementation parts of a variant feature within the same file.

Also Known As

- Condition on constant [SvGB05]
- Condition on variable [SvGB05]

Forces

It can be desirable to have different parts of variant implementations bundled in the same file. For instance, a component can consist of a class file for its behavioral code and a markup file for its visual description. It might be undesirable if creating multiple variants of this component leads to new class and markup files. This can cause a buildup of semi-identical files with many mutual similar parts.

Applicability

When a new variant can be contained in the same file, and this is desirable.

Structure

Figure A.3 shows the structure of this technique.

FileA, -B, -C
: The files that each contain parts of all variant implementation.

ProcessedFileA, -B, -C
: Processed files that only contain the parts of a single variant implementation.

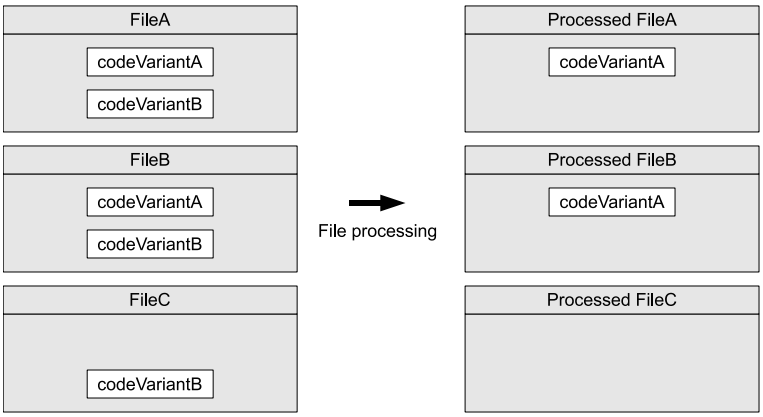

Figure A.3: Example file processing (IEV technique)

Implementation

There are multiple ways of implementing this. Several compilers support `#IFDEF` or similar preprocessor statements to mark code that should be in- or excluded based on a compiler parameter. For other file contents, a specialized file preprocessor can be used at build time to simulate the use of `#IFDEF` statements. The file processing is illustrated by Figure A.3, where VariantA is selected.

Examples

C code:

```
#IFDEF VARIANT_A
  printf ("In variant A");
  ...
#ENDIF
#IFDEF VARIANT_B
  printf ("In variant B");
  ...
#ENDIF
```

HTML markup:

```
<!-- begin_variantA -->
  <div>...</div>
<!-- end_variantA -->
<!-- begin_variantB -->
  <table>...</table>
<!-- end_variantB -->
```

Consequences

When there are multiple variation points in a single file, the comprehensibility of what will be part of the processed components will decrease. Also, when there are variation points crosscutting variants, the comprehensibility will decrease even more.

Common code in variant implementations can be moved to shared methods. However, this again leads to a decrease in comprehensibility when too much code is shared. Common artifacts in other files can be shared too. However, this is dependent on the features of the processor.

A new variant for a single variation point can be added with little effort. However, a variant often is manifested in many variation points scattered throughout the code. There is only crude tooling available to locate each variation point, which makes maintenance more difficult.

It might not be necessary to update all files when a new variant is introduced: only the affected variation points need to be updated. If a new variant requires a new variation point, this can be quickly realized for code as well as other file contents. For `#IFDEF`-statements, some IDEs have proper support.

A.3 Variant Code Adaptation

Intent

Separate variant feature code entirely from the base code.

Also Known As

Code fragment superimposition [SvGB05]

Forces

Some variant features have many variant points that crosscut throughout the base code. This decreases the readability and maintainability of the base code, and understanding of the intent of base components.

Applicability

When the readability of the base code is a very important quality factor.

Structure

Figure A.4 shows the structure of this technique.

ClassA, -B, -C

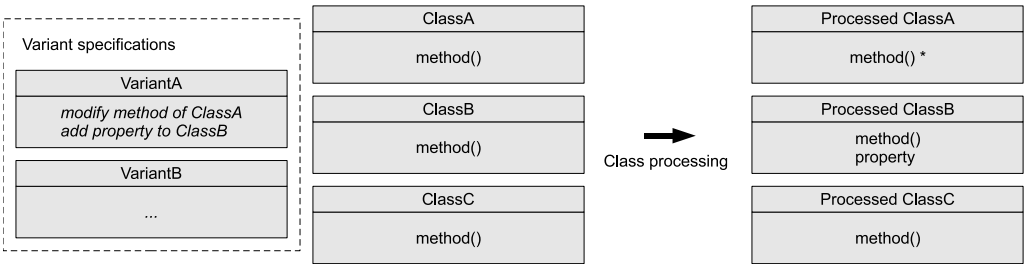

Figure A.4: Example class processing (VCA technique)

The base classes that do not contain any variation points or variant code. These classes do not provide special placeholders or directives related to variability, to keep them as readable and maintainable as possible.

