Clemson University
TigerPrints

All Dissertations | Dissertations

12-2006

Managing Product Line Asset Bases

John Hunt
Clemson University, john.hunt@covenant.edu

Follow this and additional works at: https://tigerprints.clemson.edu/all_dissertations

Part of the Computer Sciences Commons

Recommended Citation

Hunt, John, "Managing Product Line Asset Bases" (2006). *All Dissertations*. 36.
https://tigerprints.clemson.edu/all_dissertations/36

This Dissertation is brought to you for free and open access by the Dissertations at TigerPrints. It has been accepted for inclusion in All Dissertations by an authorized administrator of TigerPrints. For more information, please contact kokeefe@clemson.edu.

MANAGING PRODUCT LINE ASSET BASES

A Dissertation

Presented to

the Graduate School of

Clemson University

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

Computer Science

---

by

John M. Hunt

December 2006

Advisor: Dr. John D. McGregor

Abstract

Product lines are predicated on collecting assets common to the desired product portfolio, commonly known as the asset base. For many product lines, the size of asset base has become large enough to create a variety of difficulties. The problem of size is particularly notable for implementation assets. Recent case studies suggest that asset base size also creates difficulties in deriving products from the product line assets, thus increasing the cost of producing a product from the product line assets. The techniques for managing large product line asset bases are unaddressed in the literature.

This research presents new techniques that take advantage of asset base characteristics, unavailable in more general collections, to both reduce the number of assets and to organize the asset base that go beyond what is possible with other software collections. The result is an asset base that is more efficient to use.

Research related to improving the organization of the asset base was performed by taking the component assets of a research SPL and arranging them based on three different organizational criteria - according to the structure of the architecture, important abstractions found during domain analysis (Key Domain Abstractions), and product features. The three resulting organizations were then studied using four evaluation criteria - natural division of assets into groups (assets fit into the groups provided by the organization), easy to map assets to organization criteria (mapping between the selection of a particular product variant and the assets needed to produce it), reasonably sized groups, and similarly sized groups. The effectiveness of the different organizations was then compared and recommendations concerning asset base organization provided.

The literature indicates that large product lines are likely to contain multiple assets that provide the same functionality, but that differ in the program context that they support. The presence of the duplicative assets creates a number of problems including organization difficulties. In a Software Product Line (SPL) these differences in program context are the result of requirements expressed at the product's variation points. This limits the difference in program context to three characteristics - cardinality, optionality and feature interactions. The limited differences in program context make it practical to attempt to provide a modular solution which permits the desired variation to be assembled as needed. The research explored a number of different implementation mechanisms to provide these modular variation points. The result is a recommendation on how to implement SPL variation points provided in the form of a pattern language.

Table of Contents

Page

Title Page . . . . **ii**

Abstract . . . . **ii**

Table of Contents . . . . **iii**

List of Figures . . . . **vi**

List of Tables . . . . **vii**

List of Source Listings . . . . **viii**

1 Introduction **1**

2 Background **4**
2.1 Software Reuse 4
2.1.1 Product Lines 4
2.1.2 Background of Reuse 7
2.1.3 The Software Library 8
2.1.4 Context 12
2.1.5 Summary of Software Reuse 15
2.2 Description of Asset Base 15
2.3 Measuring the Asset Base 17
2.4 Pedagogical Product Line 19
2.5 Hard Goods Assembly 21
2.6 AspectJ 23

3 Related Work **25**
3.1 Indirect Asset Access 25
3.2 Minimizing the Number of Assets 25
3.2.1 Component Adaptation Work 25
3.3 Implementing Variability in Product Lines 27
3.4 Organizing Asset Bases 28
3.4.1 Alternatives to Organizing the Asset Base 29

4 Research Strategy **30**
4.1 Motivation 30
4.2 Research Method 32
4.3 Research Questions 33
4.4 Asset Reduction vs. Asset Organization 36
4.5 Summary 37

5 Minimize Assets **38**
5.1 Introduction 38
5.2 Implementation Mechanism and Binding Time 39
5.3 Specifying dimensions of product line variability 41
5.3.1 Cardinality 41
5.3.2 Optionality 43
5.3.3 Feature Interactions 44
5.4 Approach 45
5.5 Criteria for Evaluating the Experiment 47
5.6 Description of Experiments 47
5.6.1 Scoreboard Variation Point 48
5.6.2 Services Variation Point 49
5.7 Implementing VCS characteristics 50
5.7.1 Implementation of Feature Optionality 50
5.7.2 Implementation of Feature Interactions 53
5.7.3 Implementation of Single Value Cardinality 55
5.7.4 Implementation of Multi Value Cardinality 61
5.8 Generalizing the Implementation 70
5.9 Conclusion by Implementation Approach 72
5.9.1 Parameterization 73
5.9.2 Inheritance 73
5.9.3 Java Language Interface 74
5.9.4 Aspects 74
5.9.5 XVCL 75
5.10 A Pattern Language for Implementing Variation Points 75
5.10.1 Introduction 75
5.10.2 A Pattern Language for Variation Points 76
5.11 Evaluation of the Experiment 86
5.12 Conclusion 86
5.13 Effect of Asset Minimization on the Asset Base 88
5.13.1 Calculations for a Single Value Variation Point 88
5.13.2 Calculations for a Multi Value Variation Point 89
5.13.3 Conclusions 90

6 Organize Assets **91**
6.1 Background 91
6.1.1 The Effect of Size 92
6.1.2 Hierarchical Organization 92
6.2 Experimental Design 94
6.2.1 Evaluating Component Organization Approaches 94
6.2.2 Considerations in Organizing Components 96

6.2.3 Organization Approaches 97
6.3 Experimental Results 102
6.3.1 Key Domain Abstraction Organization 102
6.3.2 Architectural Organization 104
6.3.3 Feature Based Organization 106
6.4 Discussion 108
6.5 Conclusion 109

7 Future Work 111
7.1 Organize Assets 111
7.2 Minimize Assets 112

8 Summary and Conclusions 113
8.1 Summary 113
8.2 Organize Assets 114
8.2.1 Research Questions for Organize Assets 114
8.3 Minimize the Number of Assets 116
8.3.1 Research Questions for Minimize the Number of Assets 118
8.4 Conclusions 119

Bibliography 121

List of Figures

Page

1.1 Role of the Asset Base in SPL 1

2.1 PPL Feature Diagram 20

5.1 Cardinality Notation 42
5.2 Possible Ways to Express Cardinality 42
5.3 Concerns at the Variation Point that define the Variability Characteristic Space 45
5.4 Scoreboard Object Diagram Base Case 48
5.5 Class diagram of initial scoreboard implementation 48
5.6 Role of services in MVC architecture 49
5.7 Practice Mode implemented using a decorator pattern 54
5.8 Practice Mode implemented using Aspects 54
5.9 Practice Mode implemented using XVCL 55
5.10 ScoreBoard UML 57
5.11 Including a Digital scoreboard using the first aspect approach 58
5.12 Including a Digital scoreboard using the second aspect approach 58
5.13 Insert a Digital scoreboard using the first XVCL approach 59
5.14 Insert a Digital scoreboard using the second XVCL approach 60
5.15 Services: Monolithic vs. Modular 62
5.16 Services implemented with Aspects 66
5.17 Services implemented with XVCL 67

6.1 Key Domain Abstractions 98
6.2 Model - View - Controller Architecture 100
6.3 Top Level Feature Diagram 101
6.4 Organization Tree for Key Domain Abstraction based organization 103
6.5 Organization Tree for Architecture based organization 105
6.6 Organization Tree for Feature based organization 107

List of Tables

Page

4.1 Examples of asset base sizes . . . 31
4.2 Relevant Situations for Different Research Strategies . . . 33

5.1 Effects of Cardinality on Implementation . . . 43
5.2 Comparison of mechanisms implementing scoreboard . . . 60
5.3 Comparison of mechanisms implementing services . . . 69

6.1 Components Sorted based on Key Domain Abstractions . . . 102
6.2 Components Sorted based on Architecture . . . 105
6.3 Components Sorted based on Features . . . 107
6.4 Approaches compared by criteria . . . 109

List of Source Listings

Page

5.1 Code to omit scoreboard using Java 51
5.2 Code to omit scoreboard using AspectJ 52
5.3 AspectJ point cut to insert scoreboard code 52
5.4 Code to omit scoreboard using XVCL 52
5.5 Example of a feature interaction implemented with AspectJ 55
5.6 Example of a feature interaction implemented with XVCL 56
5.7 Example Implementing Multi Selection with XVCL 67
5.8 XVCL Implementation of the Practice Mode Feature Interaction with the Scorebaord Variation Point 71
5.9 XVCL Template to Implement Any Feature Interaction 71

Chapter 1

Introduction

The Software Product Line (SPL) approach is fundamentally a reuse strategy. To achieve the economies that justify the use of a software product line, the software product line must reuse assets common to multiple products [Clements and Northrop 2002]. The collection of assets related to software development that supports a product line is called an asset base. It differs from other collections of software assets, such as libraries, in that the asset base's membership is scoped, or limited, to those assets needed to create a particular predefined portfolio of related products. While a great deal of research is being done in SPL, little attention has been paid to the nature of the asset base on which the enterprise ultimately rests.

Optimal organization of the asset base in a product line depends on how the assets will be used to produce products, (Figure 1.1[Hunt 2006]). It has been assumed that collecting the appropriate set of assets will automatically lead to the desired result of economically produced products. "Once the product line asset repository is established, there is a direct savings each time a product is built, . . ." [Clements and Northrop 2002]. How to create the right assets and how to improve the production process have been extensively studied, but understanding the asset base itself has received little attention.

Industry experience suggests that simply having the right assets is not sufficient to allow easy assembly. One multi-company study concludes "deriving individual products from their shared software assets is a time- and effort-consuming activity" [Deelstra et al. 2004]. One problem is the difficulty of finding and selecting the desired asset. This difficulty should not be a surprise as it has long been known: "To reuse a software

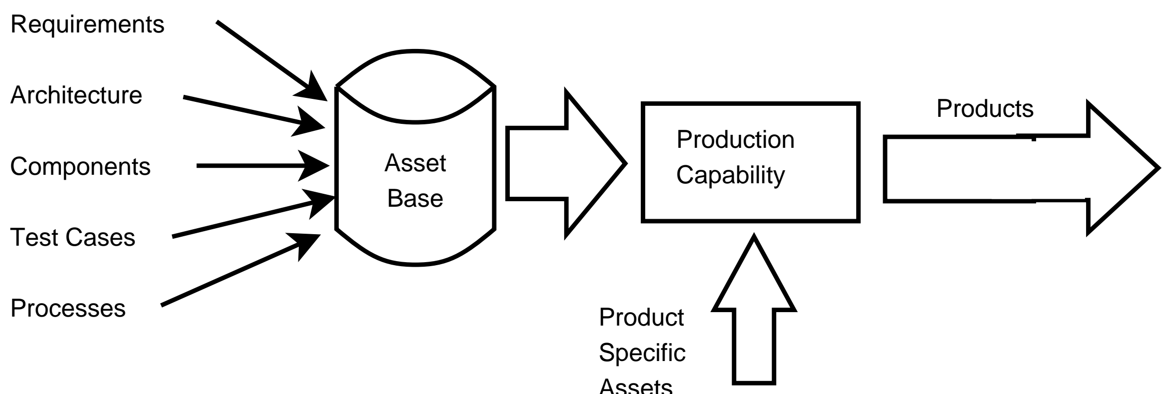

Figure 1.1: Role of the Asset Base in SPL

asset effectively, you must be able to find it faster than you could build it.” [Krueger 1992]. While the focus provided by scoping reduces the number of assets collected, as compared to more general reuse cases, the desire to build all of the products in the product line, combined with the complexity of applications that product lines have been applied to, results in large asset bases.

A necessary step in product derivation is to determine which assets will be used to build the particular product out of the possible products in the product line. This requires some description of the product that allows it to be distinguished from others in the SPL. This description provides a set of product requirements in some form or another. The assets needed to meet the product’s requirements must be found and selected. Often it is the product developer that makes these decisions as the product is assembled. The specific assets needed could be specified in a production plan [Chastek and McGregor 2002] or even a tool; however, using a plan (or tool) changes only when selections are made, not the need to make the decisions.

Finding the optimal organization for the asset base can improve the product production process. The need to formalize the coordination of various production activities has become accepted, particularly for those product lines that separate the creation of assets from building products which use those assets. Part of the production planning process should specify the organization of the asset base. The asset base provides a mechanism to buffer the outputs and inputs of various production activities, helping to provide an appropriately loose coupling between the activities.

Asset base sizes of up to 4 million lines of code (LOC) have been reported in the literature [Dikel et al. 1997] Previous experience with human ability to manage complexity on this scale suggests that this should produce problems. Deelstra et al. [Deelstra et al. 2004][Deelstra et al. 2005]report a number of difficulties in deriving products that seem related to asset base size. These include human error in configuring products, redevelopment of already existing assets, multiple interface variants of the same asset, irrelevant and voluminous documentation, and erosion of documentation. Difficulties in product derivation are particularly significant, and product line proponents typically concede that reusable assets are more expensive to produce, but argue that this expense is made up through greater efficiency in producing products.

The problem is to some degree unavoidable, since large complex software product lines will require large complex asset bases. However, it should be possible to manage the asset base in a manner that reduces difficulty, and hence the cost, of choosing from among the stored assets. The management of any collection may be approached by:

1. Minimizing the quantity of assets needed to provide the desired functionality.

2. Organizing the assets collected to improve usability.

The techniques that may be used to support these goals will vary depending upon the nature of the collection. The contribution of this research is to provide new techniques that take advantage of asset base characteristics both to reduce the number of assets and to organize the remaining assets, and that go beyond what is possible with other software collections.

The number of assets that must be considered to perform a desired operation increases the operation's difficulty and cost. Factors that affect the assets that the user must consider include the total available assets, particularly similar assets, and the organization of the assets. Complementary techniques are presented to reduce the number of similar assets and organize the remaining assets. Which approaches should be given priority depends upon the particular situation. Both approaches may reach a point of diminishing returns.

Suggesting that the number of assets collected should be minimized goes against a common feeling that when collecting assets, more is better. However, having more assets makes it harder to find which assets are actually needed. Acknowledging that keeping assets in a collection has ongoing costs provides a basis for attempting to eliminate unneeded assets. Also, collecting an asset is not the only way to obtain its functionality. In some cases it may be better to generate the asset as needed; however, the addition of a code generation capability adds tools to the development environment. Hence, there is a trade off between simplifying the asset base (by storing fewer assets), but having a more complex development environment (due to the additional tools and the skills needed to make effective use of them).

Organizing assets, by allowing some groups of assets to be ignored during a particular search, can reduce the number of assets that need to be considered. The classic approach to grouping items that will be searched is to arrange them hierarchically. The issue is to provide an organizing method that is general, yet divides the asset base for different types of products in a natural way. Different contexts, such as asset type or development activity, may require different organizing techniques.

A fundamental part of the product line approach is to scope carefully the products to be created. This significantly reduces the assets that are needed when compared to more general reuse approaches [1]. Nevertheless, software product lines have reached a point where the size of the asset base is causing difficulties; therefore, it is time to examine better ways to manage an SPL asset base. When implementing a product line we should minimize the number of assets and provide a carefully considered organization for those that remain.

[1] In general purpose reuse, where the assets provided are not tied to a particular set of products, the explosive growth in size of the collection has long been noted [Krueger 1992], [Batory et al. 1993], [BiggerStaff 1994]. Even within SPL, the effect of a wider scope than necessary on the number of assets produced has been noted, [Schmid 2002].

Chapter 2

Background

2.1 Software Reuse

As we arrive at the half century mark of software development, building rather than reuse is still the dominant paradigm in the software industry [Biddle et al. 2003]. SPL is the first approach to software reuse that has succeeded in building commercial products that are primarily assembled rather than built. SPL products frequently have reuse levels of over 90 percent [Clements and Northrop 2002]. In order to achieve this, SPL puts many restrictions on how reuse is done - restrictions that might be considered onerous if more general reuse approaches worked.

A background of modular reuse is presented, beginning with SPL, followed by a history of reuse, focusing on why general reuse has failed to provide the expected results. Finally, program context, which explains why SPL methods have succeeded where earlier, more general reuse methods failed will be discussed. Noting the difference in context found with SPL provides an opportunity to solve a number of problems that have been intractable in the general case. Specific aspects of SPL context are leveraged in both of the techniques presented in this research.

2.1.1 Product Lines

“A *software product line* is a set of software-intensive systems that share a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.” [Clements and Northrop 2002]

The idea behind the product line approach can be traced back to Parnas’s paper on what he termed program families [Parnas 1976]. He defined program families as those “whose common properties are so extensive that it is advantageous to study the common properties of the programs before analyzing individual members”. His observation did not have a lot of immediate impact, perhaps because most development organizations of that period developed software for internal use and thus needed only one variant. Also, to maximize reuse, we need to know which variants are desired at the beginning of the design stage, which requires a greater upfront understanding of product requirements than was generally available at that time.

The essential element in the ability of the SPL approach to provide products containing a high level of code reuse is identifying which products will be included. This set of products is referred to as the product portfolio. Product lines provide a structured, pragmatic, answer to which products to include. Look at the market, decide which products make business sense, and then limit the development to what is necessary to develop that set of products [Schmid 2002]. Additional analysis may result in some of these desired products being excluded from a product line because they are too different from the others, and time may bring to light other similar products that are to be added to the product line. However, what is important is the establishment early in the development process of a group of products that will be designed simultaneously.

Product line engineering inserts an additional step into the software development process to identify the variations between proposed products. The requirements for the proposed products are analyzed for product features, features which are visible to users [Kang et al. 1990]. Many features will be common to all products in the SPL; these are called common features. Common features, apart from any contained variation points, can be implemented using conventional software engineering methods. However, some features will differ between products; these are called variant features. For example, a low-end version of a product might exclude a feature. Or a product line might be offered on multiple platforms. Or some products might allow a user to choose between different implementations of a feature, such as interactive vs. batch spell checking for a word processor. Often, when the term *feature* is used without qualification, it means variant feature. The differences between products in the product line should be explainable in terms of variant features [Svahnberg et al. 2005].

The features in a product line are described by a feature model [Kang et al. 1990]. The relationship between a feature and the product can be described with a feature type. While there is some variation in the literature on feature types, a typical list of types would be: mandatory, optional, or a choice from a set of possibilities. A number of notations have been proposed to model the feature type relationship. The feature model is one of the inputs into the product line architecture. The architecture must allow for all the variation specified by the feature model.

Product lines frequently associate an additional dimension of choice of binding time with the feature [van Gurp et al. 2001]. The binding time is the point in the development process when a selection from among the different variations is made. For example, the point in the development process when an optional feature is either included or excluded from a particular product is the binding time for that feature. With study has come a lengthy list of different binding time possibilities (compile-time, activation-time, load-time, run-time, preprocessing-time, linking-time, loading-time, etc.), many of which are language or platform dependent.

One of the most important common assets developed for a product line is the architecture [Clements and Northrop 2002]. The architecture of a product line differs from that of a conventional program in that it describes all of the products in the product line, rather than a single product [Perry 1998]. The places in the architecture that will be different between products are described using variation points, which are derived from the feature model. A single feature may affect the architecture in multiple places, and thus be involved in multiple variation points. These are the same conditions needed to allow modular assembly of products, an advantage over other development paradigms that has been only occasionally noted.

In one sense, feature variation is what defines the product line approach [Thiel and Hein 2002], because without it the result would be a single product instead of a portfolio of products. Yet, what distinguishes SPL from most other reuse approaches is a rejection of open-ended variation. As Svahnberg et al. explain: "Once a variant feature is identified, it needs to be constrained . . . the purpose is not to provide limitless flexibility but to provide just enough flexibility to suit the current and future needs of the system in a cost effective way . . . To increase the understandability of the source code, facilitate maintenance and reduce the risk of introducing bugs, it is desirable to keep the variation points as small as possible." [Svahnberg et al. 2005]. The variation must serve the products.

Implementing a product line can be seen as a two-track approach. One track is to implement the common assets and collect them in the asset base. The other track is to implement individual products, taking as much advantage as possible of the asset base. Ideally, the asset base is such that the individual products can be derived from it in a highly automated fashion. In this case, a tool produces a customized product, based on input selections by the product assembler.

SPL literature has focused on the early (prior to implementation) phases of software development. Discussion of implementing a SPL is limited and focuses on issues involved in implementing variation points, such as feature types and binding times. For example, single inheritance can be used where only one option out of a set of possibilities can be chosen. Most development organizations will settle on a few techniques that are compatible with their chosen platform and the sort of variability that they most frequently implement. Lack of implementation advice regarding product lines has been noted in the literature [Anastasopoulos and Gacek 2001][Brown et al. 2002][Muthig and Patzke 2002][Anastasopoulos 2004][Muthig and Atkinson 2002].

According to the Software Engineering Institute "Software product lines represent a significant departure from software reuse schemes in which attempts are made to make assets as general as possible without the context provided by an architecture and a scope definition, and from opportunistic reuse schemes in which low-payoff assets are scavenged ad-hoc from a reuse repository."[SEI 2005]. While helpful, this leaves open

the question of why general reuse schemes have not worked. Examining these differences will highlight characteristics that are provided by SPL, but not other reuse approaches. Focusing on these characteristics provides new areas to exploit.

The next section discusses traditional approaches to reuse, centering on the software library. This will finish providing the background needed to discuss the key, but typically un-remarked, characteristic of SPL - a constrained context.

2.1.2 Background of Reuse

Kruger [Krueger 1992] has one of the more recent surveys on reuse; he defines software reuse quite broadly, exploring eight approaches. Of these, seven of the approaches focus on the implementation phase, the other being reuse of architectures. Of the seven implementation-based approaches, two are based on language, two are based on generative techniques, one is ad-hoc reuse (code scavenging), and two are based on assembling pre-written pieces of code. This last category, assembling pre-written code, is what most would identify as reuse and is the approach that will be the focus of this section.

Credit for suggesting that programs might be assembled rather than written is generally given to McIlloy for his paper “Mass Produced Software Components”[McIlroy 1968], presented at NATO’s Software Engineering conference in 1968, generally considered the first conference on software engineering. McIlroy drew an analogy with the integrated circuits (IC) that had recently become popular for producing digital hardware of various sorts. He went so far as to use the term “Software IC’s”. McIlroy’s vision was to be able to purchase pre-written software components from a catalog that would then be assembled to form a completed product.

Reusing pre-existing code pre-dated McIlroy. Ada Augusta Lovelace wrote reusable routines for Babbage’s analytical engine. Of more immediate relevance, most computer vendors in McIlroy’s day shipped sizable libraries of software functions. However, the focus of these libraries was to provide an interface for operating system functionality, such as I/O routines. The libraries were seen as a way to make platform (hardware / operating system) functionality easier to access. What was new in McIlroy’s paper was the idea of building a software product by assembling it from pre-written modules. This goal accelerated the growth in size, variety, and complexity of modules beyond what was required to provide a better platform abstraction.

Consider the steps needed to reuse a pre-existing module. Conceptually, there are four steps - find the module, understand the module, adapt the module, and use the module [Dusink and van Katwijk 1995]. While this is a logical sequence in which to list the steps, it is difficult in practice to separate them. It is hard

to try to look for something without some notion of what it is. It is hard to know if you have found what you want without understanding it to some degree. For reuse of a module to make sense, the sum of the cost or effort of these four steps must be less than the cost or effort of building the same functionality.

A key choice is the granularity of modules provided [van Ommering and Bosch 2002]. One approach is to have lots of simple lowest common denominator functions; this can be thought of as the "Lego" approach. However, unless these functions are very commonly used, the overhead of finding and understanding them will exceed the small amount of functionality they provide. Also, even if it is possible to build the product using the small, simple components the amount of effort saved will be limited.

Alternately, a module can supply functions that handle large pieces of work for the application developer. However, the functionality provided will become increasingly specialized and thus will be applicable to increasingly fewer situations [Tsichritzis 1989]. In addition, as the functionality increases it becomes hard to avoid an impact on the rest of the program [Shaw 1995]. It is desirable for a GUI library to handle keyboard and mouse events rather then just drawing the screen. To do this they need to read these devices. This quickly leads to a need to provide an event loop. To get the event loop to work the GUI library typically provides the program's main procedure. This in turn dictates how the application can be structured [Mattsson et al. 1999].

Small components quickly exceed the ability of the user to remember what to use relative to the amount of functionality provided. Large components can overcome this by providing a lot more functionality per component; however, they interact with the rest of the product in more complex ways, making them difficult to use without expensive adaptation.

2.1.3 The Software Library

The reuse sequence (finding, understanding, adapting, and using) assumes having collection of modules available, normally called a library. While it is possible to use a library to collect many different sorts of software development assets, in practice the focus on implementation assets has been overwhelming.

The first step in building a library is deciding what to include. Here libraries differ from a conventional program. A program is normally created to solve some particular problem. The functionality involved, particularly if the wish lists of various users are included, may become quite extensive; however, we will eventually completely fulfill the need. This provides an upper boundary, possibly quite large, on the program. On the other hand, libraries are intended to be useful building many programs, including those that have not yet been considered. As a result, there is no natural limit on which modules should be included in the library.

The most obvious approach was to assume that more is better; that more modules would offer a better

chance of providing just the module that is needed. The result was an effort to provide large libraries that would do everything. When it became apparent that there were too many possible modules that could be written, companies attempted to fill the library from donations [Frakes and Fox 1995][Lubars and Iscoe 1993][Neumann 1993]. After all, they were already writing a lot of software that did a lot of things. Surely, it should have been possible to break it into pieces and stash it away into a giant library to be reused throughout the company.

This approach of providing a "general purpose" library failed for at least two reasons. First, large libraries are large, which makes it difficult to find things in them. For many years the research community attempted a wide variety of techniques to improve the ability to find things in a large general purpose library [Mili et al. 1998]. None of the techniques showed a great deal of success at finding things, and many of them were quite expensive to implement [Frakes and Pole 1994][Mili et al. 1997]. Second, even when a module was found, it usually could not be used without a great deal of adaptation [Lubars and Iscoe 1993]. Modules that are not designed to work together don't [Behle 1998][Dusink 1992]. The traditional approach of applying stepwise refinement [Wirth 1971]to a design is quite unlikely to arrive at a point where a pre-existing module happens to fit without adaptation [Tsichritzis 1989]. Performing the adaptation meant first understanding the internals of the module. Thus, the cost of adaptation combined with the cost of finding the module and maintaining the collection exceeded the cost of building the functionality.

As a result, companies that had gone to the trouble to assemble large general purpose libraries that contained code from every project in the company gave up and turned responsibility for code reuse back to their product divisions [Neumann 1993][Poulin 1994][Frakes and Fox 1995][Arango et al. 1997]. An observation that arose from this process was that assets fell into different categories. There are some assets that are generic enough to be useful across products; then there are assets that are specific to a particular product domain [Poulin 1994].

Generic assets have largely migrated to language-based libraries and are often considered to be part of the platform. To succeed, they must be simple to understand, used often enough for developers to build some level of familiarity with them (at least enough to prompt a search), and able to be used with a variety of architectures without requiring too much change to the asset or impact on the architecture. There have been some notable successes in these generic assets. Collections, Data Communications (starting with the now ubiquitous socket interface), GUI's, and product database interfaces (JDBC, ODBC, etc.) are all examples where modular reuse has largely replaced building in areas that are considerably more complex than earlier libraries, which focused on better hardware abstraction. Interestingly, when a library becomes part of a

language platform, many no longer think of it as reuse; it has blended into the environment to a point where it is no longer considered [Bosch 2000]. Of course, this means that if we ever do reach a point where we assemble products completely from modules, we will have no reuse at all!

The remaining non-generic assets were considered to be aligned with a specific product or business domain. The next logical step seemed to be the creation of libraries for a particular product domain that might be useful for many products produced by many companies operating the domain [Browne and Moore 1997][Behle 1998]. At least two developments encouraged this thinking. First, returning centralized libraries back to business units worked better, and for the most part these business units were based on product domains. Second, the idea of domain analysis, that a domain needed to be understood before software could be successfully developed for it, came to the forefront. Domain analysis led to the idea that there were architectures appropriate to a given domain, which reached its peak in DARPA's Domain Specific Software Architecture project (DSSA) [Gacek 1995]. Assets collected based on domain received the acronym, DSSR (Domain Specific Software Repository) [Tracz 1994].

The goal of a single large general purpose library was replaced by the model of a platform library and smaller, more specialized, libraries based on product domain. In turn, it made sense to share domain expertise and increase reuse by sharing these specialized libraries [Browne and Moore 1997]. This approach can be seen in the STARS project (Software Technology for Adaptable, Reliable Systems, funded by DOD ARPA), conducted in the late 1980's by a number of leading defense contractors that resulted in the IEEE 1420.1 standard for library interoperability [IEEE 1995].

Small, specialized libraries provided a pragmatic solution to the problem of searching for assets without the need for the sophisticated approaches that had been (and still are) occupying researchers [Poulin 1999][Gacek 1995][Barroca et al. 2000]. It also greatly helped the problem of understanding assets, both limiting the subject addressed and standardizing the vocabulary and the relationship between elements that was accomplished by domain analysis and the related domain architecture.

Unfortunately the results from this reuse approach of implementation assets have been limited. There is an important truth embodied in attempting to provide domain-based libraries. A group developing banking software is unlikely to find an avionics-related asset of interest. However, this does not mean that a banking asset developed by one group will fit into a banking product developed by a different group. Matching domains turned out to be a necessary but not a sufficient criterion for being able to reuse an asset.

One of the places this problem surfaced was in attempting to get different products, each intended to be extensible, and to work with other products. A good early example is presented by Garlan et al. [Garlan et al.

1995]. Here a team attempted to use several independently produced software tools to automate multiple stages of a development project. Each of the tools had expected input formats, which the team was able to provide, with some adaptation, from the other tools. This allowed the data to be passed, but did not address the control flow between tools. Some tools were expected to operate in a filter fashion, waiting on input, and then processing it through to output. Others expected data to be handed back and forth in pieces, a sort of procedure call approach. Some expected to control when other stages of processing were called, in a master / slave fashion, while others didn't. Unlike the data formats, which were exposed and had explicit mechanisms to import and export data, the control flow was buried in the implementation.

Given that program level control issues were noted early on, this problem was called "architectural mismatch". This term has come to be used in several related but different ways. In its most general use, it can refer to any instance where two assets from the product domain fail to work together as expected. In a more specific sense, architecture refers to the top level of a programs design and its related structure and control flow. Modules smaller than a program and that cannot run independently don't have an architecture in this sense. Therefore, to have architectural mismatch, in the strict sense, a comparison between two different things that each have an architecture must be made. If we have a problem using a module from within a program, perhaps because it throws exceptions in the event of errors and the rest of the code does not handle exceptions, this is more properly termed packaging mismatch [Berlin 1990][Deline 1999]. However, packaging mismatch can be referred to as a type of architectural mismatch, using architectural mismatch in the more general sense.

The idea of architectural mismatch brought with it the realization that implementation assets, even if well designed and well written, would not work together unless specifically designed to do so. One result of this can be seen in the essential absence of a software component industry [Greenfield and Short 2004]. This is very different than most industries in which component suppliers have increased dramatically in importance. Another is the large-grained and very loosely coupled model provided by service oriented programming, in which large units of work are passed across a network for further processing. This model is able to isolate the differences in processing via the network protocol and thus ignore the differences between these large-scale components.

Despite decades of effort, the general problem of building components that can be assembled into products remains open. Yet, organizations that use a product line approach can reliably assemble products, in some cases having thousands of variations. To understand why, the focus must be changed from the module being built to the environment where it will be used.

2.1.4 Context

Context is anything in an asset's environment that affects whether it works correctly. The impact of context is not unique to software but affects, at a minimum, any artifact produced by man. We can see both an example of the effect of context and the degree to which software engineering has mostly ignored the issue going back to McIlroy's initial proposal of 1968 [McIlroy 1968]. McIlroy used the example of an integrated circuit (IC) catalog as a model for a complete set of software components that could build any desired software product. A quarter century later, Krueger, in his survey paper "Software Reuse" [Krueger 1992], re-invoked the same model, comparing the desired catalog of software components to the 'Texas Instruments "TTL Data Book"'. No doubt the comparison has been used many other times as well. What has been missed is that these IC's are not all-purpose products. The user of an IC catalog has a context in mind and only looks at those ICs that meet certain basic technical limitations. A particular family of IC's operates at particular voltages and frequencies. TTL's operate with an input voltage of 5 volts +/- 0.5 volts. If you have chosen a power supply that doesn't produce a voltage within this range then you don't look at the TTL catalog, since you intend to assemble, that is compose, the chip with the power supply. The components from the TTL catalog are appropriate for some types of products and not for others. The correct analogy for an IC catalog is not the general-purpose library, but one that is quite context specific.

Christopher Alexander provides a classic discussion of the problem of designing artifacts with his example of a teakettle [Alexander 1964]. He argues that there is no direct way to measure the fitness of an artifact. The best we can do is to put the artifact into its intended environment and notice where it doesn't fit; that is, find what misfits there are between the artifact and the context. There are limitless possible misfits in the way an artifact fits its context. This means that design of an artifact can never be reduced to a selection problem. In theory, the designer can never know prior to use if his design is acceptable. In practice, tradition can define the range of changes to be made to the artifact to a point that acceptable designs result with some predictability. Of course, in software we have less design tradition to rest upon than in most fields.

The extent to which tradition can be exceeded and still produce a working artifact is quite unpredictable. Here are some examples from other fields that have access to longer traditions than ours: In 1940 the Tacoma Narrows Bridge collapsed two months after completion, despite several thousand years of bridge building experience. The bridge failed for aerodynamic reasons, something that had not had significant effect on bridges previously. Electrical wiring in houses is another highly specified area. In the 1970's the introduction of aluminum house wiring caused a series of house fires. The problem was not so much the wire itself, but the

way it unexpectedly interacted with established wire connector designs, which produced a dangerous amount of heat. The properties of water pipes are specified in considerable detail in ANSI specifications, to allow pipe produced in different places to be used predictably in building. This fact, given the relative simplicity of pipes compared to software, should probably give us pause. As recently as the 1980s, an innovation away from tradition, the substitution of PVC plastic for cast iron, lead to the large scale installation of pipe which subsequently failed in normal use. This material proved to be too brittle to reliably survive the normal handling that occurred during installation. Each of these cases occurred in fields that were considered to be well understood, due to a misfit between an artifact and its context, and along a dimension that had always existed but not been previously considered.

Given that the problem of context has afflicted all other areas of designed artifacts, it should not surprise us that software is affected as well. What is more surprising is that its discovery was postponed as long as it was. Alexander has some observations that might help explain this as well. He discusses the existence of what he calls unsophisticated cultures. In this sort of culture, the user of a product is also the person that builds and maintains the product. The user knows how the artifact is misfit to the context, not through a theory or formal understanding, but by experiencing the pain of the misfit. He makes adjustments until the misfit is reduced to a level that he can live with. The user / builder /maintainer in the unsophisticated culture has an advantage in that the number of choices available to him, between those dictated by tradition and the limited level of technology, are manageable.

The early decades of the software industry operated in a way that paralleled the unsophisticated culture. The company that used the software also wrote and maintained it, or at least had considerable control over those that did. The user thus had some ability to fix the spots that pinched. However, the production of software has largely moved past the point where the same organization that uses the software also builds and maintains it. It has moved out of this mode for the same reason that societies moved beyond having individuals produce their own things; that is, specialized builders can produce more sophisticated artifacts at a lower cost.

Very little is known about the context required for software components to be reliably assembled [Seiter et al. 1996][Walker and Murphy 2000][Tracz 1991][Ossher et al. 2000]. The work on architectural mismatch has led to more formal architectural design and classification of architecture [Garlan 2000]. This has resulted in some work on predicting if two different products could work together [Gacek 1998]; however, it is not applicable within a product to the fit between modules. Deline identified seven different types of package mismatch [Deline 1999]; however, there is no reason to think this is a complete list. Given that other fields

of engineering have been unable fully describe the context in which they operate, and that theory of design suggests describing context fully is impossible, how is progress to be made?

The pragmatic solution is to limit the variation allowed in building products, from what is possible, or even desirable, to what is manageable. This is already done in a number of ways. Most organizations tend to do many things the same way out of inertia or ignorance of other possibilities. While often frustrating, it limits product variability. Returning control of reusable assets back to divisions from a central corporate library not only grouped the assets by product domain, but also by a host of other context factors, most of which passed beneath awareness. This is part of the reason it is much easier to achieve reuse within an organization than it is between organizations. It is probably a key reason why the intuitively appealing efforts to share specialized libraries, such as DARPA's STARS project, have had little impact.

Another example of pragmatically limiting the context can be seen in the development of Java, which has had a very successful platform library. The Java team chose not to attempt to have their language or libraries inter operate transparently with other languages[1], unlike Microsoft, which intended .NET to support multiple languages from the start. For example, the decision to support 'C' required that .NET to provide the ability to use pass by reference for function arguments. One reason for this was a deliberate choice to reduce complexity of understanding and meshing with the assumptions embedded in different language approaches, the language context. They also decided to bind a number of features at the language level that normal practice left open to product developers. A good example is memory management, where automatic garbage collection was built into the language. This makes the language less flexible and makes it unsuitable for some types of applications, such as those involving hard real-time processing[2]. However, it does produce a simpler context with several classes of mismatches eliminated, such as which component is responsible for freeing memory.

One way to be able to compose modules is to specify the context as much as possible, even if we do not have a full understanding of all the dimensions of context that we are choosing. Of course, if we completely specify all aspects of two different programs to be the same, they become the same program. A "reusable" module that can only be used one place in one program is not very interesting. What we need in order to

[1] Java has a number of ways to call and be called by code written in other languages. These include: Java Native Interface (JNI) from Sun, Java Runtime Interface (JRI) from Netscape, and Raw Native Interface (RNI) from Microsoft. All of these provide an explicit programming interface which the developer uses to bridge the language worlds. These require many Java features, such as garbage collection to be explicitly manged by the programmer. This is different from a strategy where compilers for multiple source languages use a common output format with the goal of allowing modules written in any of the sources languages to be called from modules written in any of the source languages using the same conventions that would be used to call a module written in the same source language.

[2] The first project of Sun's Java Community Process to evolve Java was an effort to provide real time extensions to Java (JSR 1). These changes include major modifications to the memory model to avoid garbage collection. An overview of the changes can be found in [Brosgol and Dobbing 2001]

fully exploit modular composition is a software development methodology that creates programs that are the same except for well-defined differences that are selected in a way that results in useful program variations. Conveniently, such a methodology has already been developed and proven in use - software product lines (SPL).

2.1.5 Summary of Software Reuse

Product lines differ from conventional reuse approaches in that the ways reuse will occur are planned from the beginning of the process. This happens by scoping a product portfolio and by doing a variability analysis based on product features from the beginning of the process. Products designed this way have a highly shared context, which in turn simplifies the problem of fit among implementation assets, making it easier to reuse an asset without first adapting it.

Product line literature has focused primarily on the planning and design needed to produce a desired product portfolio. Secondary consideration has been given to the problems of implementing product line variability, which is implementing variation related to feature choice. Essentially ignored have been issues involving collection and management of assets. Yet, SPLs are primarily a reuse strategy, and it has been noted you must have and be able to find an asset before you can reuse it [Krueger 1992]. While this problem is ameliorated by the product line scoping process, it is not solved. As the size of the asset base grows, many of the problems involving earlier, more general collections for software reuse, such as libraries, have come to affect product line asset bases as well.

2.2 Description of Asset Base

Clements and Northrop provide a typical description of an SPL asset base: “The asset base includes those artifacts in software development that are most costly to develop from scratch - namely, the requirements, domain models, software architecture, performance models, test cases, and components. All of the assets are designed to be reused and are optimized for use in more than a single system. The reuse with software product lines is comprehensive, planned, and profitable.”[Clements and Northrop 2002]

The goal of the asset base is to assist in creating a particular portfolio of products. One of its distinguishing characteristics is that its membership is scoped, or limited, by a decision to create some products and not others. While the choice of products in a product line is subject to evolution, an initial decision on the members of the product portfolio should be made prior to collecting the assets.

The primary operation provided on an asset base is the derivation or production of products. This is different from other reuse collections, such as libraries, which are not expected to contain the assets needed to build a complete product. This means that the principle user, the customer, of the asset base is the product assembler. A product line asset base may be used to derive thousands of different products a year [Deelstra et al. 2005], making the cost and complexity of derivation a significant issue. Methods of assembly vary from largely automated to manual assembly similar to traditional software development. While the primary use of the asset base is product derivation, it must also support product line asset development and maintenance.

Assets bases are heterogeneous and support all phases of development. Indeed, Clements and Northrop's list of artifacts focuses on the early (pre-implementation) phases of development. There are several sound reasons for this. Software engineering has increasingly emphasized the importance of the early development phases. Product line engineering extends this emphasis due to its emphasis on reuse and its acknowledgment of the difficulties of unplanned reuse. However, an excessive focus on early development ignores the goal of creating actual products. Customers buy products, not designs. The strategy of reuse applies to all phases of development, but we usually measure its success in terms of the code reuse levels of the final products. Clements and Northrop justify their study of SPLs based upon finding products with "software reuse levels in the 90 percent range" [Clements and Northrop 2002]. While we talk about the importance of design, we measure the implementation.

Relationships between assets are also important. Assets in the asset base are intended for use by more than one product in the portfolio. However, it is entirely possible that an asset will not be used by all or even a majority of products in the selected portfolio. This is a concrete way of saying that commonality is not a binary choice; rather, products have degrees of commonality. In addition, an asset may, and often will, use other assets. This may happen between assets of different types; for example, we hope that an implementation asset will be based upon a design asset and described by a documentation asset. Or it may happen between assets of the same type; for example, an implementation asset may use the services of another implementation asset. In general, the asset base as a collection should be self-contained except for using services provided by a platform.

Products are built using assets from the asset base. Typically, products will use multiple assets from the asset base, but a single product is unlikely to use all of the assets. A product may contain assets that are legitimately product specific and thus not a part of the asset base. Also, the asset base may contain options that are mutually exclusive. For example, we may have assets to build both a secure and non-secure version of a product, but no product should include both sets.

Each asset being assembled into a product has dependencies on other assets, and therefore must be compatible with those assets. However, not all assets relate to each other. The dependencies that an asset has establish the context, or environment in which it will be used. Thus, different assets will each have a different context due to the differences in dependencies that the asset has, providing a series of nested contexts. However, the addition of an asset to a product modifies the context affecting the assets, both those that have been assembled and those to be assembled. The context affecting each asset evolves as assets are selected and assembled. In the case of feature dependencies, selection of an asset may require the behavior of previously selected assets to change. One way to think of this is to consider the possible contexts as the cross product of the product line's variation points. As selections are made at variation points, context is built up. One consequence of this is that without careful design the order in which selections are made may affect the product that can be produced.

The issue of context exists for all software products, and particularly for all products that attempt to reuse pre-existing artifacts. What is different about product lines is that while the context is still very complex, it is defined well enough that attempts can be made to anticipate the variation that will be found in the context.

We typically measure the success of a product line by the degree to which products are built from assets in the asset base. We expect to see 80% or more of the product attributed to reusable assets, compared to 50% or less for other reuse approaches. Usually, a level of less than 80% suggests that the product is a poor fit to the rest of the product line or that there is additional commonality that has not been exploited. A 100% reuse level is often held up as a goal, but care should be exercised to insure that the cost of doing this does not exceed the benefits.

2.3 Measuring the Asset Base

To understand something it is necessary to measure it. What should be measured depends on what needs to be accomplished. The goal of interest in this research is how to minimize the cost of using the asset base to derive products. This goal is most appropriate for those organizations that will derive many products from a product line. This is difficult to measure; therefore, a proxy measure of how difficult it is to use the asset base will be substituted. Using this proxy measure means assuming that the more difficult an asset base is to use, the more expensive it is to use. There are two different aspects of asset bases that are of interest: first, the size of the asset base, on the assumption that more assets means that more assets have to be considered to perform an operation; and second, the variation of the asset base, on the assumption that more variation in

using the assets means that more decisions will have to be made to perform an operation.

It is necessary to decide both what to measure and how to measure it. The unit of measurement must be standardized in such a way that it produces consistent results across product lines, for the same reasons that we have given up relying on three barleycorns to determine an inch. It should measure the same thing in different development environments to allow comparisons. Since the asset base is heterogeneous, more than one kind of thing should be measured. For the most part the measures currently in use, to the degree they exist at all, are disappointing, not only for product lines, but software engineering as a whole.

Implementation assets are important to measure, because there are usually considerably more of them than other types of assets, and because the most common operations on the asset base involve implementation assets. There are a number of different implementation artifacts, such as components, classes, interfaces, and functions; however, these vary greatly in size, due to both design philosophy and technology. As a result, implementation size is almost always measured in Lines Of Code (LOC). This is problematic for a number of reasons. While standardized measurement definitions for lines of code have been developed [Jones 1986], there is no guarantee that they are used for the measurements reported. A number of case studies in the literature specify that they use the result from the Unix `wc` command to measure the LOC reported, which does not implement the standard metric. There is also the problem that different languages will require different amounts of code to provide the same functionality [Jon86]. Many product lines are written in C++, in Ada, or in Java, which, on the whole, are relatively similar in terms of LOC to provide the same functionality. Still, it should be noted that C, which has also been used in product line work, seems to require twice as much code as Java, C++, or Ada [Jones 1996].

There is one other established approach to measuring implementation artifacts, function point analysis [Oxford 2004]. Function points attempt to capture the complexity of developing functionality in a language independent manner. However, the focus of function point measurements is the amount of development effort to create an artifact. The focus of this research is on the effort to use the artifact after it has been created. This makes a function point a poor measure for this research. As such, function points will not be considered further.

The early development phases provide the most reuse leverage; also, the decisions made in the early phases are the most difficult to change; therefore, it would be good to have a measure related to the early development phases. It might also allow some measure of the efficiency or goodness of a particular implementation or implementation technology. Alas, the literature provides almost no examples of measuring early development artifacts. Our two basic questions of what to measure and how to measure it remain open.

While not a universal approach, it is common in product line engineering to begin the analysis process by grouping requirements into features [Kang et al. 1990]. One recommendation on deriving features from use cases is that there should be an order of magnitude reduction in the number of features from the number of requirements [van Gurp et al. 2001]. Features may be used to bundle other features in a manner that is both hierarchal and recursive, so counting them would require a procedure to determine something analogous to a base feature. Despite their widespread use, there is no standard measure for feature. The reporting of feature counts is rare in the literature.

The other aspect involved in using the asset base is complexity as reflected in the amount of variability. A common approach in managing product variability based on variation points. These are particularly interesting, because they affect not only the development of product line assets but also may be visible during product derivation and even, in some cases, during runtime product configuration. This means the complexity exposed through variation points may be visible not only to product line developers but also to product assemblers and, possibly, even to end users. A definition for variation points and some measurements using the definition can be found in [Sinnema et al. 2004].

One of the few papers on measuring SPL is Zubrow and Chastek [Zubrow and Chastek 2003]. Their interest is on managing the software development process, so their focus is primarily on issues such as predicting development time, predicting asset reuse, and estimating product cost. These management metrics do require some connection with the actual software. Here they recommend using either LOC or function points, without indicating a preference. Artifacts such as variation points, use cases, etc. are not considered as a basis for any of their measurements. This supports the choice of relying mainly on LOC to measure size for this research.

The section on research motivation includes examples of asset base sizes as found in the literature.

2.4 Pedagogical Product Line

Ideally the validity of the techniques developed in the research should be evaluated on an industrial product line. However, companies are unwilling to provide their code base for research purposes. This is particularly true in the product line area where the development of a product line typically confers a strategic advantage over competitors. This is not only a matter of code. A feature model for a product line indicates the likely evolution of a company's products, or more negatively what sort of competitive products it would be difficult for the company to respond to. The open source movement has made the source for many different products

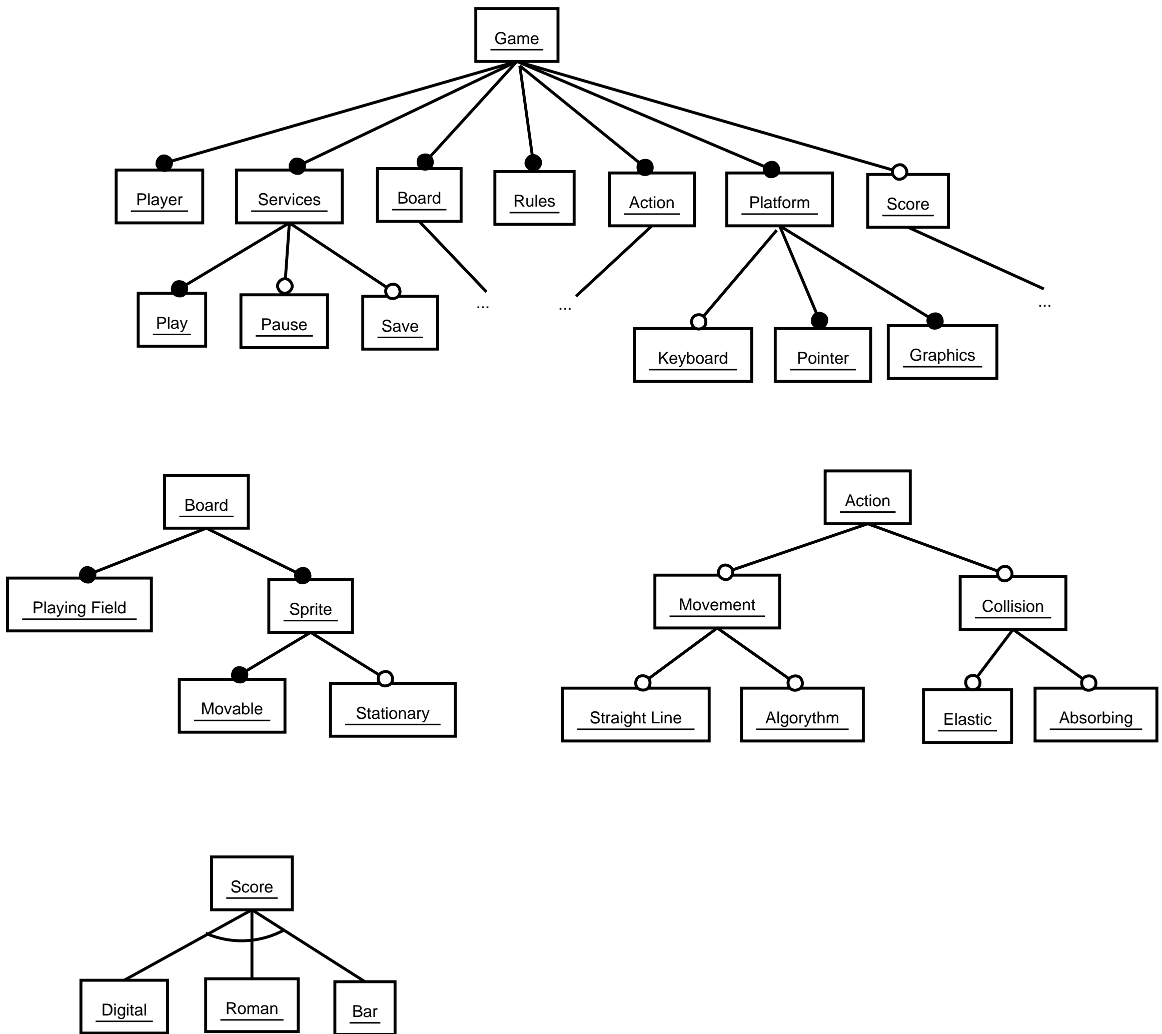

Figure 2.1: PPL Feature Diagram

available; however, none of these appear to be developed on a product line basis. Thus, open source projects lack the sort of early, non-implementation assets, such as feature model, that are needed for most SPL research. This problem of not having product line assets available extends to all sorts of academic purposes, not just this research.

The need for a complete set of product line assets for use in an academic environment has led to the development of a Pedagogical Product Line (PPL) [McGregor 2006]. The PPL models a software firm producing a product line of games for three different runtime environments, such as resource-constrained devices - cell phones, PDA, etc. The assets extend from corporate goals and organization through to a set of running products - Brickles, Pong, and Bowling. The PPL has been used in a number of courses at Clemson University and training seminars by the SEI. A current feature model for the PPL is provided in Figure 2.1.

To provide for all the scenarios required by this research, the PPL was modified in a number of ways. The initial version of the PPL used a modified Model-View-Controller architecture (MVC) that combined the model and view. The uses of a non-standard architecture was problematic in studying the use of architecture as an organization criteria. I re-factored the PPL code to fit into a standard MVC architecture. The current PPL has examples of mandatory, optional, and exclusive selection features. A feature called *Services* was added to provided an example of a multi-value selection. The addition of Services also helped to separate the model and controller code. Finally, a feature interaction example, based on a feature called *Practice Mode*, was added. Practice Mode also provides an example of a cross-cutting feature.

2.5 Hard Goods Assembly

In attempting to assemble software rather than write it, it would appear software development has become more like the manufacture of traditional hard goods. In both cases parts must be found and then fitted together. It would seem that there are some similar issues such as:

- In what order should parts be assembled?
- What are good sub-assemblies?
- How should parts be cataloged so that they can be found?

This suggests that perhaps lessons might be learned and techniques might be borrowed from hard goods assembly, generally regarded as a successful endeavor, and applied to the further improvement of software development, a field that is often considered to be in crisis. Unfortunately, an examination of manufacturing

literature shows that the differences between software and hard goods assembly are still more important than the similarities. There are at least two significant differences between software and hard goods:

1. For hard goods, function is a consequence of form.
2. For hard goods, manufacturing is more expensive than design.

For hard goods function is a result of form; that is physical characteristics. To take a simple example, given a washer of a given size, we are indifferent as to whether it will be used as a spacer between parts or to spread out the pressure of a screw. If we attempt to take this approach with software the closest we can come to physical[3] characteristics is a bit string with a particular combination of 1's and 0's. Occasionally, this "physical" view of software works. For example, in doing data compression we are happy to substitute the same compressed symbol for a given run of bits regardless of whether the bits are part of a program or a picture. Other places in which this applies are the movement of bits over a network and the movement of bits onto a device such as a disk drive. These instances are the exception. In general, we care about the functionality of a software component, and allow tools, such as compilers, to freely alter the bit pattern.

The physical manufacture of software is just a matter of copying the bits of the software system onto a physical medium, such as a CD. It is a cheap and trivial process. What we discuss as the assembly of software components is really comparable to the design of hard goods, not their manufacture.

Let's look at several examples from hard goods manufacturing literature to see the effect of these differences.

Product Design for Manufacturing and Assembly (PDMA) [Bothroyd et al. 1994] trades increased cost in the design phase for cost savings in assembly. A central tenant of PDMA is to reduce part count by combining parts that are functionally distinct, but physically similar; for example, mold the brushings for a motor as part of the motor's case. This is the opposite of what we generally aim for in software engineering, where we want to separate the parts that perform different functions. It also increases the cost of the development phase we would like to minimize, that is the design.

[3] What can be considered a "physical" characteristic of software is not obvious. The use of analogy is fundamental to human thinking and naturally comes into play when grappling with an area as abstract as software. Take the concept of robustness, the Merriam-Webster Online Dictionary [Merriam-Webster 2006] defines robust as "strongly formed or constructed" when discussing materials, but also as "capable of performing without failure under a wide range of conditions" and in fact it uses "robust software" as an example of the second meaning. However, in most cases the "wide range of conditions" is based on logical differences rather than physical differences. Consider a method that converts characters to upper case. If the program fails to handle punctuation characters its failure is base on an unexpected logical condition. There are cases where a program fails to be robust due to unexpected physical conditions. For example, the famous Ariane rocket failure, the problem with the method that failed was not that it could not handle the logical value passed to it; rather the length of the physical size caused the relevant bits to be clipped off.

In fact, since all software is composed of the same “material” - bits - applying this approach causes us to throw out classes, functions, packages, etc. to create a monolithic piece of code.

Another example comes from the design of manufacturing cells, where the assembly work is done. Let’s say we have two circuit boards that have completely different functions, but similar physical characteristics. For instance they both rely on wave soldering transistors to the board. Standard advice is that both boards should be manufactured at the same cell to take advantage of the equipment and the expertise of the assembler [Bothroyd et al. 1994]. This is somewhat analogous to using a particular workstation to write all the I/O code for dissimilar products. However, with software many different tools can be accessed easily from the same work station, whereas it would be physically difficult to put many different tools into the same manufacturing cell. Even for tools that are licensed for a limited number of “seats” these are typically virtual seats that float between workstations over a LAN. Software assembly is organized based on the logical characteristics of the product and the skill sets of the programmers used, not the characteristics of the tools or “materials”.

Finally, there are coding schemes in hard goods manufacturing to describe parts in order to automate the process of finding them. At first this seems like it might have some application to finding software components. Alas, these coding standards are based upon physical characteristics rather than functionality [Kamrani and Salhieh 2000]. They might describe a part as being a circle an inch across with a particular thickness and a certain number of holes punched in it. This is somewhat analogous to CASE (Computer Assisted Software Engineering) systems which could tell you all the methods in the system that took two integers as arguments and returned a double, but could not search based upon what the method did. The functionality of searching on a signature has dropped out of development environments. The underlying problem is that once again hard goods can ignore identifying the functionality of the part in favor of a physical description.

As these examples show the nature of hard goods remain too different from the nature of software to provide useful advice on assembly or even classification of parts.

2.6 AspectJ

Aspects provide a modular way to manage code that does not fit into the dominant decomposition of the product. This sort of code often becomes scattered through out the product. A classic example of this problem is that of logging, which will often touch almost every part of a product.