VariantA, -B
Adaptation specifications for each variant on how the classes should be modified.

Processed ClassA, -B, -C
Processed classes that have been modified according to the variant adaptation specification.

Implementation

Use a class processor that supports adaptation of code before or during compile time, for instance AspectJ. It is not advised to write your own code adaptation implementation, as the existing tooling is mature enough. Extract the variant code from the system and rewrite it as a variant adaptation specification.

At build time, one of the variant adaptation specifications is selected and the code is adjusted according to the selected specification. This is illustrated by figure Figure A.4, where VariantA is selected.

Consequences

Using this adaptation pattern separates the variant feature from the base code. This increases the readability of the base code and the maintainability of the base classes. However, many dependencies exist between the base code and the separated variant feature. When developing or changing the base code, it is more difficult to understand if the adaptations still function properly and if or how the adapted code will function.

Common code for each variant can be separated into external classes that are added to the adapted classes. Also, the variant adaptation specification can also use principles like inheritance to reduce duplicated code. This is for instance supported in AspectJ.

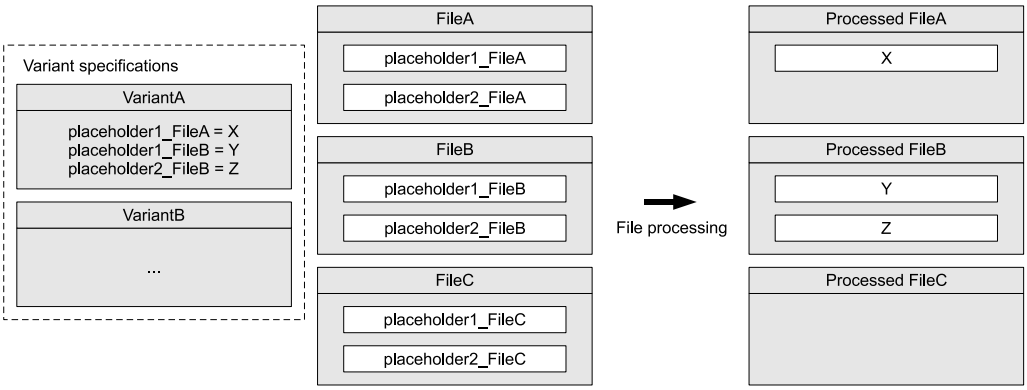

Figure A.5: Example file processing (VTI technique)

A variant feature that crosscuts another variant feature is not possible using AspectJ, as the adaptation of code is performed only once.

Adding a new variant means adding a new variant adaptation specification. Just like the variant class specialization, all code for each variation point is separated in a single file.

Currently, AspectJ has an Eclipse plugin that provides visual feedback about where aspects are applied.

A.4 Variant Template Instantiation

Intent

Support different variants of code artifacts or text files, by separating templates that differ at predefined places and the variant code or text that is filled in these places.

Forces

Some variant features do not only crosscut code but also other text files: for instance configuration files or markup files. Both code and text files can benefit from a similar technique.

Applicability

When not only code is affected by a variant feature.

Structure

Figure A.5 shows the structure of this technique.

FileA, -B, -C

The base file templates that contain placeholders with a special syntax where values can be added during processing.

VariantA, -B
Value specifications for each variant, that contain the values that will replace the placeholders in the template. These values can be any kind of text, from small markup language fragments to entire code classes.

Processed FileA, -B, -C
Processed files that have their placeholders replaced according to a value specification.

Implementation

Choose the right format for the variant template values. A Java properties file can be sufficient. Use one of the numerous libraries that provide template processing. Alternatively, build a file processor from scratch. The processor should be able to process any kind of text file.

At build time, use the file processor to fill in the template files according to the selected variant value specification.