One of the most popular aspect languages is AspectJ, which provides aspect extensions to Java. AspectJ

uses “point cuts” to specify where the aspect code should be hooked into the existing code. There are several ways to specify when the aspect code should be called, known as “advice”. `Before` advice executes the aspect code before executing the method specified in the point cut. `After` advice executes the aspect code after executing the method specified in the point cut. `Around` advice replaces the method specified in the point cut with the aspect code. The code in the `around` advice allows conditions to be checked at which time it may either allow the code to proceed done the same execution path it would have taken without the `around` advice or to execute an alternate path. If the code in the `around` uses the `proceed` it is allowing the previous path of execution to continue, this differs from calling the routine specified in the `around` advice, in that there is a lot of information (call parameters, etc.) that it does not need to be aware of. One advantage of using `proceed` in `around` advice is that the same advice will work for all of the variant implementations.

Chapter 3

Related Work

3.1 Indirect Asset Access

My research assumes that product derivation involves finding and manipulating the actual assets, an approach that can be labeled as direct asset access. An alternate approach is to interpose a tool between the user and the assets and move the effort of finding the correct asset on to the tool, which I will label as an indirect asset access approach. A tool that successfully carries this burden minimizes the effect of organization on the user. An example of this approach can be found in the COVAMOVF related tool work [Sinnema et al. 2004]. This work focuses on providing a visual representation of the products variation points and ways to control the products configuration through the related GUI. While this may turn out to be a successful approach, substituting tools for asset management has not been a successful approach for the more general software library case.

3.2 Minimizing the Number of Assets

As noted in Deelstra et al. [Deelstra et al. 2005] assets with duplicative functionality were created to adapt the asset to different variation points, binding times, etc. This class of duplicate assets is eliminated through more automated packaging. Related work for this comes from both the area of adapting components in a general case (not product line specific) and the area of implementing product line variation.

3.2.1 Component Adaptation Work

The adaptation of components to their context is a theme that arises frequently in the pattern community. Looking at the patterns listed in Gamma, et al. [Gamma et al. 1995] 4 of 23 patterns (adapter, facade, bridge, proxy) can be considered as mechanism to adapt a component to its environment. Patterns typically focus on the program design level, and ignore many implementation level issues, such as signature mismatch.

The most detailed development of packaging mismatch at an implementation level occurs in Deline's work. He develops a list of seven different types of packaging mismatch and a list of techniques to overcome

them. His later work [Deline 2001] shows an approach to splitting a component into a functional part, called the ware, and a packaging portion. He then develops packaging that can adapt the ware to a variety of contexts. For example, he is able to use a ware as a COM component and as a filter in a UNIX pipeline. His implementation is to manually write generic adapter code to bridge between a known set of external interfaces, such as COM, and the internal interface native to the ware. Another example of work to resolve packaging mismatch is Callahan's [Callahan 1993], which looks at bridging the differences in functions written in different languages and RPC calls. Callahan uses an extended build tool, which takes context into account when selecting components from the asset base and which also generates some types of adapter or bridge code.

One approach implemented in this research was to adapt Deline's flexible packaging approach [Deline 2001] to a product line context, as opposed to a general package mismatch. The context differences introduced by product line variation points can be considered as being a type of packaging problem. SPL provides a context where most types of potential mismatch have already been avoided. For example, a product will not have to handle multiple types of error handling because that product line has chosen and fixed a particular approach to error handling. As a result, the only remaining packaging issues are those related to feature selection.

Another implementation approach uses generative programming methods. "Generative programming is about manufacturing a software product out of components in an automated way" [Czarnecki and Eisenecker 2000]. One of the motivations in developing generative programming is a concern about library size and problem context fit between a pre-existing component and the precise needs of a particular program [Batory et al. 1993]. The assumption here is that storing all of the possibly useful variations of a component will not scale up well; therefore, particular components are generated when needed. The focus is on combining fragments of a component into a customized component.

Libraries supporting generative approaches are referred to as active libraries. There are a number of approaches to active libraries. They may extend a compiler or provide a domain specific language. They may also provide a meta-programming capability in which a high level decision may generate code to tailor the library. In this last sense the libraries in this research are active libraries, in that they use information about the features selected to generate code that tailors the library to the related variation points. This approach differs from the typical active library in that the bulk of the assets are hand coded; only the code needed to adapt the asset for feature choice is generated.

This approach is practical because the product line's scope limits the assets needed. Typical generative

approaches are able to generate thousands of variants for a component [Czarnecki and Eisenecker 2000]. There is, of course, no free lunch. There is a certain cost and complexity to this ability. With the product line approach we know prior to implementation which variants will be needed. In a typical case it will be only a fraction of those that could be generated. As a result, the variants actually needed are few enough to make conventional approaches economical.

This work falls outside of the mainstream of generative program work, in that it relies mostly on hand coded assets; however, it does make use of some active library techniques. For example, active libraries have the idea of providing a separate language to describe how components should be configured for a particular use, the Implementation Components Configuration Language, or ICCL. The representation used to specify feature selection in these libraries can be considered a specialized ICCL.

3.3 Implementing Variability in Product Lines

The vast majority of the work of creating a product line lies outside of implementing the software. Much of the effort needed for a successful product line lies in areas such as organizational management. An indication of this can be seen in Clements and Northrop's book [Clements and Northrop 2002] which devotes 5 of its 29 practice areas to asset implementation. Nevertheless, to ship a product it must be implemented.

Much of the effort of implementing a product line is the same as implementing any other software. The unique aspect of implementing product line software is handling the variability introduced by feature selection. This is typically described in terms of a feature type (optional, selection, etc.) for a variation point, and its associated binding time that is specified in the architecture. Thus, the first step in implementing product variability is to understand which techniques are applicable to different feature type / binding time combinations. The basic survey on implementing product line assets is Anastasopoulos [Anastasopoulos and Gacek 2001], which provides a grid of 11 techniques for 5 feature types and 4 binding times. This, of course, just scratches the surface. As the survey authors point out, implementing variability has received little attention. This is supplemented with a more recent survey by Svahnberg [Svahnberg et al. 2005].

My research uses both white box techniques, such as inheritance, and black box techniques, such as components, to implement modules. Work in this area includes Wijnstra [Wijnstra 2000], Griss [Griss 2000], Keepence [Keepence and Mannion 1999], and van Ommering [van Ommering et al. 2000].

Of particular relevance to this work is Brown et al.[Brown et al. 2002]. His approach expands on the idea of a "skeleton function", where a function provides a framework, but key pieces of functionality are passed

as function pointers, allowing variation. A classic example is a sorting routine that implements a particular sorting algorithm but has a comparison function passed in to help decouple it from the particular data types being sorted. Brown provides a more OO version, in which functions are passed into the constructor for an object and held in the object's class variable for use. Brown's paper discusses a C++ implementation, but notes work in progress to apply the techniques to Java. For this technique to work, it must be possible to separate the method signature from its implementation. The most obvious approach to this in Java is to define an interface for use where the signature is required. While Brown discusses his technique in a product line context, he does not specifically describe how to handle the different feature types.

Van Deursen and Klint [van Deursen and Klint 2001] provide an informal algorithm to move from a feature diagram to a class diagram that decomposes the features into classes that can be implemented using conventional development approaches. One limitation in their paper is that they ignore binding times. More importantly, they ignore the problem of feature interaction. In fact, van Gurp, et al. [van Gurp et al. 2001] dismisses the approach because they do not address feature interaction. Hopefully their approach can be extended, possibly by parameterizing the tangled variation using Brown's approach.

Lee and Kang [Lee and Kang 2004] in their work on feature interaction provide a pattern for implementing the control logic necessary to disable features appropriately.

While this is a diverse collection of works, they all have several shortcomings in common. The papers leave it to the programmer's intuition as to how the approach maps back to the feature level of the design. Specifically, they ignore the differences in implementing multi-value selection features instead of single-value selection features. They ignore the issue of making a feature optional. Finally, other than Lee and Kang, they avoid discussing the problem of feature interaction, again leaving it to the programmer's intuition to solve the problem.

3.4 Organizing Asset Bases

The only recent reference found in the literature referring to the organization of asset bases is one sentence mentioning that the asset base at Thales Nederland B.V. includes a "hierarchical component repository" organized by "functional component" [Deelstra et al. 2005]. Deelstra et al. also recommend that variation points be organized hierarchically, but do not tie the variation points to the asset base or the modular decomposition of the product. Given the size of the asset bases reported, it seems reasonable to assume that the development organizations involved have organized the assets based on conventional development practice, which fails to

take advantage of product line features.

Some early SPL methodologies, such as ODM [Simos 1995], and its descendants, such as DAGAR [Klingler and Solderitsch 1996], recommend using architecture to organize assets. This approach specifies the development of a parameterized "domain architecture". The next step in the development process after designing the domain architecture is "architecting the asset base", which organizes the asset base according to the domain architecture.

3.4.1 Alternatives to Organizing the Asset Base

Some advanced configuration management systems that are able to provide code snapshots based on feature variation as well as chronology could significantly reduce the number of components that need to be considered for a selection. This is helpful even if the CM system can only support selection on some of the criteria related to product variants. As CM does not provide organization guidance, our work can be used to complement the CM system. The need to tag artifacts for retrieval on attributes other than timestamps is applicable to a number of areas [Lie et al. 1989]. A CM system that allows code snapshots to be pulled base on some product feature characteristics is provided by the GEARS product [Krueger 2001].

Asikainen et al. [Asikainen et al. 2004] discusses automating component selection artificial intelligence techniques, thus freeing the developer from considering which assets are needed to derive a product. Alternatively, the developer could be given detailed guidance on which components to use. An approach based on a detailed production plan providing guidance down to the level of component names is provided by Lee et al.[Lee and Kang 2004].

An alternative approach to finding a desired asset is to provide a search capability. If we know exact asset names, then including them into a product becomes trivial regardless of the organization. Therefore, a useful search capability needs to be able to find the component of interest based on attributes other than asset name. Finding components in software libraries provides a similar problem. Searching for assets in software libraries is a well-established research topic, although it is a field that has not had much practical success. A survey of approaches to searching a software library can be found in [Mili et al. 1998]. A search approach that leverages the additional information available in an SPL could allow the development of more successful search techniques.

Chapter 4

Research Strategy

4.1 Motivation

The topic of managing an asset base is essentially untouched in the literature. This suggests that it is either a great opportunity for a contribution or there is no need for work in the area. To justify that there is a need for work in the area, I have taken a two-pronged approach: first, show that the size and complexity of an asset base are such that it exceeds any reasonable threshold of ability for users to manage assets unassisted; and second, present case studies from the literature that identify problems in using SPL in which the problems may be reasonably attributed to asset base size and complexity.

The problems of measuring an asset base are discussed in Terminology - Measuring the Asset Base. Table 4.1 presents some examples that illustrate asset base size. While the size information in Table 4.1is neither as exact nor comprehensive as desired, it is sufficient to show the scale of existing asset bases and motivate the need to manage them.

Deelstra et al. [Deelstra et al. 2004][Deelstra et al. 2005] examine three product lines in two different organizations. Their focus is on the difficulties of deriving products from the asset base. They identify eighteen problems and issues in trying to derive products; of these, 11 are related to the asset base size and complexity issues that this research addresses. However, they do not relate these problems back to the issue of managing the asset base. Here is a partial list of their problems that relate to asset base management:

- “Over-explicit documentation decreases traceability of relevant information” - too much documentation, an example of where assets should be removed.
- “Unmanageable number of variation points and variants”
- “Variation points not organized hierarchically”
- “Limited resources for realizing variability” - the problem here is that asset developers are left to handle implementing variability on their own.
- “No uniform treatment of variation points over the life cycle” - the problem here is that asset developers are left to handle binding times on their own.

Company	Product Line	Domain	Language	Variation Pt	kLOC
Axis[1]		Print Server	C++		200
Phillips[2]	Medical	MRI	C++		3,000
Northern Telecom[3]	DLC	Phone Switch	C		4,000
Dacolian[4]	Intradata	Image recognition		6,872	
Thales[5]	TACTICOS	Navel 3C			1,500
Enginio[6]		Disk Storage			800
Bosch GS[7]	Engine ctl	Automotive		7,200[8]	690

[1] [Bosch 1999]
[2] [Jaring et al. 2004]
[3] [Dikel et al. 1997]
[4] [Sinnema et al. 2004]
[5] [Deelstra et al. 2005]
[6] [Lever 2005]
[7] [Steger et al. 2004]
[8] "Calibration Parameters" precise definition not supplied. Also, 5200 "Features", definition not supplied.

Table 4.1: Examples of asset base sizes

- "Obsolete variation points not removed"
- "Different provided and required interfaces complicate component selection" - problem is multiple assets with the same functionality due to packaging issues.
- "Variation point and mechanism considered identical" - the implementation "hard codes" particular choices for variation point characteristics. Changing these choices requires modifying the code implementing the variation point.

Combining these problems found during actual experience with product lines together with the size information shows that providing better methods to manage the asset base should make it easier to produce products from the asset base.

This research addresses two strategies that might be used to manage collections:

1. minimize the number of assets that need to be stored
2. organize the assets

Both of these strategies could be applied to many different situations addressed by software engineering. Unrelated to SPL, Biggerstaff makes an argument that the increase in the number of assets, as features are added to library components, must be minimized to avoid maintenance costs becoming prohibitive [Bigger-Staff 1994]. This was one motivation behind much of generative programming / active library research. Even more generally, it is common practice in any large software project to group and organize files hierarchically

using a directory tree. There are many existing ways to address the issues of having a large number of artifacts for a software project that can also be used with an asset base.

There may be many possible ways to take advantage of SPL characteristics in managing assets. Creating one novel method for each strategy should prove that SPL characteristics can be leveraged to provide better asset management. For the first strategy, minimize assets, a method is provided to remove duplicative assets that uses information about the variation point. For the second approach, organize assets, a method is provided that uses product feature information to hierarchically group the assets.

While these methods should be useful in a large range of product lines, some assumptions about the nature of the assets and the asset base are unavoidable. One assumption is that the implementation assets dominate the size of the asset base. It assumes a relationship between features and the modular decomposition of implementation assets. This is certainly not a 1 to 1 mapping. Most features will require multiple modules for their implementation. There may be some cases of cross-cutting features, and also of feature dependency and tangling, that will affect implementing modules; however, a typical module is likely to be involved in the implementation of only a few of the product's features.

For the many SPLs applying the methods developed in this research should result in a number of advantages, including:

- Fewer assets providing the same amount of functionality.
- Assets organized in a logical, intuitive way, that expands gracefully with the product line.
- Standard ways to implement many aspects of variation points including binding times, and some types of feature interactions, which should improve maintainability.

4.2 Research Method

I begin the research process by selecting a research method. Software engineering lacks a dominant research method or a commonly followed process for selecting a research method. In the absence of a software engineering research methodology, I turn to other disciplines for guidance. A commonly cited source on the selection and application of applied science research methods is Yin [Yin 1989]. Following Yin's methodology, the first step is to choose a research method that fits the problem from the possibilities listed in Table 4.2.

The survey, archival analysis, and history would all require more access to corporate data than is available. This leaves experiments and case studies, which have many similar characteristics. How should one choose?

Strategy	Form of Research Question	Requires Control Over Behavioral Events?	Focuses on Contemporary Events?
Experiment	How, why	Yes	Yes
Survey	Who, what, where, how many, how much	No	Yes
Archival analysis	Who, what, where, how many, how much	No	Yes/no
History	How, why	No	No
Case study	How, why	No	No

Table 4.2: Relevant Situations for Different Research Strategies

According to Yin, the key difference is that a case study looks at a phenomenon within its context, while an experiment deliberately separates it from the context to gain more control over the variables being studied. For this research, the context provided by the product line is important to understanding the asset base. For example, we are not looking for organization techniques for collection in general, but rather for organizing an asset base to support a product line. Therefore, the case study method is the most appropriate choice. Specifically: "A case study is an empirical inquiry that: investigates a contemporary phenomenon within its real-life context; when the boundaries between phenomenon and context are not clearly evident; and in which multiple sources of evidence are used." [Yin 1989] Having selected the case study methodology the next task is to develop a set of research questions.

4.3 Research Questions

The goal of this research is to define and validate techniques that improve the management of the core asset base of a software product line; specifically, to define new techniques that leverage unique characteristics of SPL, to make asset base easier to use.

As explained in Section 4.1 an important class of difficulties in using the asset base is related to size. The underlying problem is that the size of the asset base makes it more difficult to select the asset needed for a particular operation. This difficulty occurs when working with implementation assets. That the vast majority of assets, by both size and quantity, are implementation assets suggests that size is an important part of the difficulty of choosing the asset. The difficulty of choosing assets extends to the activity of deriving a product from the asset base. For many product lines, derivation is a frequent activity, possibly occurring thousands of times in a year. As a result, extra cost to derive a product is an important concern.

Based on these observations the question I developed was: "How can the number of assets that need to

be examined, particularly while deriving a product, be reduced?" Note that the number of assets that need to be examined to make a selection can be reduced in several ways:

- Organize assets in such a way that groups of asset can be ignored
- Organize assets in such a way that only particular groups need be examined
- Reduce the total number of assets
- Reduce the number of similar assets

I approach this problem through two strategies to managing collections:

1. minimize the number of similar assets that need to be stored
2. organize the assets

While there are many methods that could be applied to these strategies, one method for each strategy will be developed in this research.

Minimize the Number of Assets

Based on the literature [Deelstra et al. 2004][Deelstra et al. 2005], a common problem is that of an asset base containing multiple assets that provide the same underlying functionality, but fit into the product in varying ways. This makes the selection of an asset particularly difficult because it is no longer sufficient to understand what the asset does but also requires understanding the context that each variant of the asset fits. This will be referred to as the packaging problem.

Thus a good candidate for reducing assets and simplifying selections is to address the packaging problem by storing only one asset for the intended functionality in the asset base. This method for minimizing assets will be referred to as "removing duplicative assets". While not the only method possible for pursuing the strategy of minimize assets, it has been reported as a real problem, and it complements the organize assets strategy. Hence, my research related to minimizing assets will focus on removing duplicative assets. For the removing duplicative assets method the following research questions will be addressed:

1. To what degree can component functionality be isolated from its context, via packaging?
2. Can feature interaction be controlled by packaging techniques?
3. To what extent can the approach to packaging be standardized across multiple components?

A case study will be made, using the Pedagogical Product Line as a base, to study the possibility of removing duplicative assets by separating the asset's functionality from the packaging needed to use the asset at a variation point. The assets will fit the variation point along three different dimensions - cardinality, optionality, and feature interaction. Four different techniques will be applied to accommodate the asset to the variation point - monolithic, separate packaging specific to the asset, modular packaging, and generative packaging. Assessments will be made as to the feasibility and suitability of separating the packaging from the asset functionality for different types of variation points. Also, assessments will be made of the different techniques for fit the asset to the variation point.

Expected outcomes will be measured in terms of:

- Difference in asset base before and after application of a technique
- Similarity of resulting assets
- Whether feature interactions can be confined to packaging
- Range of packaging techniques need to handle all components across all variation points

Organize Assets

Whenever we have too many items in a group to understand, the classic approach is to break the group into smaller groups. The key to making this effort profitable is to group the items in a natural way based upon their use. As Plato advised, divide "according to the natural formation, where the joint is, not breaking any part as a bad carver might." [Plato 1999]. By organizing the asset base in such a way that similar assets are located with each other and apart from dissimilar assets we allow the user attempting to find an asset without having to examine all assets.

For the organize assets strategy the following research questions will be addressed:

1. Which approach to organizing the asset base is most effective?
2. What is the effect of feature interactions and dependencies on the organization of the asset base?

A case study will be made, using the Pedagogical Product Line as a base, to study the effect of organizing the asset base. A comparison will be made of the results of organizing the asset base using three different approaches:

1. by architecture

2. by feature

3. by key domain concept (KDA)

Typically, assets for a single product are organized by architecture, while libraries are organized by key domain concepts. SPLs fall somewhere in between products and libraries. It is therefore reasonable to consider whether the existing approaches for either products or libraries will work for SPL. In addition, organizing the asset base by product features will be considered, this is a novel approach introduced by this research.

Expected outcomes will be measured in terms of:

- Natural division
- Easy to map
- Reasonably sized groups
- Similarly sized groups

4.4 Asset Reduction vs. Asset Organization

The overall goal is to make the asset base more usable. The two approaches are obviously related. If the number of assets is reduced until there are only a few remaining, then organizing them becomes superfluous. On the other hand, if the assets are organized well enough that the asset that needed easily located, then the user will be fairly indifferent to the number of assets. It would appear that one approach can be traded for the other and that fully developing a single approach would achieve the goal of providing a more usable asset base.

To see these approaches as replacements for each other is not realistic. It is not possible in the real world to reduce the number of assets in a large product line to a point where organization is not needed. The assets will have to be organized; therefore, as much benefit as possible should be derived from the organization. While organizing assets helps, some subsets of assets will not respond to organization in a way that makes selection easy. Duplicative assets, which vary only in their packaging, are particularly difficult for a user to select from, because all of them do what the user wants. Instead, the user must not only understand what the asset does, but also the context in which the asset will be used. However, it is hard to have an organization scheme that will help this situation. Both of the organization schemes identified, architecture-based and feature-based,

will group all of these assets together, leaving the user's selection problem unmitigated. It is only through eliminating the duplicative assets that the user's selection problem is made easier in this situation.

4.5 Summary

The research outlined should result in making the asset base easier to use. Two strategies to managing the asset base are identified: minimize the number of assets that need to be stored by removing duplicative assets and organize the assets to make them easier to find during the product derivation process. A technique that takes advantage of SPL characteristics is proposed for each approach.

Removing duplicative assets is pursued by separating the asset's packaging from its intended functionality. Several techniques are then applied to the packaging development. A case study will compare the techniques and provide guidance on the most effective use.

A separate case study will look at organizing the asset base by comparing an approach currently in use, organizing by architecture, with a novel approach being proposed in this research, organizing the asset base using product features. Taken together, this work shows ways that SPL asset bases can managed so that they are easier and less costly to use.

Chapter 5

Minimize Assets

5.1 Introduction

Product lines have an additional type of variation beyond that found in other types of software, the variation between products in the same product line, which will be referred to as product line variability. This variability between products manifests itself in all phases of the software life cycle, from the initial business case and requirements on through to the actual products.

Discussion of how to implement variation has been limited. Software Product Line (SPL) literature on product line variation has focused on the early (prior to implementation) phases of software development. “Literature has already addressed how to create and instantiate generic product line assets, such as domain models and architectures to generate instance specific ones, yet little attention has been given on how to actually deal with this genericity at the code level. ” [Anastasopoulos 2004] Other research noting this lack of implementation advice include: [Brown et al. 2002] [Muthig and Patzke 2002] [Anastasopoulos 2004] [Muthig and Atkinson 2002].

This lack of advice is not due to an absence of need. “Most software systems are inflexible. Reconfiguring a system’s modules to add or to delete a feature requires substantial effort. This inflexibility increases the costs of building variants of a system, amongst other problems.” [Murphy et al. 2001]

Implementation has typically been discussed in terms of a list of general techniques, with a focus on the binding time related to the technique. Examples are Anastasopoulos and Gacek [Anastasopoulos and Gacek 2001] and Svahnberg et al. [Svahnberg et al. 2005]. This approach is more helpful to an architect trying to choose a development environment than a developer trying to live in one. The techniques are discussed in a language independent manner, which makes the advice more general, but at the cost of missing details important to the developer. The semantics of inheritance, for example, varies widely between languages. Finally, the issues involved with optional features and the effect of feature interaction are not addressed.

Other available research looks at a particular technique, such as skeleton classes [Brown et al. 2002], aspects Griss200, inheritance [Keepence and Mannion 1999], and other techniques. These papers typically ignore those types of product line variation which the studied technique has difficulty handling.

My work examines product line related variability at the level of variation points. I have identified three different characteristics of product line variability - cardinality, optionality, and feature interactions. In this section I consider implementations for each of the varieties of variability and discuss how the characteristics of variability interact when providing an implementation.

The goal is to provide specific advice to core asset developers on how to build core assets that are able to provide any of the permitted product line variations. To achieve the level of specificity desired and still have a manageable project, the development environment studied is limited to Java. The development environment is enhanced two different ways:

- AspectJ - an aspect-oriented extension to Java
- XVCL - a general purpose generative engine, which can be used with Java

I consider a variety of implementation techniques that are available within the Java environment. Java is a mainstream OO language, with many similarities to other OO languages, particularly C++. Hence, many of the techniques presented may be applicable to other development environments. However, this is not pursed in the current research. Implementation techniques are provided for variation points such that the product developer does not have to modify assets or write implementing code.

5.2 Implementation Mechanism and Binding Time

Most implementation mechanisms support only one binding time; hence, the discussion of binding times tends to be intertwined with that of implementation mechanisms. This interconnection creates a problem, because changing a binding time in ways not planned for in the design may force the related assets to be re-implemented. However, providing multiple binding times in the design will likely require multiple implementation mechanisms to be used, complicating the product implementation. Also, it will usually cause the latest binding allowed in the architecture to be implemented, even if it is known by the end of the implementation process that the selections for a particular product do not require that late a binding time. Finally, it has been noted that, for many product lines the implementation used is due to familiarity with a particular mechanism, rather than the characteristics of a particular variation point. Further discussion of these issues can be found in [Deelstra et al. 2005][Bosch et al. 2001]. Van der Hoek [van der Hoek 2004] discusses the need to decouple binding time choice from asset implementation under the heading of any-time variability.

Since asset implementation is the object of study, the binding times considered can be limited to:

- Construction - which will be divided into sub-phases when implementation techniques are considered.
- Runtime - which will include all of the variabilities that are still unresolved at the end of construction. Whatever is necessary to resolve variability at runtime must be built at construction time.

The construction phase includes any techniques that result in the production of code, it may be divided into several phases. The coding phase involves the developer writing code. Processing phases involve applying tools to the code to produce a program that can be run.

Even in Java there are many possible coding techniques that could be applied. The techniques considered for the coding sub-phase are:

- Inheritance
- Java language interfaces
- Parameterization

"The Java programming language provides two mechanisms for defining a type that permits multiple implementations: interface and abstract classes." [Bloch 2001] That is both of the Java mechanisms provide methods that are dispatched polymorphically based on which implementation was chosen when the related object was instantiated. While Bloch limits this use to abstract classes, any class in Java that is not declared as `final` may be sub-classed. Use of an `interface` could be consider a type of inheritance; however, I chose to reserve that word to refer to Java classes that have an `extend` relationship. Class inheritance in Java is limited to single inheritance. One of the motivations to adding the `interface` construct to Java was to provide a substitute for multiple inheritance. Another distinction is that inheritance exposes implementation, making it a "white-box" technique, while use of an interface does not expose implementation, making it a "black-box" technique. The term *interface* has a more general meaning in software engineering than is specific to the Java language construct `interface`. I attempt to make clear when the Java language construct is being referred to by referring to it as "Java language `interface`" with the word `interface` in `code font`.

Processing can be divided into pre-processing, compilation, and post-compilation. Pre-processing involves the substitution of text strings in the source code prior to compilation. Since Java, unlike some other languages, does not include a pre-processing facility in the language, the XVCL frame processor will be used [Jarzabek et al. 2003][1] to explore some possibilities available using pre-processing. Post-compilation allows

[1] It should be noted that XVCL is not limited to working with Java and that it has capabilities that extend beyond most pre-processors. No attempt is made to fully explore XVCL, rather it is used to examine how a pre-processor might be profitably employed in implementing variation points when using Java as the implementation language.

program behavior to be modified after compilation and without modification to the source code whose behavior is being modified. Unlike pre-processing, the directives to control post-compilation are not included in the source code. To explore post-compilation AspectJ [Kiczales et al. 2001], an aspect-oriented extension to Java will be used. While aspects are a post-compilation technique, the exact binding time varies based upon the implementation. The version of AspectJ used allows the aspects to be included or omitted at class load time, similar to link time in many languages.

The three coding techniques, plus a pre-processing technique (using XVCL), and a post-processing technique (using AspectJ), results in five implementation techniques that will be considered.

5.3 Specifying dimensions of product line variability

Products in a product line differ from each other only at variation points and along the proscribed dimensions of those variation points. This allows us to assume that the rest of the product remains the same from product to product throughout the product line. At the variation points, only those types of variation that have been identified as product line variation should affect the composition of the asset with the product.

Dimensions of variation that are traditionally considered at the variation point in the design phase are cardinality, and optionality. In addition to these, feature dependency can be profitably handled in the same variation point implementation as well. These design decisions must then be implemented using some mechanism within the constraints specified by the requirements for binding times.

5.3.1 Cardinality

Cardinality is the property that specifies the number of variant value selections that can be made. A variety of notations to express cardinality have been used in SPL work. Czarnecki et al.'s [Czarnecki et al. 2005] model for variation point cardinality is used in this paper. This approach, referred to as *cardinality-based feature modeling*, specifies the number of selections allowed at a variation point using a series of natural number ranges plus the Kleene star operator *. For example, the ranges [1..3][5..*] would require at least one selection, would not permit four simultaneous selections, but would allow any other number of selections. Some more common range selections include:

0..1 , which has the meaning of optional in most notations.

1..1 , which has the meaning of required in most notations.

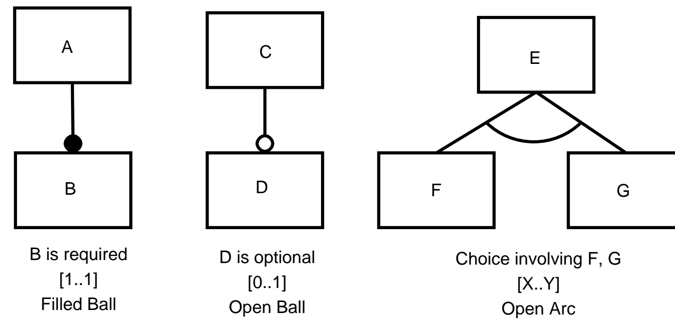

Figure 5.1: Cardinality Notation

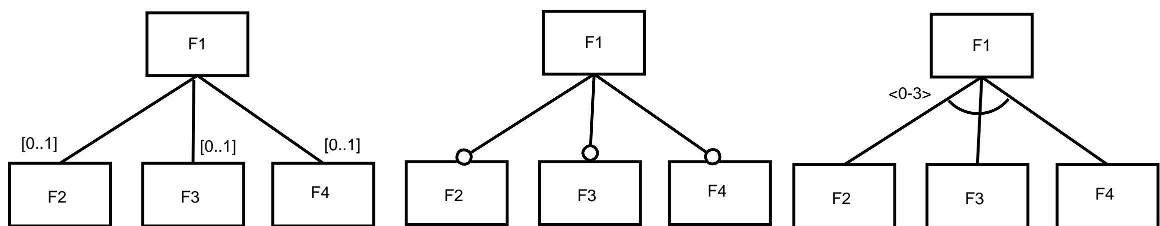

Figure 5.2: Possible Ways to Express Cardinality

0..* , which means any number including none.

In addition, the filled-ball/open-ball notation for mandatory and optional selections continues to be supported, as does the open (but not filled) arc notation to show grouped features, shown in Figure 5.1. The ball and arc notations are considered to be more readable and thus recommended for common cases. In an improvement on earlier work, the feature selections that this syntax specifies are unambiguous in the semantics that they express. However, there is still more than one way to use the notation to express the same meaning in some cases. For example, the three diagrams in Figure 5.2 all have the same meaning. One goal is to incorporate the ability to implement the cardinality typically needed in designing a SPL. The cardinality possibilities provided by *cardinality-based feature modeling* are sufficient to reach this goal. In planning these implementations, several simplifying assumptions are made. First, variation that is resolved prior to the implementation phase of development can be ignored. Second, the cardinality in the implementation code is not verified. Given a cardinality of [1..2][4..5], where a cardinality of 3 is not allowed an implementation can handle at least 5 selections is considered acceptable. Code is not included in the product to insure that a combination of features with a cardinality of 3 has not been specified.

An implementation that supports multiple values when only one selection has been made has two different kinds of costs. One is program efficiency at runtime. Another is cognitive complexity for the programmer.

Table 5.1: Effects of Cardinality on Implementation

Included	Feature is included in this product.
Selection - Single	One selection out of a set is included in the product
Selection - Multiple	More than one selection out of a set is included in the product.

Finding a program construct, such as a collection, when none is needed may cause confusion and at a minimum increases the complexity of code that needs to be understood. In contrast, a program construct that simply provides more space than is actually required, such as a growable collection may have additional run time costs, but not cognitive costs.

Open variability, where a selection between options as not yet been made, has a variety of costs and risks [Hunt and McGregor 2006]. Therefore, to whatever extent variability has been resolved by the end of the construction phase, that information is used to minimize these costs and risks by closing any unused variability. For example, if at the end of the construction phase an optional feature has been omitted, then the product code should have no trace of that feature. Similarly, for a variation point that allows the inclusion of multiple features, if by the end of the construction phase for a particular product only one feature has been chosen, only to the chosen feature should be referred to, avoiding the overhead needed for multiple features.

Begin with two possibilities: the feature is either omitted or included in the product. If the feature was included in the product, Table 5.1 shows the possible cases. Table 5.1 considers only variability that remains open in the construction phase.

5.3.2 Optionality

If an optional feature is omitted from a product, the code related to this feature should be cleanly and completely omitted from the product. For many techniques, recommended in the literature to implement variants, this is not possible. For example, if inheritance is used to implement variants the code can not be completely omitted. Either a sub-class with empty methods is created or no object is created at all. If no object is created, runtime code to check for a null reference will be required. This problem is generally shared by component approaches [Lee and Kang 2004].

5.3.3 Feature Interactions

Feature interaction involves two features call them - F1, F2. Assume F1 is a feature in the product. If F2 is added to the product, then the behavior of the system may be different than it would have been with only F1. For example, in my research, which uses arcade style games for its domain, I have a feature called "practice mode". If practice mode is included in the product then so must the feature of being able to turn off score keeping.

Feature interactions have not generally been discussed as part of variation point implementation; however, they constitute one of the ways that products vary as features are selected for a product variant. If we accept that the variation point is the place in the product where we see the consequences of feature choices then feature interaction should be part of variation point design and implementation.

Feature interaction has been understood in a variety ways in different fields of software engineering, primarily in telecommunications [Mehta and Heineman 2002]. My approach to feature interactions is based on work by Lee and Kang [Lee and Kang 2004], and takes advantage of a design pattern that they present. I extend their work several ways: I am working at the variation point / implementation level rather than a feature / design level. As a result I am working with the problems of handling feature variants and optional features while handling feature interactions. Finally, Lee and Kang do not discuss how aspects and frames can supplement components, which I include in my discussion.

The code related to handling feature interaction should be left out of the product if the particular set of features chosen does not interact with each other. It is preferable to keep the code handling the interaction separate from code implementing the feature. A given feature may be affected by more than one other feature, and thus take part in multiple feature interactions. If the code to handle the interactions has been kept separate, a new version of each feature will be needed for each possible combination of feature interactions.

Figure 5.3 summarizes the concerns to be resolved at the variation point - cardinality, optionality, and feature interaction. Each of these is affected by its own binding time and requires a choice of implementation mechanisms. For example, a decision to include a feature in a product may be bound earlier in the process than the value which will be selected for that feature. Similarly, in considering feature interaction, one of the participating features is typically selected first and decision related to its cardinality may be made before the other features related to the interaction are selected. With some techniques it may be appropriate to select and configure all of the features independently before managing the feature interactions later in development process. I call this model of product line variability the *Variability Characteristic Space* (VCS).

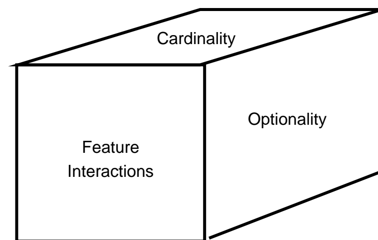

Figure 5.3: Concerns at the Variation Point that define the Variability Characteristic Space

Implementation issues in handling these VCS concerns for each of five implementation mechanisms will be examined.

Components that are included in a product at a variability point are affected by the product line variability. There are three major approaches for such components:

- Implement components as core assets that contain sufficient variability to produce the desired product portfolio.
- Implement components as core assets and modify the assets as needed to produce the desired product portfolio. Modified components may be considered as either core assets or product assets depending upon the process used at the particular organization.
- Provide only a description of the variation point as a core asset [Webber and Gomaa 2004]. Implement components as product specific assets when needed.

The first approach has the most reuse potential and this is the approach which research has focused on; however actual experience suggests the second and third approaches have been widespread. This suggests the possibility that additional ways to implement components as core assets might be desirable and provide increased reuse.

5.4 Approach

The goal is to provide components that are implemented as core assets and contain enough flexibility to produce the desired product portfolio, without modifying the component. The approach chosen is to divide the logical component into two types of parts:

- A part that implements the desired functionality, called a ware.

- Parts that implement the all of the different types of product line variability, called the packaging.

This approach of dividing a component into a ware and packaging is based upon work by Deline [Deline 1999]. Deline was attempting to provide functionality for general reuse; for example, a ware that could be used as both an ActiveX control and a Unix pipeline filter. Deline found that in this general case there are many dimensions of interaction between the component and the product that can result in mismatches. Attempting to resolve the general case led to complex and incomplete packaging solutions. However, the SPL's use of common development processes, architecture, assets, etc. among products leads to a standardized environment for components. The interaction between the component and a particular product should vary at those designated variation points. This, in turn, should greatly simplify the packaging needed. Therefore, the code that implements the product line variants should be placed into the component's packaging. This should leave the ware unmodified for different products. All of the variations specified in the VCS concerns should be provided without product-specific development. Also creating a core asset for each combination of choices that occur at a variation point should be avoided.

The first step is separating the component into the packaging and the ware. At this point each packaging module implements a single point in the VCS. However, we may need a separate packaging module for each combination of cardinality, variation, and interaction. Then the packaging is separated into the different VCS concerns in order to allow a complete packaging component to be assembled from the related pieces. For example, a packaging component could be assembled with or without a particular feature interaction. This should allow a product developer to assemble the needed variability without modifying core assets or creating product assets.

This research compares implementing the three different concerns identified by the VCS (optionality, feature interaction, cardinality) with a number of different implementations. Cardinality is divided into single and multi value cases, because these cases require different implementations.

The results of the experiments are organized several different ways. Section 5.9 organizes the result by implementation technique. Section 5.10 provides a pattern language that is organized by the VCS characteristics that need to be implemented. Section 5.11 discusses the results in the context of the research goal of managing the asset base.

5.5 Criteria for Evaluating the Experiment

After implementing some representative variation points from the Pedagogical Product Line (PPL) described in Section 2.4, The following issues will be discussed:

- Differences in the asset base before and after the application of a technique. Does this increase or decrease the size and complexity of the asset base? If so, is it harder to derive products?
- Problems regarding feature interactions. Can the difficulties raised by feature interactions to be confined to the packaging?
- Range of packaging techniques needed to handle all components across all variation points. Are there cases where a technique could not be coaxed into handling a particular type of variation?

5.6 Description of Experiments

To investigate these issues involving the implementation of variation points in concrete ways the code base for the PPL was modified using the various techniques discussed. This work was carried out at two variation points: scoreboard, selected as an example of a single value variation point, and services, selected as an example of a multi value variation point. Examples of four VCS concerns, optionality, feature interaction, single value cardinality and multi value cardinality were implemented. Implementations were done using coding approaches (inheritance, Java language `interface`, and parameterization), XVCL frame processing, and aspects. This section will provide the reader with the necessary background to understand the experiments by describing the two variation points and the implementation issues that they engender. The following section will discuss the work carried out organized by the VCS concerns.

In the initial version of the PPL, variation was implemented using parameterization, all of the variable portions of the product are instantiated in the respective product file and then passed into the various core components. As a result, all changes to a game product can be made by editing a single file. There is little else in the product file, which should make it easy to generate by, say, a wizard rather than by editing it manually. If a scoreboard is selected, a scoreboard variant is instantiated in the product file and passed as a parameter into the view component.

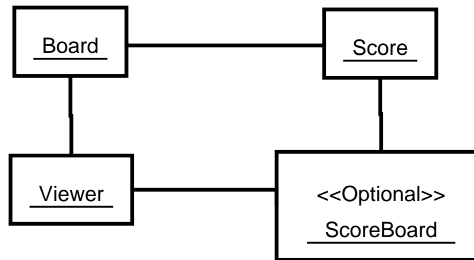

Figure 5.4: Scoreboard Object Diagram Base Case

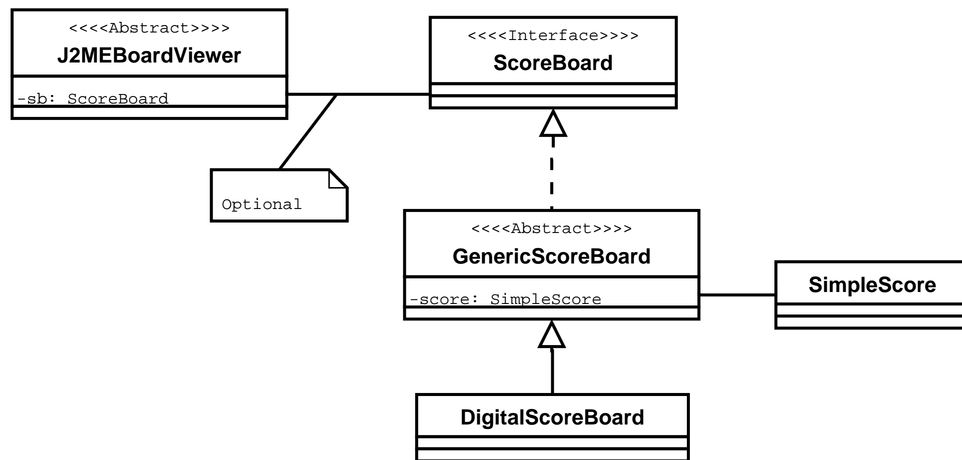

Figure 5.5: Class diagram of initial scoreboard implementation

5.6.1 Scoreboard Variation Point

For the first part of the experiment an optional PPL variation point that implements single value cardinality was selected, specifically the scoreboard. As noted the PPL uses a model-view-controller architecture. The scoreboard is a view of the current score. View-related code is platform dependent, the platform used for the implementation discussed is Java 2 Micro Edition (J2ME). A reference to the score object is passed into the scoreboard object when it is instantiated at runtime. Having a scoreboard is an optional feature. The view component contains a reference to the scoreboard component. In its paint method, the scoreboard reference is checked, and if the reference is not null, the paint method of the scoreboard component is called. The object diagram for this base case is show in Figure 5.4. A class diagram using Digital scoreboard as an example of a particular type of scoreboard is shown in Figure 5.5.

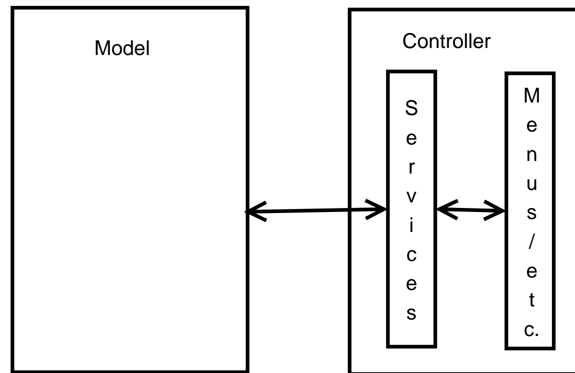

Figure 5.6: Role of services in MVC architecture

5.6.2 Services Variation Point

Services was selected to provide an example of a variation point with multiple value cardinality. Services in the PPL implement commands that are used to effect the game environment outside of the actual game play. Examples of services are starting a new game, setting the game speed, and saving a game score. Services separate the implementation of the command from the way the command is invoked. The most common way to invoke a service is through a menu selection. Separating the implementation of the service from the menu allows the same implementation code to be accessed from more than one menu, separates it from the look and feel of the menu, and allows the service functionality to be accessed through mechanisms other than the menu. From an architectural viewpoint, the services provide a controller interface that is separated from the presentation of the controller provided by menus, etc. Figure 5.6 shows the role of services within the MVC architecture of the PPL product line. Services provide an example of a feature where multiple values may be chosen for inclusion into the product simultaneously. However, not all services may be selected for every products. It makes little sense to control speed between frames in bowling. Some types of devices may not support sound, and thus do not need to control the volume. Thus, each product variant may choose a different combination of features. Some services will always be required; therefore Services is a mandatory, rather than optional, variation point.

In general, omitting a multiple value variation point can be implemented in the same way as omitting a single value feature. Also, feature interaction does not differ from the single value case already discussed. A situation which does not come up for a single value feature that does occur for multiple value features is that of optimizing the implementation when only one selection has been selected. While this does not come up in the services experiment, it will be discussed under implementations.

The initial version of the PPL product line provides a single Java language `interface`, Services, and a single implementing class, J2MEServices, for all of the services. The `interface` is used to isolate the client code from platform dependencies. Examples of the methods included in Services are:

- `void play()`
- `Score[] topScores()`
- `void setSpeed(int speed)`

This approach, which will be referred to as the monolithic approach, was used in the initial version of the product; however, does not provide the modularity needed to omit unneeded services. It does demonstrate that the natural implementation of the services involves a variety of signatures, which is problematic for some approaches. How to provide a modular version of services using these different implementation mechanisms will be examined.

5.7 Implementing VCS characteristics

5.7.1 Implementation of Feature Optionality

The key to implementing optionality, is the ability to completely omit a feature. The issues with optionality are the same for single and multi value variation points. Therefore, the work on optionality was implemented only at the scoreboard variation point. If the scoreboard has been omitted, every part of it should be eliminated, including the existing `if` statement that makes a runtime check to see if the reference is null. This is not possible in the case of standard Java, but it can be implemented using AspectJ and XVCL. It would seem possible to avoid this problem by having a more general mechanism for sub-displays. Sub-displays, including the scoreboard, could register themselves with a mechanism that would loop through and invoke a method on all of the registered sub-displays. This does not really solve the problem of omitted components. If the number of sub-displays registered is one, there should be only the reference instead of a loop through a collection, and if it is zero there should be no reference at all.

When the scoreboard is omitted, null is passed as a parameter, which the view component must test for, see Listing 5.1. This inability to completely omit the component using parameterization is shared by inheritance and interface implementations. An alternative to passing a null reference is to use a method with an empty body. This allows the call to always be made and produces no effect when the feature is omitted.

```
public class J2MEBoardViewer extends Canvas implements BoardViewer,
            BoardController {
      . . .
      private ScoreBoard sb;
      . . .
      public void setScoreBoard(GenericScoreBoard sb){
            this.sb = sb;
      }

      protected void paint(Graphics g) {
            . . .
            if(sb != null)
                  sb.paint(g2);
            . . .
      }
      . . .
}
```

Source Listing 5.1: Code to omit scoreboard using Java

All of these approaches have a greater than necessary runtime cost. An aspect may be omitted completely from a build, which would make the feature truly optional. There may be a difficulty in that aspects were not designed to insert code at an arbitrary point in the program. The code could be refactored to provide a weave point, which somewhat defeats the purpose of using an aspect to allow the complete removal of the code when the feature is omitted. List 5.2 shows viewer code that provides a dummy method to allow an aspect to insert the scoreboard code. List 5.3 shows the aspect point cut to add a scoreboard.

Pre-processors, such as XVCL, are able to completely remove code and also able to insert arbitrary code at arbitrary code locations. List 5.4 shows the XVCL frame that generates viewer code that may or may not include scoreboard code. Adapting an empty frame will generate viewer code that has a single space at the location where the frame is being adapted. This in turn will be ignored by the Java compiler. This provides the most complete solution to the problem of omitting an optional feature.

Optional features may be omitted from the product using all four implementation mechanisms - inheritance, Java language `interfaces`, aspects, and XVCL; however, the quality of the solution varied. To support an optional feature, the modular code approach requires a runtime `if` statement. XVCL is able to completely omit the feature, but requires inserting an XVCL include frame directive into the viewer. The insertion point needed for optional services must be specified at the time the code is developed. Aspects are able to completely omit the feature without requiring modification to the viewer code. Aspects provide the best separation between the optional feature and the client code. However, aspects require that the existing

```
public class J2MEBoardViewer extends Canvas implements BoardViewer,
                BoardController {
        . . .
        protected void paint(Graphics g) {
                . . .
                drawScoreBoard(g2);
                . . .
        }

        public void drawScoreBoard(Graphics g){
        // place holder - does not do anything

        }
        . . .
}
```

Source Listing 5.2: Code to omit scoreboard using AspectJ

```
public aspect DigitalScoreBoard {
        void around(J2MEBoardViewer bv, Graphics g) :
                call(void J2MEBoardViewer.drawScoreBoard(Graphics))
                && target(bv)
                && args(g){
                        . . .
        }
}
```

Source Listing 5.3: AspectJ point cut to insert scoreboard code

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE x-frame SYSTEM "default">
<x-frame name="J2MEBoardViewer.xvcl" outfile="J2MEBoardViewer.java" language="java">
        <set var="SCOREBOARDTYPE" value="Digital"/>

public class J2MEBoardViewer extends Canvas implements BoardViewer,
                BoardController {
        . . .
        protected void paint(Graphics g) {
                . . .
                <adapt x-frame="?@SCOREBOARDTYPE?BV.XVCL"/>
                . . .
        }
        . . .
}
</x-frame>
```

Source Listing 5.4: Code to omit scoreboard using XVCL

code provide a suitable hook for the optional feature. Aspects are not able to insert code at arbitrary places. If the aspects use a dynamic implementation, as is the case with the current version of AspectJ used in this research, then there will be a runtime performance penalty[2]. If the need for the optional feature is known when the client is being coded, using an XVCL frame for the optional feature provides a way to cleanly exclude the feature without a runtime penalty.

5.7.2 Implementation of Feature Interactions

If the practice mode feature is included in the product, it will interact with the scoreboard feature. If the product includes practice mode and the user selects practice mode, then the screen should display "Practice Mode" instead of the score. There are additional effects of practice mode, but here the concern is with its effect upon the scoreboard. The code used to implement practice mode should be kept separate from the code used for the scoreboard display variants. Otherwise an additional asset will be needed, one containing the practice mode related code, for each display variant.

Coding Mechanisms

To accomplish this separation when using the inheritance and Java language `interface` approaches a decorator pattern [Gamma et al. 1995] is used. This creates a wrapper class with the existing scoreboard interface and holds one of the existing scoreboard variants as a reference, see Figure 5.7. In practice mode the wrapper puts up the special practice mode message, when not in practice mode calls are passed through via the reference to the held scoreboard variant. By using a decorator approach any number of feature interactions could be accommodated. The addition of the wrapper class(s) is explicit in the code at the point where the scoreboard is instantiated.

Aspect Mechanism

Feature interaction was also implemented using aspects, shown in Figure 5.8. The `around` advice was used on the paint method call of the parent scoreboard object, GenericScoreBoard. This will invoke the aspect regardless of the scoreboard variant used. If practice mode is enabled, then the aspect will put up the practice mode message; otherwise it uses the `proceed` key word, as shown in List 5.5, allowing the normal operation of the scoreboard to continue. An advantage of this approach is that the aspect does not need to be aware of

[2]Earlier versions of AspectJ used a static implementation in which the aspect code was inserted into the source code prior to compilation. This avoids a runtime penalty, but also moves the binding time from program load time to the end of the construction phase.

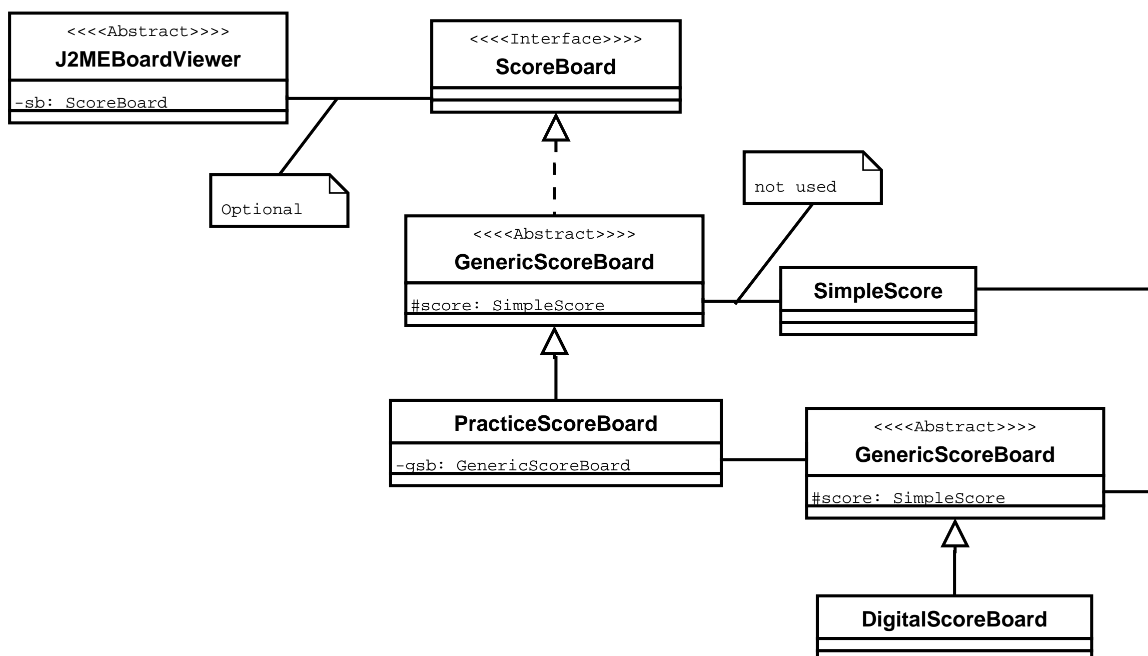

Figure 5.7: Practice Mode implemented using a decorator pattern

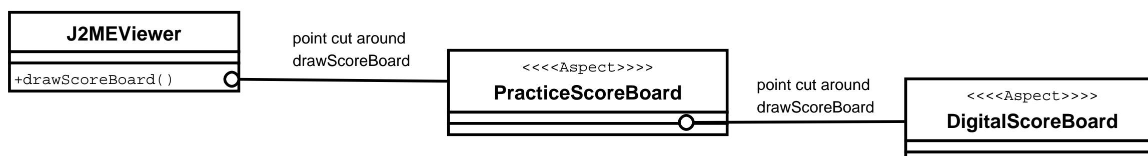

Figure 5.8: Practice Mode implemented using Aspects

which scoreboard variant is being used. Multiple feature interactions can be handled by creating an aspect for each one. The `declare precedence` keyword of AspectJ insures that the "wrapper" aspect, which handles checking for practice mode, is executed before the aspect displaying the scoreboard.

XVCL Mechanism

XVCL was also used to insert code to check for feature interaction. As shown in Figure 5.9, an additional frame is created, which has code to check the state of the modifying feature, then execute either the normal or modified behavior, depending on the state found, code example shown in List 5.6. An adapt command using a XVCL variable was placed at the variation point. Setting the variable chooses between the base behavior and the interaction check by including the appropriate frame. This can be used to omit the interaction checking code from the product if the modifying feature is not present; however, noting the existence of the modifying

```
public aspect PracticeScoreBoard {
        . . .
        void around(ScoreBoard gsb, Graphics g) :
                call(void ScoreBoard.paint(Graphics))
                && target(gsb)
                && args(g) {
                if (ActivationManager.getInstance().isPracticeMode()) {
                        . . .
                } else {
                        proceed(gsb, g);
                }
        }
}
```

Source Listing 5.5: Example of a feature interaction implemented with AspectJ

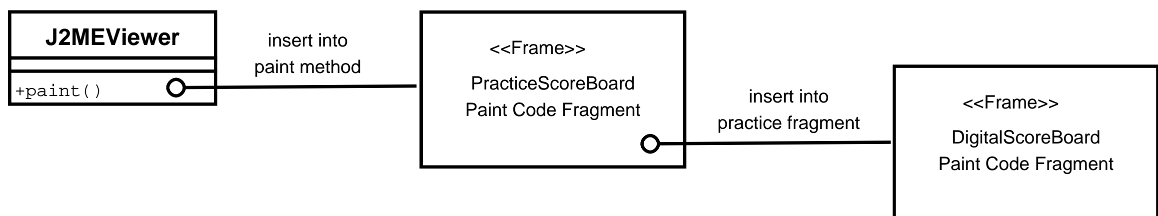

Figure 5.9: Practice Mode implemented using XVCL

feature and selecting frames accordingly is not automated by this pattern.

To handle multiple feature interactions, a frame is added for each of the feature interactions. Each frame handling a feature interaction includes an XVCL variable and an adapt command that uses the variable to specify the base and modified behavior. This allows the path through the code that would select the base behavior in a single interaction case to also select the code for an additional feature interaction. Ultimately, the runtime code to check for a feature interaction will involve an `if` statement. The frames containing feature interaction code are coded in such a way that selection for multiple feature interactions will result in a nested `if` statement in the code.

5.7.3 Implementation of Single Value Cardinality

Work on implementing a variation point with single value cardinality was conducted using the scoreboard variation point.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE x-frame SYSTEM "default">
<x-frame name="PracticeBV.xvcl">
                if (ActivationManager.getInstance().isPracticeMode()) {
                // practice mode specific code goes here
                } else {
                        // code for which ever scoreboard is selected
                        // by setting the @PRACTICMODE variable is
                        // inserted by the adapt directive
                        <adapt x-frame="?@PRACTICEMODE?BV.XVCL"/>
                }
</x-frame>
```

Source Listing 5.6: Example of a feature interaction implemented with XVCL

Coding Mechanisms

The scoreboard variants differ from each other only in what they display on the screen via their paint method. Therefore, using inheritance common functionality was placed in a parent class, GenericScoreBoard, which was extended by each variant. In addition GenericScoreBoard implements the `interface`, ScoreBoard. This allows the variant implementations to be instantiated and then invoked by the client either through the parent class or the Java language `interface`. Figure 5.10 provides a UML class diagram of this implementation. In this case the two approaches, inheritance and Java language `interface`, are used to produce the same benefit of isolating the client code from the choice of variant. Only minimal changes to the code are needed to pass either an object of the parent type or the interface. Using inheritance provides an easy way to combine the common code and variables from among the different implementations into one place. It should be possible to get a similar effect be creating an interface internal to the scoreboard type to allow common code to be shared. Depending on the situation there may not be enough common code to be worth making the effort with either approach.