Examples

Java code:

```
function aMethod() {
  ...
  /*placeholder_aMethod_aComponent*/
  ...
}
```

HTML markup:

```
<div>
  <!--placeholder_aComponentMarkup-->
</div>
```

Variant value specification file:

```
placeholder_aMethod_aComponent=someCodeFragment();
placeholder_aComponentMarkup=<input type="text"/>
```

Consequences

Using this pattern separates the variant values from the base templates, but the variation point placeholders in the templates decrease the readability of the templates.

Common artifacts cannot be shared between variants, and variant features that crosscut other variants are not supported. However, the component processor can be extended to support such techniques.

Adding a new variant means adding a file with a new set of variant values. If a new variation point is added, no variant needs to be modified unless it wants to use the new variation point. If a variation point hook is renamed, all variation value files need to be adjusted by hand, assuming that there is no refactor tooling support for a custom template format.

A.5 Variant Architecture Specialization

Intent

Support variations in the architecture, by making components of the structure variant.

Related Patterns

- Architecture reorganization [SvGB05]
- Variant component implementations [SvGB05]

Forces

Some variants of a product need to have interchangeable components in the architecture that do not have a common interface. For instance, variants are introduced to support several legacy versions of a component. Or, variants need to support different off-the-shelve components.

Applicability

When a variant of a variant feature consist of a component that should be interchangeable in the architecture but does not have a common interface.

Structure

Figure A.6 shows the structure of this technique.

VariantFeature
: An interface that each variant component wrapper has to implement.

WrapperVariantA, -B
: The wrapper classes, which implement the `VariantFeature` interface that unifies the variant component interface.

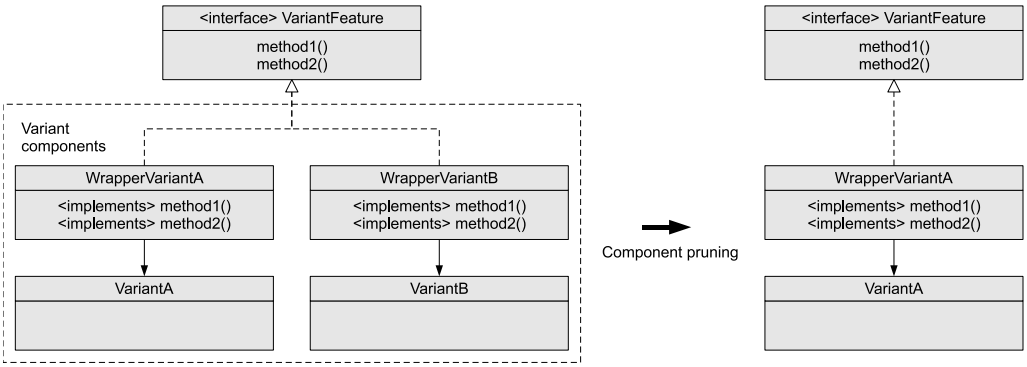

Figure A.6: Example component selection (VAS technique)

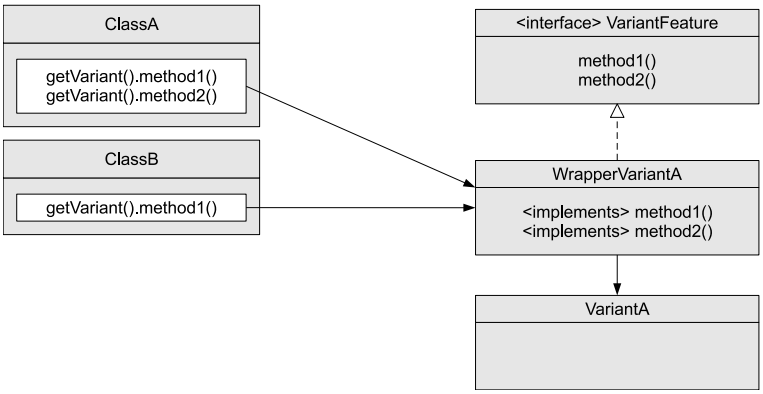

Figure A.7: Example interaction between variant and invariant code (VAS technique)

VariantA, -B

The variant components that are wrapped by its wrapper.

Figure A.7 shows the structure of the invariant classes that call the selected variant component via its wrapper.

ClassA, -B

The classes that contain method calls to the wrapped components.

Implementation

Identify which components of the architecture need to vary. Create a unified interface for the interchangeable components. Implement wrappers that wrap around each component and perform the mapping between the desired method calls and the actual component interface and behavior.

There are a few differences between the implementation of the variant class specialization technique and this technique. Class specialization focuses on mov-

ing method-size variant code to a variant class; this technique focuses on making components variant. Wit class specialization, the variant interface can simply be created by comparing code differences; this technique requires a well-thought out interface that unifies several variant-specific interfaces.

At build time, select one of the variant components that needs to be packaged in the final product and leave out the other components and wrappers. This is illustrated by Figure A.6.

Consequences

It can be problematic to create a unified interface for the variant components; the interface must be extensive enough to support all wrapped components and be concise enough not to introduce extra developer maintenance overhead. It is advised to keep variant-specific methods to a minimum.

Adding a new component variant requires a new wrapper to be written for that component. Extra development effort has to be taken into account when the new component requires an extension or adjustment of the common interface. Also, the common interface often needs to be extended to support new variation points.

A.6 Variant component generation

Intent

Support variant components, by generating component code based on a variant component model that is expressed in a domain specific language.

Forces

It can be desirable to describe the structure and behavior of different variant components using a domain specific language (DSL). In a DSL, a component can be expressed in the idiom and at the level of abstraction of the component domain; this increases understanding and maintainability of the functionality of the component.

Applicability

When the variant component structure and behavior can be expressed in a DSL more clearly than pre-existing languages would allow. This implies that the DSL must be less comprehensive and yet more expressive in its domain than any pre-existing language.

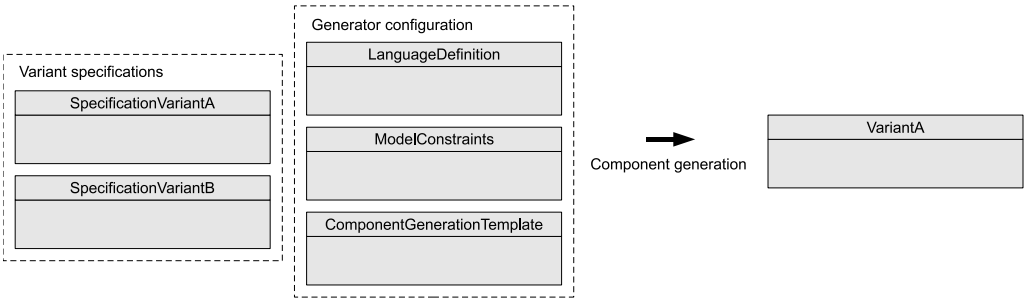

Figure A.8: Example component generation (VCG technique)

Structure

Figure A.8 shows the structure of this technique.

SpecificationVariantA, -B
The specifications that generate the different variant components.

LanguageDefinition
The definition of the domain specific language, which maps specifications written in the DSL to a variant component model.

ModelConstraints
The constraints that limit the variant component model which is created by a specification.

ComponentGenerationTemplate
The template file that is used to expand a variant component model into the generated code for the variant component.

VariantA
The generated component.

Implementation

Design a DSL that is expressive enough to model the required variant components, but is not overly comprehensive. Create a parser for the DSL that can interpret the DSL and create a variant component model based on a variant specification or use supportive DSL tooling to generate such a parser. Create a component generator that transforms a variant component model into generated code based on a generation template or use DSL tooling to generate such a generator.

At build time, select a variant specification and perform component generation. Include the generated results in the final product. This is illustrated by Figure A.8.

Consequences

The transformation of the variant component model into actual code can be a very complex process. This can quickly become a performance bottleneck during component generation.

Also, the code that is generated might not be just as optimal as its hand-coded variant, which implies a potential loss of processor and memory efficiency at runtime. A solution is to optimize code during code generation, based on the properties of the variant component model. However, this requires extra development effort and is not always possible.

Domain specific languages make it possible to express a solution at the level of abstraction of the variant component domain. This helps domain experts because they can understand, validate, modify, and even develop domain-specific language specifications.

Some of the costs of building variant components are moved from the code developer to the domain expert. This is a trade-off however, as the developer has to design, implement and maintain the DSL in return.

A DSL can quickly become a limited, general-purpose programming language. This can be desirable in some cases, but it is difficult to balance domain-specificity and general-purposiveness.

Because DSLs are more abstract and can use idioms from the problem domain, less documentation is required for each variant component; the DSL specification should be clear on the behavior.

Sharing common specification parts between different specifications is dependent on the expressiveness of the DSL. For instance, if the DSL allows a specification file to be imported, this file can be used to share a common specification.