Inheritance and Java language interface can be used to provide complimentary benefits. A module may have both common and variant functionality. The choice of how to factor the common code is independent of how it is presented to the module's client. In the case of the scoreboard, inheritance was used to gather the common code together in a parent class with the variant code in sub-classes. The complete module, common code plus a selected variant, was then presented to the client through a Java language `interface`.

Parameterization provides a good way of propagating a selection and of decoupling modules from the act of making the selection. However, it does not in itself provide an implementation. In this situation with the scoreboard, inheritance and Java language `interface` were used to isolate the client code from the

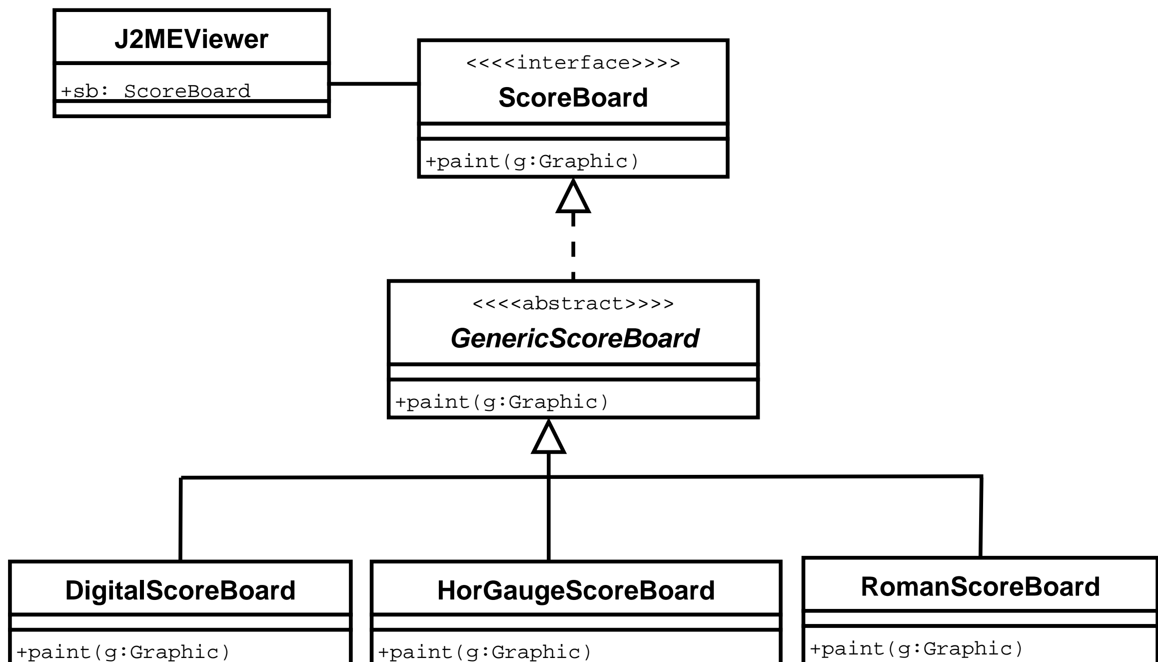

Figure 5.10: ScoreBoard UML

selection of different variants. The architecture may affect which approach is the most suitable. A framework often specifies that inheritance should be used to provide specialized functionality so that the framework may invoke the specialized functionality via polymorphic inheritance. Alternately, the architecture may require an interface. The scoreboard can be implemented using either inheritance or a Java language `interface` construct.

Aspect Mechanism

The scoreboard code was implemented two different ways using aspects. First, the abstract paint method of GenericScoreBoard was changed to be a concrete method with an empty body. Then an aspect was used to replace the empty paint method with one of the scoreboard implementations by using around advice. This approach, show in Figure Figure 5.11 worked, but seemed to confer no real advantages. The second approach was to eliminate the ScoreBoard-related classes and add the aspect directly into the viewer, as shown in Figure 5.12. The problem is that the code must go into the middle of the method. A dummy method was created and called to provide a place in the code to weave in the aspect. This worked, and since it replaced the ScoreBoard related classes, it could be consider simpler. On the other hand, the same code could be placed into the method and called just as easily.

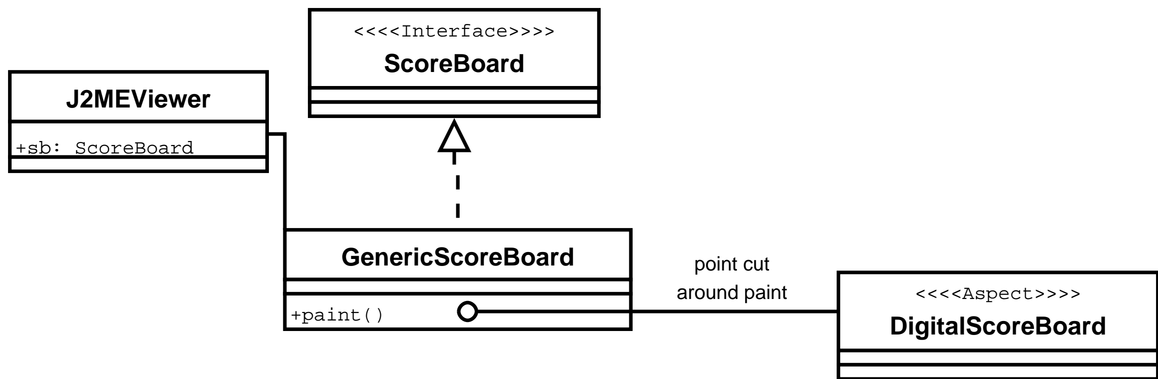

Figure 5.11: Including a Digital scoreboard using the first aspect approach

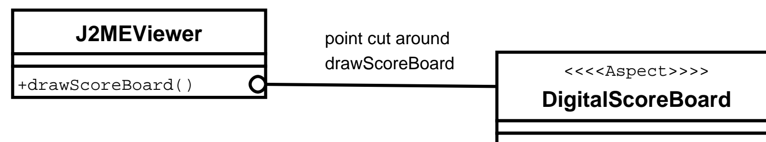

Figure 5.12: Including a Digital scoreboard using the second aspect approach

Aspects are selected as part of the build or class load process, they are not selected in the products source code. This may be either an advantage or a problem depending upon the situation being addressed. For scoreboard display, inheritance, interface, and parameterization all rely on the code instantiating a particular scoreboard variant. Changing the variant requires a change to the code, but it is explicit where this occurs. If the display variants are implemented as a series of aspects, only one of them may be selected as part of the build process. The level of granularity that AspectJ provides for this is that of the jar file. Different display aspects must be placed into different Java packages and then jar each package separately. Then the appropriate jar file must be selected based upon the product variant that should be included in the build process. This requires a build process that allows the jar file to be specified.

Aspects can be used to display the score; however, using aspects moves code from where it would normally be expected, the ScoreBoard paint method, to another file. While it is possible to replace a method call with a point cut, doing so adds complexity. In addition, displaying a scoreboard fits well with the main decomposition of the program. Hence, one of the main advantages of aspects, handling functionality that cuts across various program modules is not applicable. Using aspects for score display requires the scoreboard variant to be selected as part of the build process. Moving the selection of variants out of the code into the build script does not seem to be an advantage in this case. Even if it is preferred to select the variant in the build process, since these variants are replacements for each the same effect can be achieved by variants into

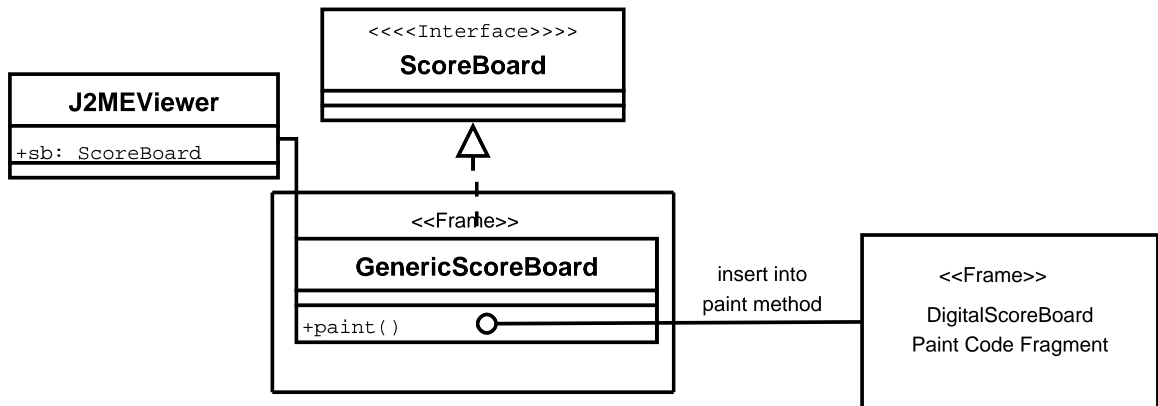

Figure 5.13: Insert a Digital scoreboard using the first XVCL approach

different libraries and then including only one library in the product.

XVCL Mechanism

The scoreboard was implemented two different ways using XVCL. For the first approach, shown in Figure 5.13, the abstract paint method of GenericScoreBoard was changed to a concrete method with an empty body. A frame was created for each scoreboard variant. Then the frame was inserted using an adapt command. Basing the frame name on an XVCL variable allows a particular scoreboard variant to be selected by changing the value of the variable. This approach created a working product, although it resulted in the same number of assets. The amount of Java code was reduced slightly because the overhead needed to define the scoreboard subclasses was removed. There is also a slight runtime improvement from replacing a polymorphic method call with a simple method call.

The second approach, shown in Figure 5.14, was to eliminate the ScoreBoard related classes and add the display code directly into the viewer. Once again, a frame was provided for each scoreboard variant. Then the contents of one of these frames was inserted using an adapt command. Using an XVCL variable for the frame name allows a particular scoreboard variant to be selected by changing the value of the variable. This approach also worked and allowed a number of improvements. The GenericScoreBoard class and ScoreBoard interface were eliminated, reducing the number of assets. ScoreBoard was eliminated as a parameter to the viewer. At runtime, the creation of a ScoreBoard object and related class variables was eliminated. Also, the call to a separate paint method for the ScoreBoard was removed.

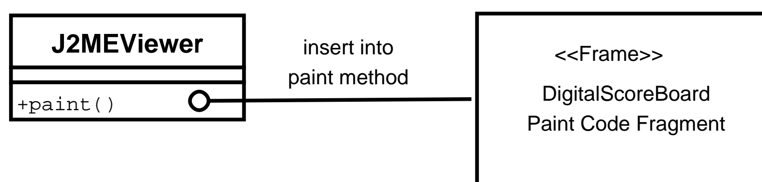

Figure 5.14: Insert a Digital scoreboard using the second XVCL approach

Table 5.2: Comparison of mechanisms implementing scoreboard

Mechanism	Number of Files	Runtime Objects	Compile Checking	Runtime Performance
Modular	6	1	Best	OK
XVCL	4	0	Best	Best
AspectJ	4	0	OK	OK

Summary

Scoreboards provide an example of a single value variation point. Both inheritance and Java language `interface` had similar advantages and disadvantages because they are closely related semantically. In both cases the Java language requires code overhead in the form of class and/or interface structure in order to separate out the optional code portions. The use of inheritance or `interface` requires that the different variants use the same method signatures. This was not a problem with scoreboards. In the single value cardinality, the variants are replacements for each other, which means that it is likely that their method signatures will be applicable to all variants. Having placed the variant code into a class, the class must be instantiated into an object, and a method called on that object at runtime in order to access the variant functionality. This causes more runtime overhead than placing the variant code directly into an existing class.

XVCL provided the most direct approach to inserting code fragments; code is simply inserted as needed using frames. The syntax needed to specify a frame and its insertion disappears by the time that code is produced. No runtime overhead is added; making this approach is the most runtime efficient of the modular approaches studied.

Aspects provide a way to add variants without affecting the existing code; this may be particularly useful for products already in the maintenance phase of development. There is some runtime penalty for using aspects relative to hard-coding the functionality [Hilsdale and Hugunin 2004]. Also, aspects trade some flexibility for compile-time error-checking; for example, a naming mistake will not cause a compile error. Table 5.2 compares the different mechanisms used in the test to implement scoreboard. In this example there

was an optional feature, scoreboard, with 3 variants and 1 interacting feature. For both XVCL and aspects, the table displays the second approach to the scoreboard variation point where the viewer was modified directly rather than modifying a scoreboard class.

The modular implementation uses a parent class to factor out common code and uses a class for each scoreboard variant. A Java language `interface` is provided as well. There is one subclass for each of the 3 display variants. The feature interaction uses a wrapper class following the decorator pattern. XVCL uses a frame for each display variant and an additional frame to support the feature interaction. AspectJ uses an aspect for each display variant and an additional aspect to support the feature interaction.

Compile time checking of code was not originally targeted as a research criteria; however, during code development using the different techniques for this research it become apparent that some techniques are better able to catch errors at the compile stage. In Table 5.2 the criteria for the "Compile Checking" column is the detection of name or signature mismatches between the client code and the components implementing the variation point. Techniques that produce an error for name / signature mismatches are marked as "Best". Techniques that do not produce an error until runtime are marked "OK". Both modular and XVCL produced code that takes full advantage of Java's thorough compile time checking. Aspects, in an effort to be more flexible, do not generate a compile time error if the method name / signature in the point cut does not match the client code. This means that accidental misnaming will not be caught until runtime.

Runtime efficiency was not originally targeted as a research criteria; however, during code development using the different techniques for this research it become apparent that some techniques are more runtime efficient than others. The observed inefficiencies were related to the need for additional memory allocation and / or additional levels of indirection. These observations are reported in Table 5.2 in the "Runtime Performance" column. Techniques that require additional memory allocation and / or additional levels of indirection are marked "OK". Techniques that do not require these are marked "Best". The modular approach creates a scoreboard object and makes a method call on it at runtime. Both XVCL and AspectJ insert code into the existing viewer class, which does not result in additional runtime objects. AspectJ does have some runtime overhead to call the aspect code. It should be noted that the performance differences were not apparent when running the products.

5.7.4 Implementation of Multi Value Cardinality

Work on implementing a variation point with multi value cardinality was conducted using the services variation point.

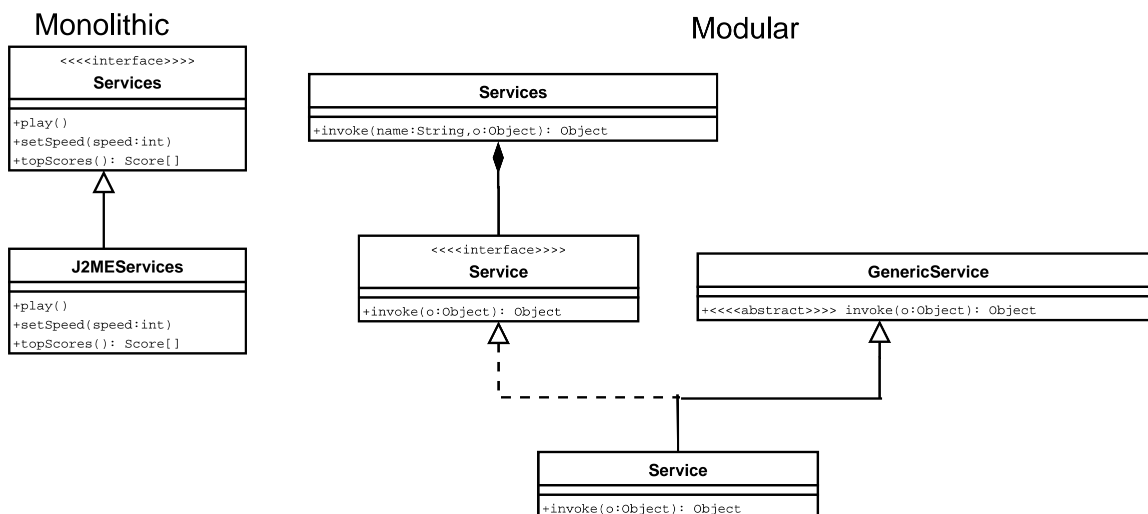

Figure 5.15: Services: Monolithic vs. Modular

Coding Mechanisms

A key difference between the single value and multiple value case is the need to support a set of variants in the product. As noted, parameterization can provide access to variants, but it does not actually create the variants, which is what is required here. By creating both a parent class, for common code, a sub-class for each variant's code, and an `interface` for the parent, both inheritance and `interface` may be used by the client. In this approach both inheritance and the `interface` mechanisms can be used to decouple the use of a particular variant from the client code. As a result references to the object implementing the variant may be selected either in terms of the parent class or in terms of the Java language `interface` implemented by the object implementing the variation.

To allow services to be selected independently, each service was placed into its own module. Doing this in standard Java requires making individual services their own classes, many of which were quite small, which will increase runtime overhead as each will have to be instantiated while provide only small additions to functionality. The existing services class becomes a composite, creating and providing access to a collection of service modules. Services no longer need a Java language `interface`, as it has no platform dependences. All the platform dependencies are in particular services, which are hidden behind an `interface` at the individual service level. Transforming the Services variation point in this way results in the *Command pattern* [Gamma et al. 1995]. This transformation is shown in Figure 5.15. An alternative approach is to group the services associated with a particular feature into the same module. For example, both setSoundLevel and

getSoundLevel are included in products that have the sound feature. This was not done for the pragmatic reason that most features have one to three services and the advantage of groups of this size did not seem to offset the complexity[3] of adding another level.

There were two problems in changing the initial monolithic implementation into a modular implementation:

- The variety of method signatures
- Common state implementation

Avoiding methods that are not common to all services is preferable. In Java, an implementing class must support all of the methods specified in the `interface` that it implements. With inheritance there is a little more flexibility. Each subclass could have its own methods. However, the methods specific to a particular subclass can not be accessed through polymorphism, but will require an explicit cast to allow them to be called. Using a cast introduces the potential of a runtime error. Checking the type to avoid a cast error adds runtime overhead. So inheritance also benefits from selecting a common set of service methods.

To accommodate all of the services, a very general method signature was introduced:

```
Object invoke(Object o)
```

For those services that need no arguments, the argument is ignored by the method and a null may be passed in. For those methods that have no need to return a value, a null is passed out and may be ignored by the client. While this is sufficiently flexible to handle all cases, it converts a class of errors from compile-time to runtime errors. Also, it requires additional runtime overhead; for example, primitives have to be wrapped. These issues are typical of using a Command pattern. They may be reduced by making the client aware of the context that the command is operating in; however, one reason for using an MVC architecture is to separate the commands from the model, which contains most of the context information.

Clients no longer access a service by directly calling a method. Instead the *Services* class, which holds the collection of different services, acts as a proxy. The client calls a method on *Services* with the signature:

```
Object invoke(String name, Object o)
```

The *Services* class searches the collection based on the name. Once found, the *Services* class calls invoke for that service, passes the `Object o` in, and returns the result to the client. Thus this proxy layer also moves

[3]To implement this an additional class to hold a group of related services is added. This class would have a collection to hold the objects for the services making up the group. An additional identifying parameter needs to be added, which must be supplied by the client with each service call, to specify the group the service being called belongs to. (Alternately, the group name could be looked up via a map.) Two runtime objects are added for each group of services (one for the services group object, one for the collection holding the services in the group). At runtime the group name must be looked up in addition to the service name.

a class of errors, incorrect service name, from compile to runtime. Also, overhead is added to find the correct service in the collection. As with all proxy approaches a single call is replaced by two calls, doubling the cost.

If a Java language `interface` for Services could be expanded with new methods in parallel with expanding the methods of the implementing object for Services a much cleaner interface could be provided to the client. While not possible with the J2ME used in this case study, it can be done in standard Java by taking advantage of reflection and the dynamic proxy classes in the Java library. This approach is known as an Enhancement Pattern [Hunt and Sitaraman 2004].

The second problem is the state information needed by services. Some examples include the name of the persistent storage location, the number of top scores recorded, and references to elements of the game, such as the board. Different services require different state information. In the monolithic version, class variables made this state information global to all of the services to either use or ignore. Placing each service into its own class requires the needed state information to be accessible in those classes. It would be possible for each of the classes to have variables, and the related constructors, specific to that particular service; however, this level of customization seems excessively complex. It also results in multiple copies of the state information. The solution adopted was to keep the state information in the services class that holds the collection and pass a reference to this class when creating the individual service objects. Since the individual service objects are not intended for stand alone use, no loss of generality results from further tying them to the holding class.

This approach places most of the implementation of variability at runtime. The current implementation hard-codes which service objects are instantiated and placed into the collection. In some ways, this gives us the worst of both worlds - the inefficiency of runtime binding with the inflexibility of a construction binding time. With a fairly small amount of effort, reflection could be used to convert this fully to runtime binding if standard Java is used; however, J2ME does not support reflection. Even if J2ME did support reflection, in this case runtime binding is not needed. As can be seen in this example, runtime binding can result in additional classes of runtime errors and performance overhead without providing additional benefits.

Another issue is efficiency when a single variant is selected. The collection and the need to search it are unnecessary overhead for a single item. This can be handled by a proxy class that contains a reference to a single service, rather than a collection. This provides increased efficiency without affecting its clients. Continued use of a proxy class continues the overhead of two calls.

The underlying problem for both inheritance and Java language `interfaces` is that they do not allow modules to consist of arbitrary code fragments. As a result, even small bits of functionality require the

overhead of a class and the instantiation of a separate object. In addition, these must be composed at runtime, which in many cases provides flexibility we don't want at a cost we would rather not pay. The services example is something of a worst case scenario, as it provides small grain functionality, ranging from a single line of code in many cases up to a page of code at most. In those cases where larger amounts of functionality are provided, the overhead involved becomes less significant.

Aspect Mechanisms

Aspects allow features to "install themselves" without modifying the main line code. If features are designed appropriately, aspects can provide important advantages in building the product. This may increase the complexity of the feature code, but in general, the resulting simplicity in the main line code is a good trade off. Implementing services with aspects provides a good example of a favorable trade off. Initially, services were not broken into smaller units, in part due to complexity it would have added to the client code. The self installing nature of aspect allows services to be broken into smaller units without increasing the complexity of the client. Also, by installing the correct aspect for a particular platform, client code is isolated from platform variants without using an `interface`.

In the aspect implementation of services, shown in Figure 5.16, all the code related to optional features is removed from the J2MEServices class. The required services remain in the J2MEServices class, which is still explicitly instantiated and called by clients. This provides a convenient place to hook the aspects for optional features. The aspects for the various optional services instantiate themselves where the instantiation of the required services takes place, using `after` advice. Menu classes make calls to empty methods for the various optional services, which the aspects for these services then hook onto using `around` advice. Since the client code is unaware of which aspect is being hooked, its code does not need to be edited when the platform variant is changed. The required services are platform independent. As a result, the Services `interface` can be eliminated without introducing platform dependencies into the client, allowing some code simplification. Unlike the modular class approach, there are no issues with grouping related services into the same aspect. As is generally recommended with aspect development, aspects that had large amounts of code were factored into supporting classes. The case where only a single optional service is selected does not require a change in the code, unlike the coding approaches that use a collection.

Overall, aspects were successful in allowing us to exclude optional features from the product while improving code cohesion. The aspects to be applied are selected in the build process, typically at a jar level of granularity, which means that the asset base must be organized carefully to support the use of aspects.

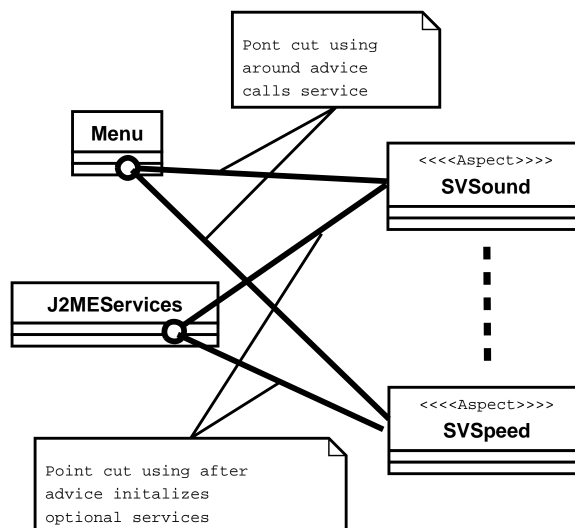

Figure 5.16: Services implemented with Aspects

XVCL Mechanisms

Using XVCL allows code fragments to be used directly without creating new additional classes, see Figure 5.17. An arbitrary amount of code can be placed in a frame, allowing as many or as few services in the frame as is convenient. Therefore, one frame per feature, rather than one per service, is used. As many methods as needed by a particular service are added in a single frame. This allows private methods and class variables to be shared between services for the same feature. A set of frames for a particular product variant is specified using a multi-var, an XVCL variable that holds a list of values, as shown in List 5.7. A loop on the multi-var includes those particular frames. Method signatures specific to each service are used, which means that name and signature mismatches generate compile time rather than runtime errors. It also avoids the runtime overhead of maintaining and searching a collection and of the additional calls on a proxy object.

Frames allow the code for various platforms to be plugged directly into the services class. If the platform does not affect the method signature, there will be no effect on the client from changing platforms, eliminating the need for a separate Services Java language `interface`. Also, there is no change in the coding approach when a single feature selected; simply insert less code.

Using XVCL provides the advantages of the monolithic approach, while inserting only those services that are needed for a particular product. In cases where the code is called from only one location and where the amount of code for each variant is small, using XVCL frames to implement variants may be the best choice.