If a new variant requires features that the DSL does not yet provide, the DSL needs to be extended and generation templates need to be adjusted. The DSL should be expressive enough to avoid this.

There is some IDE support to create and implement domain specific languages. Also, tooling exists to visually create a language or to write a domain specific language and code generation templates. This tooling reduces the effort to introduce this technique in the system.

Appendix B

Survey Results

This appendix gives an overview of the unprocessed results of the feature analysis survey.

B.1 Questions in Survey

For each technique, the participants rate the following statements to assess its documentation quality.

- I recognized it.
- I had no trouble reading it.
- I can remember it easily.

Possible answers are:

1. Strongly disagree
2. Somewhat disagree
3. Neither disagree or agree
4. Somewhat agree
5. Strongly agree

For each technique implemented in a single system, the following questions are asked to assess the impact on the system's qualities, compared to a set of variant implementations of the system without variability mechanisms.

- How much effort does the initial implementation cost?
- How much effort does the creation of a new variant cost?
- How much effort does documenting the code cost?
- How much memory do the similar variant products use during a test run?
- How much CPU power is consumed by the similar variant products during a test run?
- How much effort does modifying an invariant part of the code to become variant cost?
- How much effort does debugging the variant code cost?
- How much effort does refactoring the variant code cost?
- How much of the system code is structured?
- How much of the system code is well readable?
- How much of the system code is complex?

Possible answers are:

1. Much less
2. Less
3. Similar
4. More
5. Much more

B.2 Results

This section gives the unprocessed results of the survey. The first column represents the quality, which matches the order of each question in the survey. The results of the eight participants are represented in the eight columns. The number represents the answer given.

B.2.1 Variant Class Specialization

Quality	P1	P2	P3	P4	P5	P6	P7	P8
Recognition	3	5	1	4	5	5	5	4
Document readability	2	4	1	5	4	2	3	4
Memorization	4	5	3	4	1	4	4	4
Implementation effort	4	4	2	3	4	4	4	4
New variant effort	2	2	2	3	2	2	2	2
Documenting effort	3	3	2	4	4	2	3	4
Runtime memory usage	3	3	3	3	4	3	3	3
Runtime processor usage	3	3	3	3	4	3	3	3
Modification effort	2	3	3	3	3	2	4	2
Debugging effort	3	4	3	3	4	3	3	4
Refactoring effort	2	3	3	3	2	2	2	3
Code structure	2	5	4	3	4	2	2	4
Code readability	2	4	2	3	3	2	2	3
Code complexity	4	3	4	4	3	4	4	4

B.2.2 Inner Entity Variant

Quality	P1	P2	P3	P4	P5	P6	P7	P8
Recognition	4	5	5	5	5	5	5	5
Document readability	2	5	4	5	5	5	5	5
Memorization	4	5	5	5	5	5	4	5
Implementation effort	4	4	2	3	4	3	3	3
New variant effort	3	3	3	3	2	3	3	3
Documenting effort	5	3	2	3	4	2	4	4
Runtime memory usage	3	3	3	3	3	3	3	3
Runtime processor usage	3	3	3	3	3	3	3	3
Modification effort	4	4	2	3	4	2	4	2
Debugging effort	4	4	3	3	4	3	3	4
Refactoring effort	5	4	2	3	4	4	3	4
Code structure	1	2	4	2	2	2	3	2
Code readability	1	2	2	3	2	2	3	2
Code complexity	4	4	4	3	4	3	3	4

B.2.3 Variant Code Adaptation

Quality	P1	P2	P3	P4	P5	P6	P7	P8
Recognition	1	1	1	1	4	1	1	4
Document readability	2	3	4	4	1	2	3	3
Memorization	4	3	4	2	2	4	5	4
Implementation effort	5	4	2	3	4	3	5	4
New variant effort	3	3	3	4	3	3	3	4
Documenting effort	3	4	4	4	2	3	5	4
Runtime memory usage	3	3	3	3	3	3	3	3
Runtime processor usage	3	3	3	3	3	3	3	3
Modification effort	2	4	4	3	3	2	3	3
Debugging effort	3	4	5	3	4	3	4	5
Refactoring effort	3	4	4	4	3	4	3	4
Code structure	4	4	4	2	4	2	3	4
Code readability	4	1	2	2	3	2	2	3
Code complexity	3	4	4	3	4	4	2	4