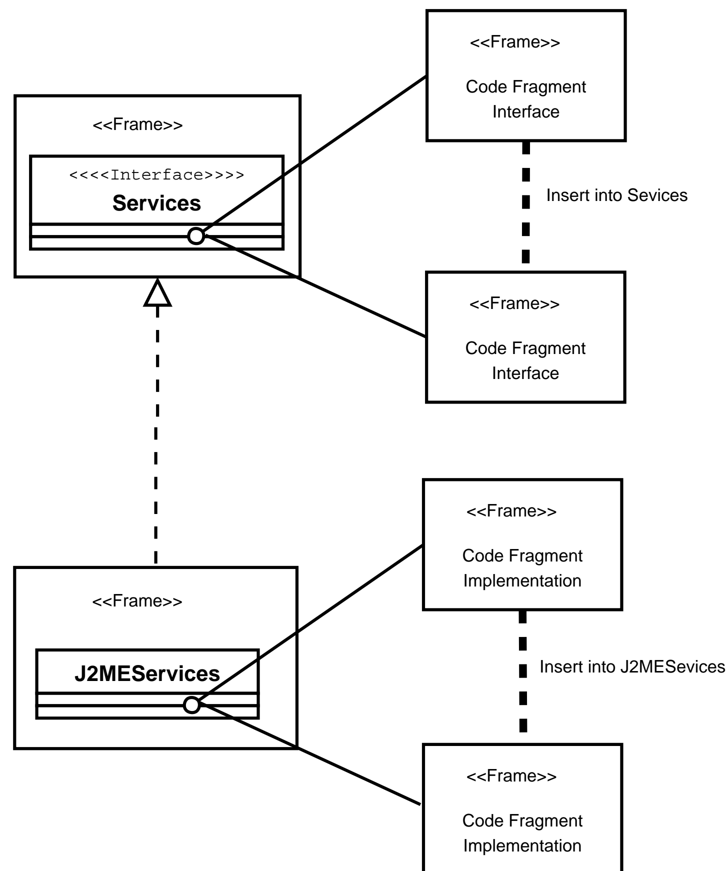

Figure 5.17: Services implemented with XVCL

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE x-frame SYSTEM "default">
<x-frame name="J2MEServices.xvcl" outfile="J2MEServices.java" language="java">
<set-multi var="ServiceChoices"
        value="SVPauseUn,SVSaveLoad,SVScores,SVSound,SVSpeed"/>

public class J2MEServices implements Services {
        . . .
        <while using-items-in="ServiceChoices">
                <adapt x-frame="?@ServiceChoices?.XVCL"/>
        </while>
}
</x-frame>
```

Source Listing 5.7: Example Implementing Multi Selection with XVCL

Summary of Services Variation Point

Services provide an example of a multi value variation point. All four mechanisms provide ways to insulate client code from platform dependencies without using a Services `interface`.

Both inheritance and Java language `interface` had similar advantages and disadvantages. In both cases Java requires code overhead in the form of class and/or `interface` structure in order to separate out the optional code portions. Modules based on class and/or `interface` required choosing a single method signature for all services. This resulted in a very general signature which reduced the comprehensiveness of compile time checking. Related services were not grouped to avoid an additional level of selection. This approach required the creation and searching of a collection of services at runtime. The runtime search provides a whole new category of runtime errors that can occur. Runtime overhead is added to each service call, both to find the right service and to marshal and un-marshal argument and return values from the internal data values into the common signature format. This proxy approach also required an additional call for each service invocation. Platform differences are handled by instantiating different service implementations at runtime.

XVCL provides the most direct approach to inserting code fragments; code is simply inserted as needed using frames, albeit only at pre-defined points. Related functions are inserted together by placing them in the same frame. The syntax needed to specify a frame and its insertion disappears by the time code is produced. No limitations are placed on method signatures, allowing code to take full advantage of compile-time checking. This is the most runtime efficient of the modular approaches studied. Platform differences are handled by inserting different frames at pre-processor time.

Aspects provide a way to add additional services without the existing code being aware it is modified; this may be particularly useful for products already in the maintenance phase of development. The complexity of the inserting the additional service is hidden from the client code. As a result, issues such as having to initialize each service separately do not affect the client code. Aspects are able to group related services together. There is some runtime penalty for using aspects relative to hard-coding the functionality. Also, aspects trade some flexibility for compile-time error-checking; for example, a naming mistake will not cause a compile error. Platform differences area handled by specifying different jar files during the post-compile build process.

Table 5.3 compares the different mechanisms used to implement services. In this example there were 5 required services and 8 optional services in four related groups of 2 services each. The monolithic mechanism places all of the services related code into one file. The second file is an `interface` to allow implementations

Table 5.3: Comparison of mechanisms implementing services

Mechanism	Number of Files	Runtime Objects	Compile Checking	Runtime Performance
Monolithic	2	2	Best	Best
Modular	16	15	Poor	Poor
XVCL	5	1	Best	Best
AspectJ	7	7	OK	OK

for other platforms to be added without affecting the client. Note that the monolithic version does not meet the requirement to be able to exclude optional features from the product. It is included as a base line of comparison for other approaches. The modular code uses a class to hold a collection of services. The implementation uses a service parent class to factor out common code and uses a class for each individual service; related services are not grouped. XVCL uses a frame for the services class that includes the required services and a frame for related features. AspectJ uses a class for the services that includes the required features and an aspect for each group of related features. In addition, two of the feature groups had significant amounts of code, which led the related aspects to have an additional class factored out of them.

Compile time checking of code was not originally targeted as a research criteria; however, during code development using the different techniques for this research it become apparent that some techniques are better able to catch errors at the compile stage. In Table 5.3 the criteria for the "Compile Checking" column is the detection of name or signature mismatches between the client code and the components implementing the variation point. Techniques that produce an error for name / signature mismatches are marked as "Best". Techniques that do not produce an error until runtime are marked "OK". XVCL produced code that takes full advantage of Java's through compile-time checking. In putting all of the services behind the same method name and signature, the modular approach largely eliminates compile-time checking. It also introduces a class of runtime errors where the client requests a service that doesn't exist in the collection. Aspects, in an effort to be more flexible, do not result in a compile time error if the method name / signature in the point cut does not match the client code. This means that accidental misnaming will not be caught until runtime.

Runtime efficiency was not originally targeted as a research criteria; however, during code development using the different techniques for this research it become apparent that some techniques are more runtime efficient than others. The observed inefficiencies were related to the need for additional memory allocation and / or additional levels of indirection. These observations are reported in Table 5.3 in the "Runtime Performance" column. Techniques that require additional memory allocation and / or additional levels of

indirection are marked "OK". Techniques that do not require these are marked "Best". The monolithic code creates a single runtime object for all of the services. The modular approach creates on object for the services access point, which holds a collection. The collection is counted as a single runtime object. An object is needed for each of the services. Code created by XVCL creates a single runtime object for all of the services that were specified at construction time. The AspectJ example uses a conventional class for the required services. There is an aspect for each group of optional features, each of which will result in a runtime class. Two of the aspects had enough code to warrant being factored into an additional class private to the aspect, resulting in two additional runtime classes. It should be noted that the performance differences were not apparent when running the products.

5.8 Generalizing the Implementation

Thus far I have presented implementations that are specific to particular variation points. Can the results be used to provide implementation artifacts that can be reused at more than one variation point? This depends on the implementation techniques used. Different variation points will most naturally use different types, each of which will have its own methods, and method signatures. Implementation techniques that do not rely on or can abstract types can be generalized. In some cases this can be done by having the types implement the same `interface` or sub-class the same parent. Once this is done objects created by the types can be manipulated without concern over which particular type is used.

XVCL is able to manipulate code independently of types (classes and `interfaces`) occurring at the Java language level. For example, List 5.8 shows an XVCL frame which implements the part of the practice mode feature interaction with the scoreboard. It is fairly straightforward to make this code more abstract through increased parameterization, changing it from being an implementation for a specific feature interaction to being a template that can be used for any variation point that follows the Lee and Kang feature interaction pattern [Lee and Kang 2004]. The abstract version is shown in List 5.9.

Other variation point characteristics, such as multi value selection, that have been implemented in XVCL should be amenable to similar abstraction into templates. As always, there are trade offs. The template frames may be more difficult for a product developer to understand, a problem common to increasing the level of abstraction. Additional frames will be needed, increasing the number of assets in the asset base. On the other hand it should be noted that XVCL lends itself to automation and that the additional frames are resolved at construction time and do not add any runtime performance penalty. Also, since XVCL produces code which

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE x-frame SYSTEM "default">
<x-frame name="PracticeSB.xvcl">
        if (ActivationManager.getInstance().isPracticeMode()) {
                g.setColor(color);
                String scoreStr = "Practice Mode";
                int offset = (g.getFont().stringWidth(scoreStr)) / 2;
                g.drawString(scoreStr, r.getLocation().getRealX() -
                        offset, r.getLocation().getRealY(),
                        Graphics.TOP | Graphics.LEFT);
        } else {
                <adapt x-frame="?@PRACTICEMODE?SB.XVCL"/>
        }
</x-frame>
```

Source Listing 5.8: XVCL Implementation of the Practice Mode Feature Interaction with the Scorebaord Variation Point

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE x-frame SYSTEM "default">
<x-frame name="PracticeSB.xvcl">
        if (<adapt x-frame="?@CONDITION1?.XVCL"/>) {
                <adapt x-frame="?@INTERACTIONCASE1?.XVCL"/>
        } else {
                <adapt x-frame="?@REGULARCASE1?.XVCL"/>
        }
</x-frame>
```

Source Listing 5.9: XVCL Template to Implement Any Feature Interaction

will then be compiled, substituting the wrong type of frame (although not the wrong choice of a frame) will probably generate a compile time error. This helps to avoid runtime bugs, as results from resolving the type problem by forcing all the methods at the variation points to look the same, as is done in the *selection proxy* approach, Section 5.10.

XVCL can be used to provide a nice division of labor where the XVCL frames are focused on providing the variation point characteristics, while the Java code produced focuses on the business logic of the application.

Java is intended to be a strongly typed language, and method signatures are very much a part of its typing system. This has an impact on the use of Java implementation techniques, inheritance and `interface`. Much of the work done to implement a modular version of the multi value selection involved working around Java's type system by giving all types that can be selected the same methods, even though they were providing somewhat different functionality. While the solution presented has a number of significant issues (increase in runtime errors, performance penalties, the need to marshal / un-marshal parameters and return values, etc.) which have been discussed in Section 5.7.4, it is quite general in approach, and has few ties to the particular variation point being implemented. It would take only a little effort to further abstract the code used to handle any multi value selection variation point. However, extensive use of code designed to bypass the Java language type system is problematic. Also, this approach makes the implementation of the variation point a focus of the class decomposition, which may distract from the business logic that should be the programs main goal.

AspectJ's contribution to variation point implementation is to act as glue code, seamlessly integrating separately written Java classes via a point cut. The difficulty is that the point cut code becomes very specific to particular places in the program. There does not seem to be a good way to abstract the point cuts, which make up most of the aspect oriented code used in implementing variation points.

Thus, depending on the implementation technique used it may be possible to get some significant implementation artifact reuse between the variation points in the asset base.

5.9 Conclusion by Implementation Approach

Based on my experiences with implementing variation points in the PPL, I draw the following conclusions about the different implementation approaches examined.

5.9.1 Parameterization

Brown et al. [Brown et al. 2002] presents parameterization as an approach to implementing variation points; however, parameterization in itself does not create functionality. It might be better seen as providing a good way to present or access functionality created elsewhere in the program. As such, it is a useful technique and can be combined with either inheritance or Java language `interface`. Either an `interface` or an object via a parameter can be passed in to a client module.

Parameterization is part of a strategy to allow knowledge of the choice of a particular variant to be decoupled from the parameterized client. Parameterization was used to allow all the variants in the product to be created in one place, a factory pattern, and then passed to the modules that needed them. Parameterization can be used in conjunction with selecting variants dynamically at runtime. A feature may be "omitted" by passing a null pointer through the parameter; however, the client code must check for a null reference before each access attempt, which both complicates the client and affects the runtime performance. Parameterization may be used in a feature interaction context by either passing or presenting different objects, depending upon the state of the system.

5.9.2 Inheritance

Object Oriented (OO) design has become a basic approach to providing program functionality. There is a wealth of information on using OO to build products. This makes OO a good approach for implementing the basic functionality of a feature. For a feature with variants, inheritance can play two roles. Within the module implementing the feature, the parent class provides a way to collect and reuse the code common to all variants, while the sub-classes provide a way to collect and associate the variant code with the module. It also provides a way to encapsulate a variant feature so that the client of the module is isolated from the variations, which are called via polymorphism. When inheritance is used as the variation mechanism, a variant is selected by instantiating the appropriate subclass at runtime. Variation points implemented in this way may provide runtime binding. However, if desired a particular variant selection can be coded during program construction to provide an earlier binding time. A feature may be "omitted" by using a subclass containing methods with empty bodies, which affects the runtime performance; although not as much as if variant processing occurred.

It might seem that the additional functionality needed to support feature interaction could be added via a subclass and invoked using polymorphism. However, Java forces us to hard-code the name of the class

being extended, which severely limits its applicability. This is a case where C++, which allows the choice of parent class to be parameterized, has an advantage. XVCL could be used to specify the parent class in a more flexible way. Given the Java limitation, a different subclass is needed for each variant or a variant for each feature interaction. As a result, a *decorator pattern* was used to add feature interaction into any inheritance implementations.

5.9.3 Java Language Interface

Java language `interface` is another well understood and general mechanism to implement program features. For a feature with variants, a Java language `interface` can play two roles. Within the module implementing the feature, an `interface` can be used to access the variant. It also provides a way to encapsulate a variant feature, providing a single way to call any variant. With an `interface`, a variation is selected by instantiating the appropriate class, which allows runtime binding. A feature may be "omitted" by using a module containing methods with empty bodies, which affects the runtime performance; although not as much as if variant processing occurred.

5.9.4 Aspects

Aspects provide the ability to add functionality without modifying existing code. Aspects are added relative to existing code such as a method call. It may be difficult to add the aspect code to an arbitrary point in the program; therefore, aspects are not recommended for implementing optional features. However, aspects work well to provide optional variants of a required feature. Aspects can be used to provide method bodies; however, doing so seems to offer no advantage over conventional methods. They require defining a point cut, separate the functional code from its call, and add complexity to the build process. Therefore, I do not recommend aspects to replace variant code. Aspects are good at modifying existing actions. This makes aspects an excellent choice for handling feature interactions.

The inclusion of aspects, unlike classes and Java language `interface`, is handled outside of the code as part of the build process. This adds a second opportunity to make product choices. While this provides flexibility, it also requires management. The granularity with which aspects are added to the product is that of a jar file. Typically a package should not be split between jars; therefore, when organizing code into packages, we need to take into account which aspects need to be separated. For example, the scoreboard variants were implemented using aspects (not recommended), then each of the scoreboard variants will have

to be put into its own package and jar. If they were put into the same package / jar, then including that jar in the build would cause multiple choices for the display code to be active in the product at the same time. On the other hand, if there are different pieces that may be added or removed from a product at the same time, placing them in the same jar may work well.

Using aspects requires more attention to the way source code is organized into packages. Including a jar in the build that contains an aspect changes the program's control flow. If aspects are used to handle feature interactions, then it becomes particularly compelling to organize the code based on features as discussed in "Organizing the Asset Base for Product Derivation" [Hunt 2006].

5.9.5 XVCL

XVCL is able to completely omit feature code from the product, providing the best way to omit optional features. It is also able to insert feature code at arbitrary points in the product. This may allow a simpler and more efficient implementation. For example, the initial class-based approach required a parent class to support various subclasses. Each of the subclasses required some overhead code to declare the subclass. However, XVCL has no need for the parent class or the overhead involved in declaring the subclasses. Only the code to display the particular scoreboard is needed. XVCL can be used to insert code to check for feature interactions. However, the processor directive to include the check must be placed in the main feature. Thus, the developer must be aware of the existence of the interacting feature and will have to modify the code to handle new interacting features. XVCL may be the best way to include small sections of code, particularly if they are used in a single place.

5.10 A Pattern Language for Implementing Variation Points

5.10.1 Introduction

In this chapter[4] I have discussed three implementation issues that have not generally been addressed - cardinality, optionality, and feature interaction. It is desirable to make advice related to these issues applicable to many situations, yet detailed enough to provide useful guidance. The format I have decided to use to provide this advice is a pattern language.

"A *pattern language* is a structured method of describing good design practices within a particular do-

[4] Content in this section previously published in [Hunt and McGregor 2006]

main. Pattern languages are used to formalize decision-making values whose effectiveness becomes obvious with experience but that are difficult to document and pass on to novices. They are also effective tools in structuring knowledge and understanding of fundamentally complex systems without forcing oversimplification" [Wikipedia 2006]

5.10.2 A Pattern Language for Variation Points

Name: ***Implement a Variation Point***

Pre-Condition: In the course of implementing a variant SPL feature we realize the need to provide for variation at a particular point in the code; that is, we recognize the need to implement a variation point.

Problem: SPL research has provided guidance for variation points at the design level, but has provided only limited guidance on implementing a variation point. Much of this guidance has been deliberately language independent to make it more general. This results in guidance that does not deal with the language limitations with which a developer must cope.

A variation point must provide the opportunity to select a particular value from a set of variants. Depending on the problem addressed, and thus the design for solving the problem, the selection might be limited to a single value or open to include multiple values in the same product. This difference is simply expressed at the design level using a feature model. However, implementations for single and multi value selection will differ from each other. Advice on these differences has not typically been provided.

At the design level features may be optional, meaning that they may be omitted from the product. However, the problems of omitting code related to an optional feature completely from an implementation are rarely addressed.

Feature interaction is not normally discussed in relation to variation points. However, feature interaction requires that an implementation modifies its behavior depending upon the features selected for a given product. Since we would like to limit the effect of the feature selection on an implementation to the variation points, we must cope with feature interaction as part of the variation point implementation.

Implementation advice for developers should address the consequences of these three types of design choices - cardinality, optionalness, and feature interaction.

Therefore:

Solution: Our pattern language provides advice for handing these three different types of design decisions that affect a variation point. They are:

- *Implement Variation Point Cardinality*
- *Implement Optional Variation Point*
- *Manage Feature Interactions*

Either *Implement Variation Point Cardinality* or *Implement Optional Variation Point* may be done first. *Manage Feature Interactions* should be done after the other two. All variation points must address cardinality; hence, will use *Implement Variation Point Cardinality*. Variation points related to mandatory feature will not need to *Implement Optional Variation Point*. Not all features interact with other features, so not all variation points will need to *Manage Feature Interactions*.

Constraints: This pattern language provides advice specific to implementing a variation point using Java. Two additional tools will be discussed that are compatible with Java - XVCL, a language independent general purpose tool that we use as a pre-processor, and AspectJ, which provides an aspect extension to the Java language. While we use XVCL, the techniques we discuss should be applicable to any pre-processor.

Name: ***Implement Variation Point Cardinality***

Problem: For some features only one value at a time may be selected. For example, a car must have exactly one transmission; however this may be either a manual or an automatic transmission. In other cases selecting multiple values at the same time is possible. A car's sound system may (or may not) have a radio, a CD player and a tape deck. The implementation of the variation point must be able to support the cardinality of features specified in the design.

Little distinction has been made between implementing a single value variation point as opposed to a multi value variation point. An example of the differences can be seen in the widely discussed approach of using inheritance to implement the variants. To isolate the client code from the choice of a particular variant we can write the client code in terms of the specification provided by the parent class and implement the variations as sub-classes. For the single selection case we hold a reference of the parent type and instantiate one of the sub-classes. This naturally gives us a single choice. However, in the case of a multi selection variation we need to create an object for each selected variant. We will need to hold these in a collection which will then have to be managed. Yet, the additional problems of managing the collection are typically ignored when inheritance is recommended as a way of implementing variants.

A mechanism that can manage multiple selected values can also manage a single value. Thus, it could be argued that only the multi value case needs to be considered. However, using a multi value mechanism for

a variation point that holds only a single value imposes unnecessary costs. These costs occur both in terms of program execution, requiring additional memory and CPU cycles; and cognitive costs to the programmer, due to more complex code. Separate advice specifically for the single value variation point will be provided.

Therefore:

Solution: Based on the product design the developer first notes if a single or multi value variation point is called for. If the design specifies that only a single variant may be selected for a product then *Implement a Single Value Variation Point*. If the design specifies that multiple variants may be selected for a product then *Implement a Multi Value Variation Point*.

Name: ***Implement a Single Value Variation Point***

Pre-Condition: The design allows for several possible choices at this site in the code and only one of these choices may be included in a particular product.

Problem: The design allows for several possible choices at this variation point; however, we wish to isolate the client code outside the variation point from changing when different variants are selected. Aspects are not recommended for this case because they do not naturally exclude each other.

Therefore:

Solution:

- If an early binding time is acceptable and the variant requires small amounts of code that appears in only a few places, consider *Bind Single Value Early*.
- Both inheritance and interface implementation can be used to build features and to isolate which variant of a feature is selected from client code. The approach to prefer is largely dependent on the architectural context of the product. Both inheritance and a Java language interface provide a natural way to limit selection to a single variant. For inheritance, this is done by having the sub-class that provides the variant extends a parent class used by the client code. For Java language interface, this is done by having the class that provides the variant implement the interface. In this case, the client holds a reference to the interface and instantiates one of the classes that implement the interface.

  Chose an interface definition that will suffice for all the variants; which should simplify having each variant work with a parent class or interface. This should be possible in the single selection case as

the variants should be substitutes for each other. Both inheritance and Java language interface select a variant by instantiating an object. This means that the binding time may be as late as runtime, if the selection is from a closed set of variants known at construction time. It is possible in Java, but not J2ME, to extend these to an open set of variants by using reflection to instantiate classes not known at construction time. Selection and instantiation of variants may be centralized using the GOF *Factory Pattern*, and made available throughout the code using parameterization.

Name: ***Implement a Multi Value Variation Point***

Pre-Condition: The design allows for several possible choices at this site in the code and more than one selection may be included in the product.

Problem: The design allows for several possible choices at this variation point; however, we wish to isolate the client code outside the variation point from changing when different variants are added to the product. The implementation must support multiple selections being included and active in the product at the same time.

Therefore:

Solution:

- If an early binding time is acceptable and the variant requires small amounts of code that appears in only a few places *Bind Multi Value Early*.
- If a later binding time is desired *Bind Multi Value on Load* may work if there are appropriate places to hook on aspects.
- If places to hook aspects can not be found, or if runtime binding is required, then a *Selection Proxy* may be used.

Name: ***Implement Optional Variation Point***

Pre-Condition: The variation point belongs to an optional feature.

Problem: If the feature has been omitted from a particular product variant we should completely omit the related code at the variation point without leaving a trace of the feature. Note that the binding time of including or omitting a feature may be different than the binding time of selecting a variant. For example, we can

decide that a particular product variant will include a scoreboard at construction time, but allow the user to select which type of scoreboard will be displayed at runtime.

Therefore:

Solution:

- If we know whether to include the feature at construction time *Implement Optional Variation Point Early*.
- If there is a suitable program construct, such as a call that can be used as a hook for an aspect, and a binding time of program load is acceptable to decide if the feature is included *Implement Optional Variation Point*.

Name: ***Manage Feature Interactions***

Pre-Condition: The variants for a variation point have been implemented. The developer for the variation point code has realized that this variation point will affect another existing variation point.

Problem: Feature interaction involves two features - F1, F2. If F2 is added to the product, then the behavior of F1 changes depending on the state of F2. An example from the PPL is to consider the scoreboard as a feature whose behavior is changed if the practice mode is added to the product.

When implementing feature interaction, we want to leave the code related to the feature interaction out of the product if the feature that introduces the interaction is left out of the product. Preferably we would like to implement the interaction in such a way that a feature is not aware of interacting features. A given feature may be affected by more than one feature, and thus take part in multiple feature interactions.

Therefore:

Solution: Use aspects to implement the feature interaction without affecting the feature implementation code. This is a great advantage in avoiding code tangling between features and one that only aspects were able to deliver. Multiple feature interactions can be implemented by defining aspects for each, which has the nice property that they can be unaware of each other.

Place the feature interaction code in its own Java package. Write an AspectJ point cut to add the needed calls to each of the variation points in the feature affected by the feature interaction, the point cut file is added to the package containing the feature interaction code. Produce a jar file containing the package. Including

the jar on the command line running the program, along with the AspectJ runtime jar file, will cause the feature to install itself into the product at program load time. Note that the feature code does not need to be modified to add the feature interaction code.

This pattern can be extended to handle multiple feature interactions. We assume that the different interactions involving the feature are independent of each other, but each feature needs to have an opportunity to check the state it is concerned with before the default behavior for the variation point executes. This functionality is provided by adding a *declare precedence* keyword to the point cut file of the modifying feature.

Consequences:

- To control the addition of the feature the code must be organized into its own package and the build process must produce a jar file.
- Features may be added without modifying existing code.
- Features may be omitted as late as program load time.
- The code must have program features, such as methods, that can serve as hooks for the point cuts. Aspects are not designed to insert code at arbitrary points in the program.

Name: ***Implement Optional Variation Point Early***

Problem: We wish to completely omit an optional feature from the code without leaving a trace. We know by construction time if the feature is being included in the product.

Therefore:

Solution: Use XVCL to cleanly omit the variation point's code while inserting the variation point at an arbitrary place in the code. Begin by creating two frames; one containing the code related to the feature at this variation point, the other frame has no content. An adapt command using a XVCL variable is placed at the variation point. Set the variable to choose between the frame with the feature related code and the empty frame. Select the empty frame to omit the feature code from this variation point. An empty frame has the effect of inserting a single space into the code which will not cause any Java code generation.

Consequences:

- The choice to include the feature must be made during the build phase, prior to compilation.

Sample Code:

```
<set var=``SCOREBOARDTYPE'' value=''Digital''/>
. . .
<adapt x-frame=``?@SCOREBOARDTYPE?BV.XVCL''/>
```

Related Patterns:

- Providing multiple frames and related variables extends this pattern to *Bind Single Value Early*.
- Include an empty frame as a choice to allow a multi selection feature to be omitted from the product, thus combining this pattern with *Implement Optional Variation Point Early*.

Name: ***Bind Single Value Early***

Problem: We wish to select from among different variants to implement a variation point. Selections are mutually exclusive.

Therefore:

Solution: Create a frame to hold the code for each variant. An adapt command using a XVCL variable is placed at the variation point. Setting the variable chooses a particular variant by including its frame.

Consequences:

- The choice of which variant to include must be made at during the build phase, prior to compilation.

Sample Code:

```
<set var=``SCOREBOARDTYPE'' value=''Digital''/>
. . .
<adapt x-frame=``?@SCOREBOARDTYPE?BV.XVCL''/>
```

Related Patterns:

- Include an empty frame as a choice to handle optional features as described by *Implement Optional Variation Point Early*.
- To allow multiple selections to be chosen at the same time *Bind Multi Value Early*.

Name: ***Bind Multi Value Early***

Problem: We wish to select one or more variants from a set of possible variants.

Therefore:

Solution: Create a frame to hold the code for each variant. The name of the frame for each variant to be included is set into a XVCL multi var. XVCL multi var's allow a list of values. An XVCL while command feeds each of the selections from a multi var into an adapt command.

Consequences:

- The choice of which variant to include must be made at during the build phase, prior to compilation.
- Efficient code is automatically produced when only one variant is selected.

Sample Code:

```
<set-multi var="ServiceChoices"
  value="SVPauseUn,SVSaveLoad,..."/>
. . .
<while using-items-in="ServiceChoices">
<adapt x-frame="?@ServiceChoices?.XVCL"/>
</while>
```

Related Patterns:

- To insure an exclusive selection from a set use *Bind Single Value Early*.
- Including an empty frame as a choice extends this pattern to handle optional features as described by *Implement Optional Variation Point Early*.

Name: ***Implement Optional Variation Point***

Problem: We wish to completely omit an optional feature from the code without leaving a trace.

Therefore:

Solution: Place the optional feature code in its own Java package. Write an AspectJ point cut to add the needed calls to each of the variation points affected by adding the feature, the point cut file is added to the feature's package. Produce a jar file containing the package. Including the jar on the command line running

the program, along with the AspectJ runtime jar file, will cause the feature to install itself into the product at program load time.

Consequences:

- To control the addition of the feature the code must be organized into its own package and the build process must produce a jar file.
- Features may be added with modifying existing code.
- Features may be omitted as late as program load time.
- The base code must have program features, such as methods, that can serve as hooks for the point cuts.

Related Patterns:

- Breaking code for a feature into multiple packages and jar files results in *Bind Multi Value on Load*.

Name: ***Bind Multi Value on Load***

Problem: We wish to select one or more variants from a set of possible variants.

Therefore:

Solution: Use aspects to implement the multi-selection case, each aspect can be added independently without concern for being an exclusive choice. Aspects install themselves, this is particularly useful if multiple points in the code are affected. Place code for each separately selectable feature in its own Java package. Write an AspectJ point cut to add the needed calls to each of the variation points affected by adding the feature, the point cut file is added to the feature's package. Produce a jar file for each selection containing the package. Include the jar on the command line when running the program to install the feature at program load time.

Consequences:

- To control the addition of the feature the code must be organized into its own package and the build process must produce a jar file.
- Selections may be added with modifying existing code.
- Selections may be omitted as late as program load time.

- The base code must have program features, such as methods, that can serve as hooks for the point cuts. Aspects are not designed to insert code at arbitrary points in the program.

- The case where only one selection is made is handled in an efficiently without additional coding.

Related Patterns:

- If there is a single selection to be either included or omitted this approach becomes *Implement Optional Variation Point*.

Name: ***Selection Proxy***

Problem: We wish to select one or more variants from a set of possible variants. Binding time for our selection may be as late as runtime.

Therefore:

Solution: Separate the different selections into separate classes to make the inclusion of different choices modular. These classes should be accessed in a consistent way. Either sub-class off of a common parent class, which will allow the parent class to act as a common interface; or have each of the classes implement the same Java language `interface`. In either case, instantiate one object for each choice resulting in a set of objects at runtime.

The set is hidden from the client code making it appear as a single object. The details of the accessing the collection are hidden from the client code using *Selection Proxy*. While the variants in the multi selection case are related to each other, they may not be substitutes for each other as they are in the single selection case. This may make it difficult to provide a single interface definition for all of the variants. If a single interface definition is not naturally available it may complicate both the inheritance and interface implementations.

Create a proxy object that holds and controls access to the set of variant objects. This could also be considered an example of a facade pattern; however, the interface provided in this application has an essentially one-to-one mapping with the contained variant objects, rather than providing the simplified convenience interface normally associated with a facade. The client makes a call on the proxy object which searches the collection for the appropriate variant, passes the call onto it, and returns the results back to the caller.

Consequences:

- This approach has a number of runtime inefficiencies including: The creation of multiple objects

(proxy, collection, and one for each variant). An extra call for each access, one on the proxy object, one on the variant object. The time required to search the collection for the right variant. The need to marshal and un-marshal parameters and return values from a general form for the proxy to a specific form for the variant.

- This pattern causes additional classes of runtime errors, such as not finding a requested variant in the collection.
- Additional code is required to avoid the runtime costs of the collection when only a single variant of the set is included in the product.
- Variants can be added even during runtime from either a closed set known at build time or by using reflection may add new variants after build time.

5.11 Evaluation of the Experiment

Different implementation techniques affect the core assets in different ways. Relying on Java language features - inheritance and `interface` - to implement variants resulted in more assets than other approaches. Selecting a variant implemented with inheritance or `interface` requires editing product code. XVCL allows variant choices to be made by setting the values in frame variables. Aspects install themselves, but require the core asset developer to correctly organize the code into jar files to make this work.

Can feature interactions be confined to the packaging? In the two cases studied, all of the techniques examined were able to successfully handle feature interactions in the packaging without affecting the ware code. Some techniques handled this more gracefully than others.

In Java, both pre-processing and aspects required additional tools. However, these tools are available without cost, are reasonably mature, and integrated with major development environments.

5.12 Conclusion

My goal was to provide core assets that contain enough flexibility to produce the desired product portfolio, without having to modify the assets for a particular product. The first step in reaching this goal was to consider the characteristics of variation points. Variation points were defined in terms of three concerns - cardinality, optionality and feature interactions. While it has been common to discuss variation points in

terms of cardinality and optionality, the issue of feature interaction has not typically been discussed; when it is discussed it has not been considered a characteristic of the variation point. However, by including feature interactions when considering variation points it becomes possible to account for all of the factors that differ between products in a given product line. Also, prior to this work little attention has been paid to completely removing code related to optional features when they are not selected for a product.

While there is a great need for advice on implementing variation points, most of the literature either provides a catalog of techniques at a high level that ignores language issues or presents a single technique which does not handle all aspects of a problem. In contrast, the work described here starts with the problem of implementing a variation point completely and then explains detailed examples that consider how a variety of techniques may be combined to provide a complete solution. To look at implementation in increased detail while keeping the work to a manageable size a trade off is made to limit the work to Java. Two variation points from the PPL, were selected for study:

- Scoreboard display, an example of a variation point that is option but if present requires the selection of a single value and has a feature interaction.
- Services, which is a multi-selection variation point.

For each of these examples, several coding techniques that have appeared in the literature and are available in Java were studied, specifically parameterization, inheritance, and Java language `interfaces`. In addition, applying a pre-compilation technique and a post-compilation construction technique were studied. Java by itself does not support pre- or post- compilation construction techniques, so we use additional tools that are compatible with Java.

Implementing these examples using this set of techniques allows the strengths and weakness of the various approaches in implementing each of the concerns identified in the VCM model to be analyzed. I summarized these by presenting a pattern language that is general enough to cover a variety of situations, but specific enough to provide useful guidance to developers.

Recommendations:

- Use a pre-processor such as XVCL to control the inclusion of optional features.
- Implement large scale features using inheritance or `interfaces`, the choice between them depends on the architectural context.
- Implement features requiring a small amount of code in a single place with a pre-processor.

- Implement feature interactions with aspects.

5.13 Effect of Asset Minimization on the Asset Base

Having developed a better understanding of how to implement a variation point, the effect of different implementation approaches on the asset base can be determined. The number of logical cases provides a base case which will be used to understand the effect of providing a modular approach to implementing the variation point. The focus of this section is on the number of assets that need to be stored in the asset base. Single and multi valued variation points require somewhat different calculations for the number of assets needed to implement the variation point.

5.13.1 Calculations for a Single Value Variation Point

To calculate the number of logical cases for the variation point:

1. Count the number of variants allowed.
2. If the feature is optional, add one for the case where it is omitted.
3. Count the number of interacting features.
4. Apply Formula 5.1

Each combination of interactions may be used with each variant. Also, the interacting features may be omitted. Taking a case similar to the PPL scoreboard, optional feature with three variants - V1, V2, V3, but with two feature interactions - I1, I2, the following sets are generated: omitted, V1, V1I1, V1I2, V1I1I2, V2, V2I1, V2I2, V2I1I2, V3, V3I1, V3I2, V3I1I2. Formula 5.1 shows the general case. If each logical case is coded as a module, than the number of modules will go up exponentially, based on the number of feature interactions. It should be noted that this assumes that all possible combinations are actually needed by the SPL.

$$\begin{aligned} lc &= \text{number of logical cases} \\ v &= \text{number of variants} \\ i &= \text{number of feature interactions} \\ lc &= 1 + v + v^i \qquad (5.1) \end{aligned}$$

If the code that handles the feature interaction is kept separate from the code that provides the feature variant then fewer modules are required. This case starts again with the number of variants and adds one for the case of an omitted feature. However, it is now assumed that the code for each feature interaction is in a module that will work with any of the variants and any of the other feature interactions. This generalizes into Formula 5.2. This assumption allows a reduction in the the number of implementation modules from 13, assuming that all combinations are actually developed, to 6.

$$\begin{aligned} ic &= \text{implemented cases} \\ v &= \text{number of variants} \\ i &= \text{number of feature interactions} \\ ic &= 1 + v + i \end{aligned} \tag{5.2}$$

5.13.2 Calculations for a Multi Value Variation Point

Find the number of logical cases for the variation point, beginning with the number of variants allowed. Any number of variants may be chosen in any combination. Also, the feature may be omitted. Then consider feature interactions, each combination of interactions may be used with each set of variant. Also, the interacting features may be omitted. This is generalized to Formula 5.3.

$$\begin{aligned} lc &= \text{number of logical cases} \\ va &= \text{number of variants available} \\ i &= \text{number of feature interactions} \\ lc &= 1 + 2^v a + (2^v a)^i \end{aligned} \tag{5.3}$$

Taking the above example of three variants (V1, V2, V3) and two feature interactions (F1, F2), but constructing the multi selection case, the number of logical cases (lc) is 73. Note that this is the number of logically possible cases. For a particular SPL it may be determined that some cases will not actually be needed and development on them may be avoided.

If the variants are implemented as modules which can be combined to get the combinations that are desired for a particular product and if the code for each feature interaction is in a module that will work with any of the variants and any of the other feature interactions, then the number of modules is shown by

Formula 5.2. This reduces the number of implementations needed in this example from 73, assuming that all combinations are actually developed, to 6.

5.13.3 Conclusions

These examples show that breaking a variation point into its constituent parts, based on cardinality and feature interactions, allows the number of assets needed to implement the product line to be reduced. This is particularly true in the multi selection case. Even in this small example, three variations and two feature interactions, implementing the variation point in a modular way reduces by dozens the assets needed.

In the various implementation techniques described there was often some overhead involved. For example, implementing the multi selection case with inheritance requires a proxy class and a Java language interface file. However, these assets occur a fixed number of times for a particular technique and do not increase as the number of variants or feature interactions increase. As a result, the main point that the number assets may be reduced continues to hold true.

Chapter 6

Organize Assets

6.1 Background

In this chapter[1] I consider the activity of assembling pre-existing components into the different product variants supported by the product line, which I will refer to as deriving products. Other assets, such as requirements, schedules, etc. are also stored in the asset base; however, these are mostly used for activities other than deriving products. The focus will be on how the asset base supports this product derivation by providing the components needed for assembly and how the organization of the asset base effects the providing of parts.

What interests us in the derivation process is the problem of selecting components from the asset base to produce a product, thus reusing the asset. However, selecting a component is not a simple process. In order to successfully reuse a component the developer must find, select, understand, and adapt the component [Dusink and van Katwijk 1995].

The process of finding may produce multiple components that will be candidates for the selection step; at the same time, the find step may miss some components that should be considered for selection. The selection step chooses one of the components found, which will be used in the further steps. There are several results of the find step which will cause problems for the selection step. If the find step produces no candidates then selection is not possible. Alternatively, the find step may produce so many candidates that the user is cognitively overload and his difficulty choosing from among the candidates in the selection step.

In deriving a product supporting assets as well as components need to be selected. However, many supporting assets that assist in building a product, such as an instantiated feature diagram, are singletons assets which logically exist only once per product. Others, such as an attached process, come along as part of other selections. In contrast a product line typically has a large number of product components which if presented directly to the user without an effective find step to reduce the number of items presented will be difficult for the user to select from. Therefore, my focus will be on component selection.

[1]Content in this section previously published in [Hunt 2006]

6.1.1 The Effect of Size

The point at which a group has become too large for efficient selections is a human performance issue, which makes precision difficult. An important factor is how familiar the user is with a particular collection. Work on selecting from menus would seem to be basically applicable to our concern of selecting components. Menu collections are example of applying trees to a selection problem. In Landauer and Nachbar [Landauer and Nachbar 1985] only menus of up to 16 items were studied because of the fall off in performance. Landauer and Nachbar also recommended broad shallow trees over narrow deep trees for selections, and cited a number of other researchers who had reached the same conclusion. Most fundamental in this area is the Hick-Hyman Law [Hick 1952][Hyman 1953], which suggests a linear relationship between the items considered and the time it takes to select one. Humans have a finite bandwidth to access information and a finite time span in which they remember the information, these factors limit the ability to work with lists. As a result, we will assume that group size should be limited to 20. Since this is several magnitudes smaller than the of number of components we expect to have in a product line, it would seem cognitively imperative that the asset base be arranged into smaller groups.

6.1.2 Hierarchical Organization

Given a collection of items from which a selection will be made the most elementary presentation is a list. This simple approach skips the find step to narrow the possibilities and confronts the user with the entire collection. As the list becomes longer it becomes difficult to select an item from the list. Ordering the list by some criteria, perhaps alphabetically or by date, can help, but only to a point. While not a product line, Sun's Java library provides an example of the problem of finding a desired component. On Sun's Java API web page the main access mechanism is a scrolling box that lists the class and interface names alphabetically. As the libraries have evolved, the list has become longer, reaching a point where it is difficult to find an item, even if its name is known. In early Java releases the list was short enough to be browsed with the user examining names that looked like plausible choices. The longer lists make this sort of browsing increasingly impractical.

The alphabetical ordering of class names is often not helpful in finding the asset that we need. Consider the task of selecting a growable collection that allows the program to control the order of the items held, both Vector and ArrayList are candidates, but also we should consider using the List interface. Looking at the List interface shows that there are many other implementations that support this type of behavior - LinkedList,

RoleList, etc. The alphabetized list does not help us to understand that these choices have similar underlying functionality. A presentation that groups classes with similar functionality together, the find step, would provide a better group from which to make a selection.

The Java library web interface also allows a particular package to be selected. This can be considered as a find step which breaks the collections into more manageable pieces prior to the selection step. Unfortunately, the groups formed by packages are in many cases not particularly intuitive. The Javadoc mechanism which generates the web pages ties the presentation to the user with the structure of the code. Even with this limitation there is room for improvement. It appears that a number of package choices do not take into account the problems a user might have in selecting a component. Also, only a single level of grouping is provided and the relationship between packages is not specified. Specifying a relationship may be difficult due to the very general nature of the Java libraries. What is a useful finding relationship that ranges from XML parsers, graphics, and I/O?

A list may be improved upon by breaking the list into groups and expressing some relationship between groups, creating a hierarchy. Using a hierarchy allows the task of selecting an item to be broken into a series of smaller and simpler choices. The particular hierarchy formed is a result of the criteria we chose to decompose the collection into groups. The importance of hierarchy on the ability to find components is noted in the A7E Avionics work by Parnas et al. [Parnas et al. 1984]. One of the artifacts developed in this work was a “Module Guide”. The guide was deliberately arranged in a hierarchical manner to allow relevant modules to be identified without studying irrelevant modules. The focus of this work was to allow code maintainers to find modules; however, it was later applied to product derivation in a SPL context [Ardis et al. 2000].

Conversely, the lack of hierarchy has been noted as a problem in cases studies on product derivation in industry: “Variation points not organized hierarchically. One of the causes to the first problem [Unmanageable number of variation points and variants] is the fact that neither product family explicitly organized variation points. As a consequence, during the product derivation, software engineers are required to deal with many variation points that are either not at all relevant for the product that is currently being derived, or that refer to the selection of the same higher level variant.” [Deelstra et al. 2005] This paper by Deelstra et al. refers to variation points, their own area of research; however, when describing the actual selection process they make clear that it is components that are being selected.

In this research a hierarchy of groups will be used to guide the find / select process. The user begins at the root of the hierarchy. The children of the root constitute the first group of candidates found and offered to the user for selection. The user selects one of these to proceed to the next group, which acts as the new set

of candidates for selection. The process proceeds recursively until a particular asset is selected for use or it is established that no suitable asset is in the collection. The question is how to make use of this tree structure to provide the most benefit in selecting parts for product derivation.

6.2 Experimental Design

The experiment reported here compares three approaches to organizing the components in an asset base. We first present the criteria we used to evaluate the result of applying each approach, then describe the approaches and finally describe the context in which the experiment was conducted. We will arrange the components found in the PPL into a particular hierarchy of groups. A successful hierarchical decomposition will allow us to find the right component by making a series of decisions on groups of a manageable size. What we will attempt to determine is which organization criteria will produce a hierarchy that allows easier product derivation.

6.2.1 Evaluating Component Organization Approaches

Our goal in organizing the asset base for product derivation is to make assets *easy to find* by users to fulfill requests to build particular products. Easy to find is hard to measure directly. I will use a set of four criteria as a proxy for easy to find. These criteria which will be used to evaluate the asset base organization are:

- Natural division
- Easy to map
- Reasonably sized groups
- Similarly sized groups

Natural division - There is a place for everything and everything is in its place. The organization approach being evaluated should select a single group for each of the components. An approach that selects multiple categories creates ambiguity about where to place and find the component. A good natural division should make it easy to place a component into the correct group. I was able to quantify natural division by examining the number of misfits. By considering the problem in terms of a component either fitting or not fitting into a group, we have a two value answer, which can be considered as a binomial. Provided there are enough trials a normal distribution can be assumed. In this case each attempt to place an asset into the organization

is considered a trial. As long as there are more than 32 assets a normal distribution can be assumed. To determine the statistical significance of the number of components that do not fit into groups we use a Z-test. Statistical results are found in Section 6.4.

Easy to map - At some point in the product derivation process the user's description or request for a particular product must be mapped into a particular set of components, in order for the product to be produced. What is the cognitive distance between the user's description of the product requested and the terms used in the asset base organization? One way to measure this distance is to determine the degree of overlap in the words or terms used to describe the product when the user requests it and the words or terms used to traverse the organization's hierarchy. This requires knowing specifically how the customer specifies the product, which is not available for the PPL. Not having a quantified measure of distance I must resort to somewhat subjective discussion of this distance for each of the organizations evaluated. An additional consideration is whether the relationship between groups is the same for the entire tree or changes as the tree is traversed.

Reasonably sized groups - Each group should be of a size that is humanly manageable to facilitate selecting the item of interest from the group. Human factors research described in Section 6.1.1 suggests this to be approximately 20 items. Reasonably sized groups mean that the result of a find step is cognitively manageable for the user. Organizations which allow an arbitrary number of levels have a greater ability to keep groups to a reasonable size. One way to achieve this is to find an approach that can be applied recursively as the collection is decomposed into a hierarchy. A broad tree is better than a deep tree for selection purposes [Landauer and Nachbar 1985]; therefore a criteria that provides more branches should work better while keeping the size manageable.

Similarly sized groups - Each group should be approximately the same size. According to information theory, similar sized groups maximize the average amount of information provided by each choice [Shannon 1948]. In working our way through a tree to find a component we will need to make multiple decisions. Similar sized groups will minimize the number of choices that need to be made in the average case over time. The ability of an organization scheme to allow levels to be added to only one part of the tree helps to keep groups to a similar size; while also avoiding the need to reorganize the top portions of the tree. To measure if groups are similar in size, I count the number of assets in each group, excluding the groups of misfits, and compare the counts statistically. Counting should be analyzed by using non-parametric statistics. For this criteria I test a hypothesis that the groups are equal in size using a Chi-squared test.

6.2.2 Considerations in Organizing Components

The product developer may not manipulate the components directly, but may use some interface which represents the component. For example, a component might be represented as an icon in a GUI. These are the items we will organize, those with which the user interacts to select a component. An index or view onto the components rather than rearrange the actual components might be used.

All of the approaches we consider are essentially static; we wish to create at least the top several levels of the organization and then leave them for the duration of the project. I assume that frequent and large reorganizations of the components will be confusing to those trying to find them; therefore, the organization trees are not re-balanced after they are established.

Ideally each of the components can be assigned to a single group; however, during our research we identified three circumstances that would frustrate this goal:

- Needs Splitting - the component does more than one thing and lacks proper cohesion. As a result it may belong to multiple groups, depending on which part of its functionality is considered. This is an indication that the component should be refactored.
- Indeterminate - the component does not fit into any of the available groups.
- Cross-cutting - The component is cohesive, but fits into more than one group. This occurs when more than one group is responsible for the same functionality.

There are several ways components may be ordered with in a group. Frequent choices for ordering are: alphabetical by name, date created or modified, version number, or frequency of access. Components within different groups may be ordered differently. Assuming that the groups are organized in a manner that makes them reasonably and similarly sized most of the work of selecting the right component is done by the time we arrive at a particular group; therefore, we do not investigate the issue of organization within a single group further.

Our focus is evaluating each of the proposed organizations for the purpose of product derivation. It is possible that the needs of different stakeholders require different logical views onto a single set of components. Providing multiple views, based on different criteria, could be useful to accommodate these differences. We evaluate each organization individually and without switching between views. This provides guidance for organizing the asset base in those situations where only one view is supported. In situations where tool support is available for multiple views our work should provide guidance on which views are most likely to be useful.

We focus on the task of selecting components based on the the product description provided by a customer. There are other tasks that might be performed by the product developer. For example, resolving a dependency relationship, where including one component into a product requires the inclusion of another. For this task we would want easy access to the dependent assets. In architectural modeling there is no single view that is adequate for all uses of the architecture, instead multiple views are needed. This seems to be true for using the asset base as well.

We will use a simple notation to present organization trees graphically. This notation uses circles represent groups of components and arrows to represent possible traversal paths. The paths represent a relationship between groups based on organization scheme represented in the diagram. Different organizations define different types of relationships between groups. To find a particular asset the user begins at the root and decides which path to follow, level by level.

Having attempted to be clear about our presuppositions, we present three possible approaches to organize components for the purposes of product derivation: Key Domain Abstractions, Architecture, and Features.

6.2.3 Organization Approaches

All organization approaches examined are "generally applicable"; that is they do not rely on characteristics of a particular problem domain. This allows the research to be used in a variety of places. If a particular problem domain is of interest a domain specific organization could be proposed and the evaluation criteria developed in these research used to test its effectiveness.

Key Domain Abstraction Organization

The first approach we studied is to use key domain abstractions; which has been recommended for organizing software libraries [McGregor and Sykes 1992]. While libraries differ in key ways from an SPL asset base they share a number of characteristics. Libraries provide parts that are intended to be composed into products. The user of the library is typically different from the developer and relies on the organization of the library to help locate the desired component. The typical SPL process includes a domain analysis activity, which should provide enough domain information to apply this approach.

The PPL's problem domain is arcade games. The key domain abstractions identified are: game, board, sprites, and scoring. The game encapsulates both the type of game (brickles, pong, etc.) and also the game session. The board is both the logical playing field and holds the collection of the game elements that the

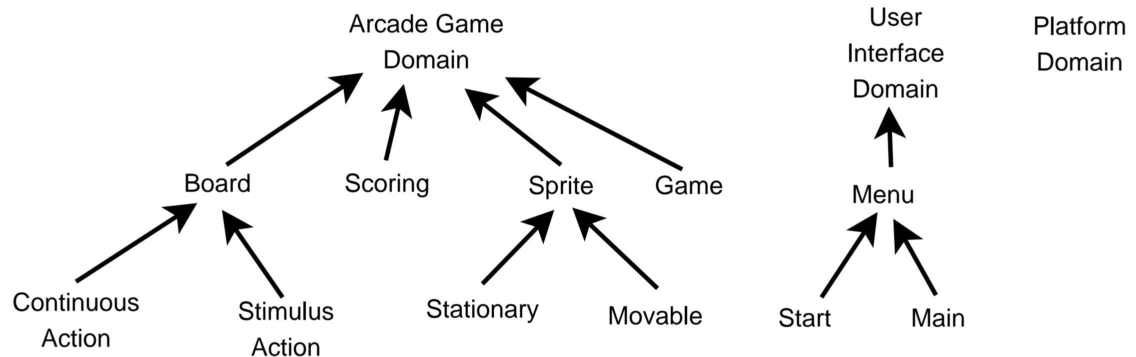

Figure 6.1: Key Domain Abstractions

player interacts with. The sprites are the logical game elements. The abstraction provided by the sprite allows us to re-use most of the code for layout, moving, and collision handling between games. In addition we have two supporting solution domains - user interface domain, and the platform domain. Figure 6.1 shows the types of objects identified by our domain analysis.

We begin our organization by creating a group under the root for each problem domain and each solution domain that we have identified. We must then identify the top level abstractions in each domain. Domain modeling makes use of both IS-A and HAS-A relationships. If we use both of these relationships in forming groups then selecting a group is ambiguous. Does moving into a group under game give us a particular game, such as brickles, or does it give us a component for a game, such as a scoreboard? Many components will place equally well in the group that considers it as a sub-component or the group that considers it as a sub-type. Also, we are likely to create cycles, which will add complexity to the organization. To avoid these problems, we choose one of the relationships to use for organizing components. Which should be used as the basis for the organization will depend as the basis for the organization on the domain and the approach emphasized by the domain analysis. OO analysis and subsequent OO development is distinguished from other approaches by finding and exploiting IS-A relationships, making it likely that IS-A relationships will dominate in cases where OO analysis is used. Also, the product developer is more likely to have a general idea of the type of part needed and is looking up the specific choices. This is a better fit for an IS-A relationship. Whichever type of relationship is primary in the particular domain analysis provided as input should be used for the organization.

Having selected one of the relationships we can identify the top level abstractions. In the case of the PPL, a thoroughly OO approach, we use the IS-A relationship for our organization and use the top level

abstractions from the domain analysis of the Game Domain which are: Board, Scoring, Sprite, and Game. To draw more detailed distinctions, we will subclass the top level abstractions and continue for as many levels as necessary. For example, within the group *Sprite* we could have the groups *Stationary* and *Movable*. How much detail to use is a purely pragmatic decision. We continue to further divide the organization tree until the number of items in a group becomes cognitively manageable, that is less than 20 items per group.

Architectural Organization

The second approach we studied for organizing components is to use the software architecture of the product line. Simply understanding the problem domain does not provide sufficient information to successfully structure a program. In current software engineering practice the software architecture acts as a bridge between the problem domain and the solution domain, provided by the computing platform [Coplien 1999]. To organize the components we map the architecture onto a tree. This is done routinely for single product projects, often without much reflection, when files are stored into file directories. In SPL work we develop an architecture to be used by all of the products in the product line. While there are some differences between the product line architecture and the architecture for a single product, there is sufficient similarity that we can adapt this approach.

There are some issues in using the architecture to organize components. The same architecture may be applied to product lines that vary greatly in size, but provide the same number of groups at the top level; which may result in large groups. Many architectures are not designed to be applied recursively, as a result the number of organizing levels that they provide guidance for is fixed independently of the problem. This is true of the MVC architecture used by in the PPL. Some architectures can be applied recursively. For example, a filter in a pipe and filter architecture might be constructed out of other pipes and filters. Even here we will want to be able to use all of the filters independently. Hence all of the filters will be placed at the same level of the hierarchy. This will result in a very shallow and very bushy tree, which is fine unless this results in too many branches to be understood easily. At this point we will need some other criteria to group our filters such as filter purpose, but now we are no longer using a pure architectural approach. A layered architecture may in fact work recursively with large layers being divided into sub-layers. An organization that can be applied recursive is more likely to be able to divide a set of components into reasonably sized groups.

Another issue is that architecture exists to provide guidance independent of a particular problem, since it avoids the particulars of a given problem it fails to provide guidance for organizing domain related components. For example, the MVC architecture employed in the PPL does not supply advice on structuring

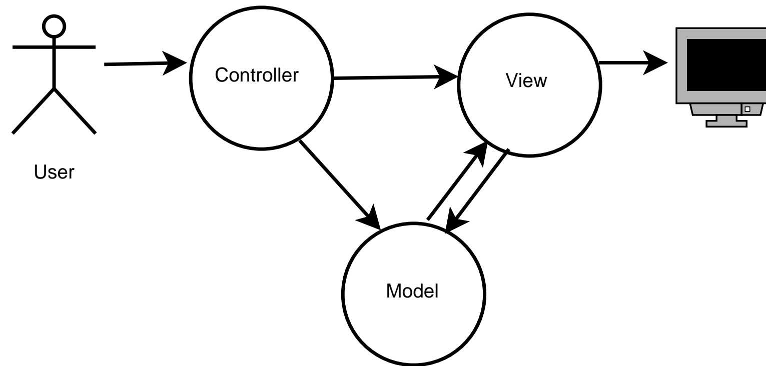

Figure 6.2: Model - View - Controller Architecture

the model; this makes it more general in its application, but does not help in organizing the largest group of components[2]. If we begin incorporating elements specific to a problem, in our case arcade game elements, it ceases to be useful for problems that do not involve arcade games. It becomes a high level design rather than an architecture.

As shown in Figure 6.2, the architecture provides structure for set of components and defines the services they provide each other. In this case the service relationships form a graph with a cycle rather than a tree, a result not uncommon to many architectures. The architecture used provides us with a set of top level components. In addition, an architecture for one or more of the components may be provided. For example, a client - server architecture could specify the existence of an authentication server as a component of the server. In decomposing the top level components either Is-A or Has-A relationships could be used.