B.2.4 Variant Template Instantiation

Quality	P1	P2	P3	P4	P5	P6	P7	P8
Recognition	2	5	5	4	5	5	2	5
Document readability	2	4	5	4	5	5	4	2
Memorization	3	5	4	4	5	5	2	4
Implementation effort	4	5	4	3	2	4	4	4
New variant effort	2	3	2	2	2	2	2	3
Documenting effort	4	4	2	3	3	4	5	3
Runtime memory usage	3	3	3	3	3	3	3	3
Runtime processor usage	3	3	3	3	3	3	3	3
Modification effort	4	4	2	3	2	2	2	2
Debugging effort	3	4	3	3	4	4	5	4
Code structure	2	4	4	3	2	2	2	2
Code readability	2	2	3	3	3	2	2	2
Code complexity	4	4	4	3	2	4	4	3

B.2.5 Variant Architecture Specialization

Quality	P1	P2	P3	P4	P5	P6	P7	P8
Recognition	5	2	5	5	4	5	4	5
Document readability	2	4	5	5	2	3	5	4
Memorization	3	2	4	5	2	5	3	4
Implementation effort	5	4	4	4	2	5	4	5
New variant effort	2	3	2	2	2	3	3	3
Documenting effort	2	4	4	2	3	4	3	4
Runtime memory usage	3	3	4	3	4	3	3	3
Runtime processor usage	3	3	3	3	4	3	3	3
Modification effort	3	5	3	3	4	5	3	2
Debugging effort	3	4	4	3	4	4	4	4
Code structure	5	3	4	4	4	4	4	5
Code readability	4	2	2	4	4	4	4	4
Code complexity	2	4	4	2	2	2	2	4

B.2.6 Variant Component Generation

Quality	P1	P2	P3	P4	P5	P6	P7	P8
Recognition	4	5	4	4	4	5	4	5
Document readability	2	5	4	5	3	5	5	5
Memorization	3	5	4	5	4	5	5	5
Implementation effort	5	4	4	3	5	3	5	5
New variant effort	2	2	2	2	2	2	3	3
Documenting effort	2	4	2	2	4	2	3	2
Runtime memory usage	4	4	4	3	4	4	4	3
Runtime processor usage	4	4	4	3	4	4	4	3
Modification effort	2	2	2	3	4	2	4	2
Debugging effort	4	4	4	4	5	2	5	3
Code structure	4	4	4	4	4	4	4	4
Code readability	4	2	4	4	2	5	4	4
Code complexity	2	4	2	2	4	2	3	2

List of Symbols and Abbreviations

Abbreviation	Description	Definition
AHP	Analytic Hierarchic Process	page 24
BA	Base Application	page 13
CBD	Component Based Development	page 24
DME	Development and Maintenance Environment	page 5
DSL	Domain Specific Language	page 18
GQM	Goal/Question/Metric	page 23
IDE	Integrated Development Environment	page 91
JS	Job Submitter	page 2
MCDM	Multi-criteria decision making	page 24
oAW	openArchtectureWare	page 49
SAAM	Software Architecture Analysis Method	page 21
SCM	Software Configuration Management	page 22
SoC	Separation of Concerns	page 24
SPL	Software Product Line	page 22
VRT	Variability realization technique	page 7
WA	Web Applications	page 2
WFPS	Wide-format printing systems	page 1

List of Figures

1.1 Overview of the Web Application architecture 3
1.2 The mainline and a branch of the Web Application 6
1.3 A model for variability 8

2.1 Architecture of the Base Application 13

4.1 Scatter of response on features 35
4.2 Spread of response on features 36

5.1 Component architecture of Job Submitter 41

A.1 Example variant selection process (VCS technique) 90
A.2 Example interaction between variant and invariant code (VCS technique) 90
A.3 Example file processing (IEV technique) 93
A.4 Example class processing (VCA technique) 95
A.5 Example file processing (VTI technique) 96
A.6 Example component selection (VAS technique) 99
A.7 Example interaction between variant and invariant code (VAS technique) 99
A.8 Example component generation (VCG technique) 101

List of Tables

4.1 Description quality scores 34
4.2 Efficiency scores 34
4.3 Maintainability scores 37
4.4 Combined description quality, efficiency and maintainability scores . . 37
4.5 Methods sorted by most positive features 37
4.6 Methods sorted by least negative features 38

6.1 Impact on qualities of variability realization techniques 73