The essence of this organization is to provide a logical view of the architecture. We provide a group for each architectural component, starting at the top level of the architecture. We continue this process until all of the product line assets have been placed in groups of reasonable size or we run out of architectural information. In this example, the main components of the architecture - model, view, and controller - form the groups under the root we will use for organizing our components. In this case none of the components is decomposed further by the architecture.

[2]It is certainly possible to provide more than one level to an architecture. For example, a top level might be a 3-tier architecture and the front tier could use an MVC architecture. However, unless we want to make *architecture* a synonym for *design* there are going to be certain issues that the architecture address and others that it does not address. If the areas that it does not address have implementation artifacts then it is not going to provide guidance on how to organize them

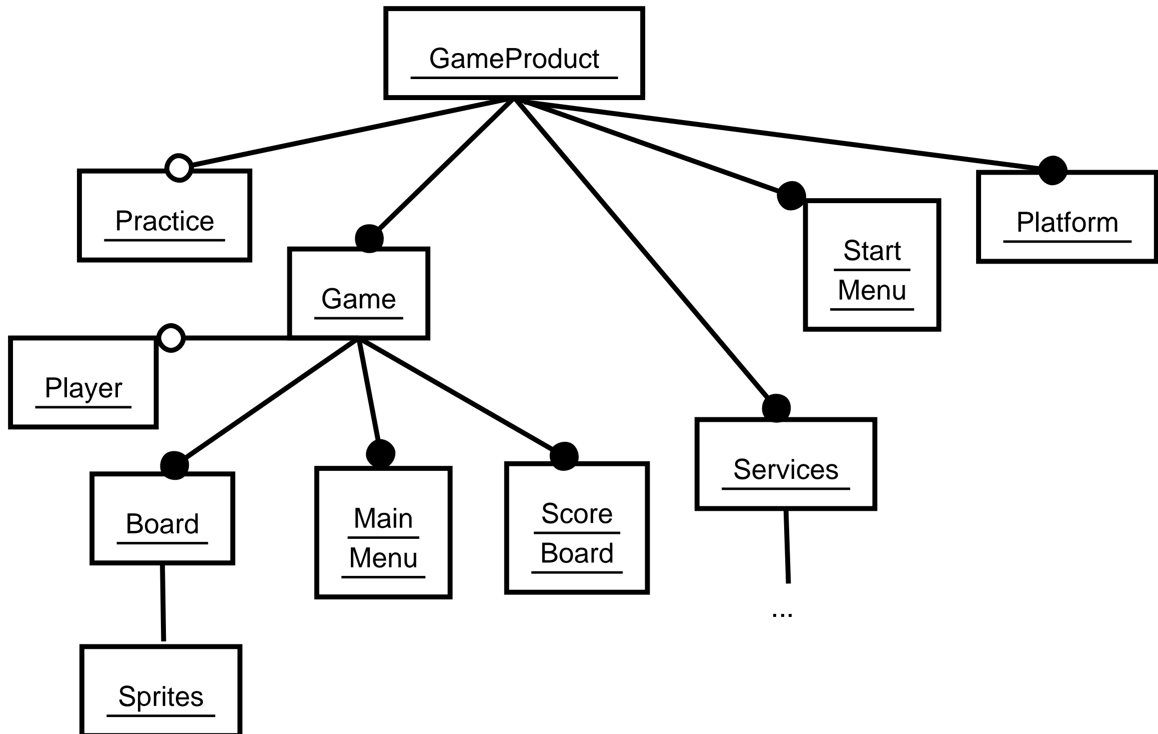

Figure 6.3: Top Level Feature Diagram

Feature Based Organization

A final approach to organizing the asset base is to use features. “Features are user-visible aspects or characteristics of a system and are organized into a tree of And/Or nodes to identify the commonalities and variabilities within the system.”[Clements and Northrop 2002] Features have played an increasing role in SPL since they were introduced for domain analysis [Kang et al. 1990]. Features are identified very early in the development process, making them available to organize early assets. Most current work on identifying and implementing variations is based on features [Svahnberg et al. 2005]. The feature diagram has become a key tool in developing the SPL architecture. Many development methodologies recommend attempting to design components such that they implement a single feature [Atkinson et al. 2001].

Using features gives us the flexibility to take into account both the problem and the solution domains. Guidance for identifying features and their relationships may be found in the literature starting with [Kang et al. 1990]. The results of feature modeling are commonly represented as a feature diagram, which depicts a tree. A feature diagram for the PPL product line is shown in Figure 6.3.

We will use a feature diagram for developing our organization tree based on features. Since the feature

Table 6.1: Components Sorted based on Key Domain Abstractions

Groups	Count	Percent
Sprite	15	28.3019
Board	10	18.8679
Game	3	05.6604
Scoring	9	16.9811
User Interface	8	15.0943
Platform	7	13.2075
Cross-cutting	1	01.8868
Total	53	100

diagram is in the form of a tree it is straightforward to convert it into an organization tree. The root of the feature tree is used as the root of our organization tree. The features on the feature diagram become groups in the organization tree, and retain the same parent / child relationships that they did in feature diagram. We will need to store components for an optional feature regardless of whether a particular product uses them. So for organizational purposes required and optional features are handled the same way. Our decision of how far to go down the tree remains a pragmatic one, based on getting reasonable sized groups. We expect that everything needed for a product will be characterized by a feature; therefore, we do not expect to have indeterminate assets when using a feature based organization. We may still have cross-cutting; however, we can often encapsulate a particular type of cross-cutting as its own feature, which will allow us to organize all of the assets involved into the same group. In the PPL, Practice Mode provides an example of cross-cutting and its encapsulation into a feature, which will be discussed at length.

6.3 Experimental Results

6.3.1 Key Domain Abstraction Organization

Figure 6.4 shows the organization tree that results from using key domain abstractions as the organizing criteria. Circles represent groups. Arrows indicate hierarchical relationships between groups needed to form the tree. The result of sorting the components into the organizational tree are shown in Table 6.1. Notice there is a component that does not fit into any of the domain categories because it is cross-cutting. This component is a singleton which holds the state indicating if a game is in practice mode. This state is set in the

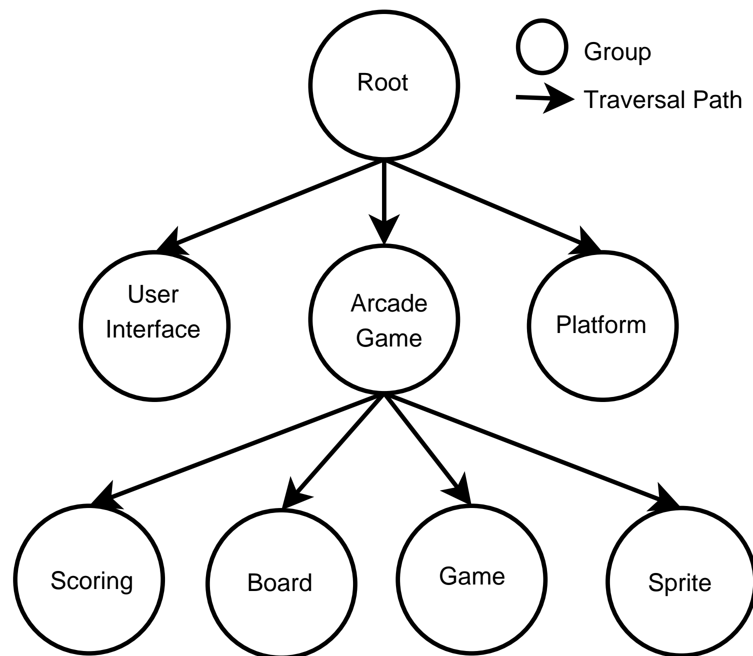

Figure 6.4: Organization Tree for Key Domain Abstraction based organization

user interface, via a menu. The state is read in the sprite KDA and the scoring KDA. This approach succeeds in dividing the components into manageable groups, sprites being the largest group with 15 components or about 28% of the components. We could attempt to break sprites into smaller groups and in this case there is a logical domain abstraction for doing so, the difference between movable and stationary sprites. Since the number of components in the sprite group is under twenty we will leave this group alone.

Assessment of Key Domain Abstraction Organization

- Natural division - Succeeds, while this organization produces a misfit this is not statistically significant. This technique works best if the component development relies on the same domain abstractions. Components acquired from third parties may fit well into this organizational approach. Not all components fit into the groups provided.

- Easy to map - Open, it is an open question how directly concepts from the problem and solution domains related to product descriptions. Much of the terminology for the domain will be familiar to the customer and may be used. However, the terminology may be general enough to apply to all the

products in the product line rather than differentiating between them. Also, the terminology used in the related tool domains (GUI and Platform) may not be used in product descriptions. There is a need to examine how the customer would specify a particular product to answer the questions. The assets to do this are not available for the PPL.

- Reasonably sized groups - Succeeds, none of the groups exceeds the limit of 20 items per group. Most complicated domain abstractions can be broken into more primitive domain abstractions. Approach does not impose a limit on the number of levels used.

- Similarly sized groups - Succeeds, differences in size between groups is not statistically significant. Most complicated domain abstractions can be broken into more primitive domain abstractions. This can be repeated until the desired group size is reached.

6.3.2 Architectural Organization

The PPL uses a Model-View-Controller based architecture. The groups based on this architecture are: model, view, and controller. Figure 6.5 shows the organization tree that results from using the architecture as the organizing criteria. Circles represent groups. Arrows indicate hierarchal relationship between groups.

The results of sorting the components are shown in Table 6.2. The number of components in the Model group is large both in terms of the percentage of components and in the absolute number of components grouped together. We would like to split Model further by adding another layer to our organization, but the architecture provided for PPL does not describe further detail for these components. We continue to have a cross-cutting component, the singleton which controls whether a game is in practice mode. This is set in the controller, via a menu and read in View and Model. We also have four components that do not fit into any of the groups. All of these involve interfacing the program with the platform.

Assessment of Architecture Organization

- Natural division - Fails, this organization produces a statistically significant number of misfits between cross-cutting and indeterminate categories of misfits. Not all components needed to build a product are related to architectural features.

- Easy to map - Fails, it is unlikely that a customer will understand the architectural terms - model, view, controller, etc. Therefore, the product developer will require an understanding of the mapping between

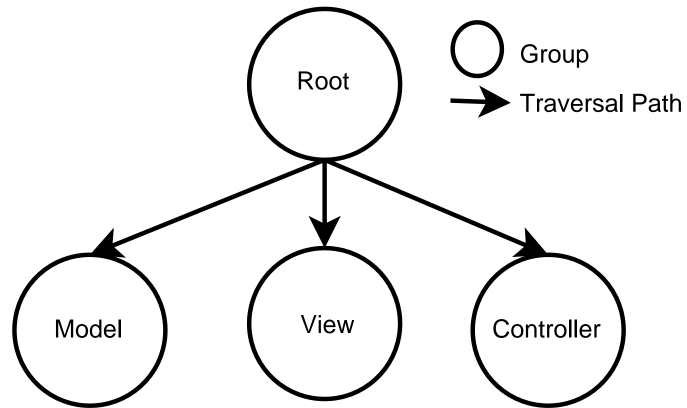

Figure 6.5: Organization Tree for Architecture based organization

Table 6.2: Components Sorted based on Architecture

Groups	Count	Percent
Model	32	60.3774
View	8	15.0943
Controller	8	15.0943
Cross-cutting	1	01.8868
Indeterminate	4	07.5472
Total	53	100

an architecture and the problem domain. For example, menus are a type of controller. This requires a more complex cognitive mapping than the other approaches.

- Reasonably sized groups - Fails, the model group exceeds the limit of 20 items per group. This organization does not provide guidance for more then one level of groups given the MVC architecture used. Number of groups does not change with problem complexity, number of products, or number of features.
- Similarly sized groups - Fails, the difference in size between these groups is statistically significant. The groups vary widely in size, most of the code in this example belongs to the model. This is not unexpected for MVC architectures.

6.3.3 Feature Based Organization

The features of the PPL are divided into menus, platform, game, services, product, and practice mode at the top level. Game contains the groups player, board, and scoreboard. Within board is the group sprites. Figure 6.6 shows the organization tree for a feature base organization. Circles represent groups. Arrows indicate hierarchal relationship between groups.

The result of sorting the components is show in Table 6.3. When we first organized groups, Board was larger then we desired - 22 components. This was easily remedied by gong another level down on the feature model to break Sprites out into their own group. Unlike the domain abstraction and architecture approaches the feature approach fits all components into unique groups.

Assessment of Feature Organization

- Natural division - Succeeds, no misfits.
- Easy to map - Succeeds, most feature terminology originates in describing the products of interest during the scoping phase. There is likely to be a high level of connection between this terminology and that used by the customer to specify the product.
- Reasonably sized groups - Succeeds, all groups less than the limit of 20 items per group. The feature model can be traversed until the desired size is reached.
- Similarly sized groups - Fails, the difference in size between these groups is statistically significant. I had assumed that dissimilar sized groups would be caused by groups larger than average. However,

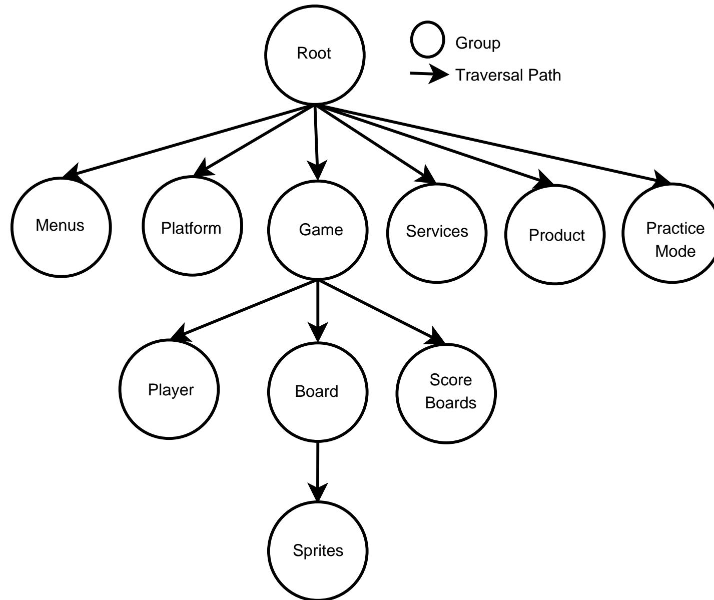

Figure 6.6: Organization Tree for Feature based organization

Table 6.3: Components Sorted based on Features

Groups	Count	Percent
Board	8	15.0943
Sprites	14	26.4151
Game	4	07.5472
Menu	6	11.3208
Platform	6	11.3208
Player	1	01.8868
Practice Mode	5	09.434
Product	2	03.7736
Scoreboard	5	09.434
Services	2	03.7736
Total	53	100

the feature model can be traversed until the desired size is reached. The problem with the feature organization related to this criteria is that it produces a number of very small groups, an unexpected result.

6.4 Discussion

It is desirable for an organization approach to find a group in the organization for every component. In the architecture approach a number of components were indeterminate; they did not fit into any of the provided groups. Both architecture and key domain abstraction were unable to select a group for cross-cutting components. Only the feature approach was able to organize all of the components we examined into groups.

Practice mode is a good example of a cross-cutting requirement that affects the product at multiple variation points, including the board, scoring, score boards, and menus. Practice mode gives the user an unlimited number of chances at the game while suspending scoring. Practice mode is an optional feature which may or may not be included in any of the games in the product line. It would be convenient for product derivation purposes to keep these components grouped together with each other and separately from other components.

Grouping components by feature puts all of the practice mode components in the same group, where they are easily found for those products which include practice mode and easily ignored for those which don't. By contrast, in the key domain abstraction organization practice mode adds components to sprite, scoring, scoreboard, menu, and leaves a cross-cutting component. In architectural organization the rule and score changes go into the model, the score board changes go into the view, menu changes go into the controller, and one component is cross-cutting. By their very nature a cross-cutting feature will not fit within any single aspect of the architecture or a particular domain abstraction. The ability of the feature organization to group components related to cross-cutting requirements together provides a clear advantage over the other approaches.

To determine the statistical significance of the number of components that do not fit into groups we use a Z-test. By considering the problem in terms of a component either fitting or not fitting into a group, we have a two value answer, which can be considered as a binomial. Given our n of 53 we can assume a normal distribution. Our hypothesis is that all components fit into some group. Running the Z-test shows that feature and key domain abstraction approaches satisfy the hypothesis, but that the architecture approach does not. The results hold for an alpha of .02, a result of high statistical significance.

We would also like to compare how well the approaches do on producing groups of equal size. As noted,

Table 6.4: Approaches compared by criteria

Approach	Natural Division	Easy to Map	Reasonable Sized Groups	Similar Sized Groups
Domain	Succeed	Open	Succeed	Succeed
Arch.	Fail	Fail	Fail	Fail
Feature	Succeed	Succeed	Succeed	Fail

information theory holds that equal sized groups result in a more efficient selection in the average case. We ignore the indeterminate and cross-cutting components for the purpose of comparing equality of group sizes. For our hypothesis we divide the number of components by the number of groups to determine an expect result of equal group sizes and then apply a Chi Square test to see how close our actual organizations come to this ideal. The key domain abstraction provides the most equal grouping followed by features, and finally architecture. The Chi Square test confirms the hypothesis of equal sized groups for the key domain abstraction, while rejecting it for both the feature and architecture approaches. The results hold for an alpha of .001, a result of very high statistical significance.

6.5 Conclusion

The growing size and complexity of the SPL assets base can make it difficult to derive products from it. As with most collections that have become difficult to use, an important step is to organize the collection to make it easier to find things. In this paper we have considered three approaches to organizing the components in an asset base:

- A key domain abstraction approach, derived from current software library practice.
- An architecture based approach, derived from current single product practice.
- A feature based approach, derived from a FODA approach to SPL development.

Table 6.4 summarizes the way the different approaches met our evaluation criteria.

- Natural division: The feature approach grouped all of the components. Key domain abstraction and architecture approaches were not able to group components related to cross-cutting features. In the case of key domain abstraction the number of components left out of groups was not statistically significant.

For the PPL the MVC architecture used was not able to group components related to interfacing the product with the platform.

- Easy to Map: The issue in question is how directly related are the groups provided by an approach to the way a customer specifies a particular product. This depends, in part, on how the customer's selections are expressed. Ultimately for a product to be chosen it must be described in terms with which the customer is comfortable. This starts the process of building a product. Features are designed to work on this customer oriented level, as a result they form groups that are aligned with the concepts that customers use. Key domain abstractions also work with concepts that are familiar to customers, but may focus on characteristics common to all products, rather than those which differentiates products. Customers do not see products in architectural terms. Therefore, grouping components by architecture requires extra mapping steps between the descriptions customers use to express the product they want built and the locations of the components needed to build it. The customer says I want a product with this kind of menu. The product developer using an architecture base organization must make the connection that a menu belongs in the controller group. People can be trained to do this, but by using other organizations we can eliminate this step from the process.

- Reasonably sized groups: Architecture produced much larger groups than the key domain abstraction or feature approaches. In the case of the PPL, the architecture approach is not able to add additional groups. Both key domain abstraction and feature approaches are able to add groups as needed to achieve reasonably sized groups.

- Similarly sized groups: Key domain abstraction did the best at providing similarly sized groups, followed by feature, and finally architecture.

While little guidance has been provided for organizing SPL components the only advice present in the literature is to use architecture. However, in our experiment, architecture was the least effective in fulfilling our criteria. This suggests that other approaches should be considered.

In our experiment feature and key domain abstraction approaches turned in broadly similar performances. The feature approach was the most successful in handling cross-cutting functionality and should work particularly well for projects using FODA for analysis. The key domain approach may be preferred in situations that use acquired components whose functionality and structure is based on the problem domain.

Chapter 7

Future Work

The motivation for managing the asset base is to reduce the cost of deriving a product. In this research, I assume that making it easier to find assets will have the effect of reducing the cost of deriving a product. That is I assume that ease of use is a proxy for cost. This is necessary because a true cost of deriving a product can only be measured in an industrial setting. A logical continuation of this research would be to apply the techniques developed in this research to an industrial software product line in a way that allowed the actual cost reduction to be measured.

7.1 Organize Assets

I have focused on the selecting components needed to meet a particular product requirement, a sub-task of deriving products. However, the asset base of a SPL contains many different types of assets used by different stakeholders to accomplish different tasks. In many cases these tasks are even less studied than product derivation. A related question is whether the feature organization is sufficient for other stakeholders and tasks. If not then we should examine which other asset base organizations and supporting tools are needed. Simply providing a complete categorization of assets and the tasks that operate on them is an open work item.

My work using the PPL as a testing ground is useful for an initial set of insights; however, it provides only a single data point. Examining the organizational issues of other product lines is an obvious next step.

In this research easy to map was not quantified. It should be possible to quantify this if the method used by a customer to request the product is available for a product line being studied. To do this the vocabulary used by the customer would be compared to the vocabulary used to describe the groups in the organization. A measure of similarity would need to be developed probably in terms of a percentage. The underlying assumption being the more similar the two vocabularies the less cognitive effort there is to move between them and thus the easier it would be to find the right assets.

7.2 Minimize Assets

I provided a variety of implementation techniques. It would be good to provide better quantitative measures of the different techniques to allow better comparisons between them. An obvious starting point for this is to calculate maintainability metrics [Oman and Hagemeister 1994] for the different approaches.

Due to space and time limitations this research was focused exclusively on a Java based development environment. Java is a mainstream OO language, with many similarities to other OO languages, particularly C++. Hence, many of the techniques presented may be applicable to other development environments. The techniques presented could be implemented in other languages with the differences noted. Obvious candidates for this include C++ and C#. To provide the same context that I have used this would first require implementing the PPL in the target language.

Chapter 8

Summary and Conclusions

8.1 Summary

I have presented research involving the management of the SPL asset base. As the size of the asset base increases the time needed to find assets, holding other factors constant, increases. Since the primary reason for constructing an asset base is to reduce the effort needed to derive products this is an important issue.

The asset base has many different stakeholders - managers, testers, etc. - who use the assets base as an input for a variety of activities - scheduling information, provision of test cases, etc. My work examines a single use - product derivation - from the viewpoint of a single stakeholder - the product developer. This in turn means that my work is largely concerned with those assets that are assembled into the product, usually referred to as components. Different uses even when they involve the same type of assets may lead to different needs in relation to the asset base. For example, the needs of a core asset developer for a bug fix may be quite different from what is needed to best support product derivation.

To pursue the issue of managing the asset base I explored two complimentary branches of investigation:

- Organize Assets
- Minimize Assets

The basic goal of *organize assets* is to arrange the asset base into logical groups so that the user is able to more easily find a particular asset. The basic goal of *minimize assets* is to remove similar assets from the asset base, reducing the number of assets that must be considered by a user attempting to make a selection. Reasonable organization schemes will place similar assets into the same group, where the user will examine them to make a selection. By reducing the similar assets that will be placed together *minimize assets* enhances the ability of *organize assets* to provide groups that are easier for the user to select from.

One further topic might be mentioned; that is what effect does the creation of modular assets needed to implement my approach to minimizing assets have on organizing assets. A topic that I include here since it does not fit entirely under either branch. The answer is that all of the new packaging assets should be kept in the same group as the ware asset they are related to. This splitting of the assets used to implement a variation

point into packaging and ware assets may cause the size of the group to expand to a point where it is larger than desirable. If this is the case an attempt should be made to extend the organization to an additional level.

8.2 Organize Assets

For this research the component assets of the pedagogical product line (PPL) were organized in accordance with three different criteria - key domain abstractions (KDA), software architecture, and features. Each of these different organizations was then evaluated based on the planned comparison criteria - natural division, easy to map components, reasonably sized groups, and similarly sized groups. Finally, the results of the comparison criteria for the different organizations were compared. The detailed results are presented in Section 6.5, particularly Table 6.4. With this completed work as a foundation I am able to address the research questions.

8.2.1 Research Questions for Organize Assets

1. Which organization is the most effective at providing the characteristics of a good organization?

2. What is the effect of feature interactions and dependencies on the organization of the asset base?

Having built a case that organizing the asset base is important, the next step is to provide guidance on how the asset base should be organized. The only, admittedly very limited, advice present in the literature on SPL organization is to use architecture. However, in my experiment, architecture was the least effective in fulfilling the proposed criteria. This suggests that other organizations should be considered. In my experiment feature and key domain abstraction criteria turned in broadly similar performances. The feature organization was the most successful in handling cross-cutting functionality and should work particularly well for projects using FODA for analysis. The key domain organization may be preferred in situations that use acquired components whose functionality and structure is based on the problem domain.

There was a statistically significant difference in the size of groups formed by the three organizational schemes. For *Similarly Sized Groups* the size of each of the groups generated by the organizing criteria was compared to an expected value of the number of assets divided by the number of groups. A Chi Squared test was then applied to determine if actual group sizes differed from the expected group size by a statistically significant amount. *Natural Division* was quantified by considering how well an organization accounted for all of the assets. For this purpose assets were considered to either fit or not fit into a group. This is a two value

determination and given the sample size available statistics based on a binomial distribution are applicable. Applying a Z-test allows us to determine if the number of misfits is statistically significant. If we have a statistically significant number of misfits then we can conclude that the organization does not provide a good natural division.

Reasonable Group Size is quantified by counting the assets in each of the groups and comparing them to the selected threshold value. The difficulty is that our threshold is based on human factors for related problems, such as menus. This does not provide a precise answer for the upper limit on the number of items that a human can cognitively manage, only an approximation. Applying results generated from measuring what happens when a user selects items from a menu to my somewhat different problem of selecting components from an asset base only increases the uncertainty that is always present when measuring human performance on cognitive tasks. The selected threshold of 20 is a reasonable one and, no doubt, the correct order of magnitude, but lacks precision.

The remaining criterion, *Easy to map*, was evaluated in only qualitative terms. Easy to map looks at how the user's description or request for a particular product is mapped into a particular set of components, in order for the product to be produced. What is the cognitive distance between the user's description of the product requested and the terms used in the asset base organization? One way to measure this distance is to determine the degree of overlap in the words or terms used to describe the product when the user requests it and the words or terms used to traverse the organization's hierarchy. This requires knowing specifically how the customer specifies the product, which is not available for the PPL. Not having a quantified measure of distance I must resort to somewhat subjective discussion of this distance for each of the organizations evaluated.

An initial concern was that the various organizations would fail to produce reasonably small groups. Indeed, this was a shortcoming when organizing the asset base using architecture as a criterion. What I failed to foresee was the problem of having an organization criterion split the assets into groups that were so small that similar sized groups were not produced. This occurred with the feature organization. What I found here were examples of what were clearly features - that is, a user visible characteristic of the system - that simply did not require many components to implement. A result of this was the failure of the feature organization to produce similarly sized groups, the only evaluation criterion that the feature organization failed to perform well on. Given the small sample size, I considered the idea that this was an anomaly. However, upon further reflection, it seems reasonable to suppose that in many cases there will be assets that are clearly features that can be readily implemented without a lot of code. This suggests that the inability to produce similar size

groups is an inherent drawback to the feature organization criterion.

Question 2 regarding feature interactions and dependencies is really worded too narrowly because our concerns are not limited to features. Underlying the concern that feature interactions and feature dependencies might effect the organization of the asset base is the fact that different characteristics of the product affect one another, and might do so in a way that would affect the organization of the asset base.

One unexpected way that this concern manifested itself was in the ways that assets failed to fit into a proposed organization. Assets produced to implement cross-cutting features also cut across architecture and domains. The feature organization successfully grouped the assets involved in cross-cutting features apart from other assets. The architecture and KDA organizations were not able to separate these assets out. Successfully handling cross-cutting features is clear advantage for the feature organization in considering the feature interactions and dependencies.

Question 2 was also included out of concern that some organizations of assets might be difficult to build. Particularly, feature based organizations which will typically have a larger number of groups than the other organizations. Also, there is less experience in using feature organizations. This did not become a problem. Feature analysis techniques, such as FODA, typically organize features into a tree. At the beginning of the research I had a concern that using features to organize assets would result in references between different parts of the organization tree, due to feature dependencies, etc., which would make it difficult to build products. While there are references to other assets in the different nodes of a tree organized by feature, they are no different in their build impact than references between assets found in other organizations. Since no additional problems were discovered in using a feature organization, no additional steps needed to be taken.

8.3 Minimize the Number of Assets

The goal of *minimize assets* is to avoid having multiple assets with the same basic functionality in the asset base. While there are a variety of reasons to avoid this situation - for example, increased maintenance costs - my motivation in addressing this issue involves the increased difficulty of asset selection during product derivation.

The first question is why assets that duplicate each others functionality, a problem which has been reported in the literature, exist in the first place. One reason reported is that a perfectly adequate asset exists in the asset base, but that the product developers were unable to find it. Hopefully, the ideas on organization presented in this research address this problem.

Another reason is that the program context at the variation point differs between products in the SPL. As a result the component is copied and modified to work in the different context. Once this occurs there are two possible ways to handle the “new” asset. We can decide that it is specific to a particular product and not put it into the asset base. In this case, should another product need the variation the product developer is unlikely to find it and the variation will have to be created again. Alternately, we can decide to put the new asset in the asset base. In this case, each time a new product is derived the product developer will have to examine both (all) of the similar assets and select from among them. Since they provide the same underlying functionality the differences may be subtle, which makes it difficult to select from among them. Since most organization criteria will place assets with the same functionality into the same group, improved organization is not likely to solve the problem of selecting between such similar assets. My goal was to include the basic functionality of the asset in the asset base only once, thereby addressing this problem.

In analyzing variation points, I found that the differences that should be expected to occur in program context could be limited to three characteristics:

- Cardinality - specifically, is a single or multiple value selected?
- Optionalness - is it an optional feature that is being implemented at this variation point?
- Feature interactions - does the behavior of this feature change as a result of another feature in the product?

When implementing a variation point the issue of binding time must also be addressed. It should be noted that each of these three characteristics may have their own binding time requirement. For example, whether a feature is included in a particular product, its optionalness, may be determined by construction time, while the particular value selected for the feature may not be decided until runtime.

The strategy I used was to divide the asset between a piece that implemented the underlying functionality, called the ware, and a piece that addressed the needed characteristics in the program context, called the packaging. The possible combinations that need to be addressed by the packaging could lead to a large number of assets. To address this, the packaging was further divided into modules that could be combined with both each other and the ware component to build a complete variation point in a modular manner.

To research the effectiveness of this strategy, two variation points, one a single-value selection and one a multi-value selection, were chosen from the PPL and implemented using a variety of programming techniques. Three programming techniques available in Java - inheritance, Java language interface, and parameterization - were used. Also, the variation points were implemented using XVCL, a general purpose frame

processing language compatible with Java, and AspectJ, an aspect oriented extension to Java. The use of XVCL allowed the exploration of some generative technique and the effects of earlier feature selections and bindings times than is achievable with Java. AspectJ allowed the exploration of later feature selections and binding time than is available with Java.

Implementing all three variation point characteristics (cardinality, optionality, and feature interactions) were attempted with the five implementation techniques. The experience gained was summarized by the creation of a variation point implementation pattern language. Having established this background I move on to the research questions related to minimizing the number of assets.

8.3.1 Research Questions for Minimize the Number of Assets

1. To what degree can component functionality be isolated from its context, via packaging?
2. Can feature interaction be controlled by packaging techniques?
3. To what extent can the approach to packaging be standardized across multiple components?

In my experiments I was highly successful at isolating the base functionality from the context with all of the techniques proposed. None of the techniques required more than one copy of the ware in the asset base. None of the techniques had awareness in the ware code of the characteristics of the program context in which it was being used. The issue that emerged was the amount of overhead added by particular packaging techniques. The amount of overhead varied considerably between techniques and the situations being addressed. None of the techniques affected product performance in a way that was noticeable to an end user playing a game.

In my experiments I was able to confine the feature interaction code to packaging modules. Furthermore, I was able to construct packaging modules containing only the code for a particular feature interaction in such a way that a particular feature interaction could be included or excluded from the variation point without affecting the coding of other packaging modules. Thus, for the cases studied, feature interaction can be controlled by packaging techniques in a modular way.

Feature interaction via packaging was implemented using a variety of techniques. I succeeded in separating the feature interaction code from the ware using all of the proposed techniques. However, most techniques required the client code to reference the module providing the feature interaction in order for the feature interaction to be included into the product. In all of the straight Java implementation this required instantiating

an object for each of the feature interactions to be included in the product. In the XVCL implementations this involved specifying a frame containing the feature interaction code. However, AspectJ was able install the needed feature interaction code without modifying the client code. This allowed the code related to various interacting features to be more decoupled from each other and may be a considerable advantage.

I was also able to standardize the approach to packaging the various characteristics needed to implement a variation point. This standardized approach is to provide guidance in the form of a pattern language as found in Section 5.10.

A related question is: To what extent can this standardized approach be embedded into implementation artifacts that can be reused for multiple variation points? This depends on the implementation techniques used. The main issue to be overcome is that different variation points will most naturally use different signatures. XVCL is able to manipulate portions of the code independently of signatures and could be used to provide standard templates that could be used for multiple variation points. For Java, the Selection Proxy pattern presented could be abstracted and used for multiple variation points, although the pattern has a number of drawbacks. Details of such reusable implementation artifacts can be found in Section 5.8.

8.4 Conclusions

The ability to find the needed assets to derive a product can be improved by managing the assets base. The SPL approach to software engineering provides many early process artifacts which can be leveraged to provide improved management of the asset base. To show the feasibility of improved management I have proposed two complimentary investigations:

- Organize Assets
- Minimize the Number of Assets

In my research, I found that the highly standardized program context provided by SPL reduces the variation in program context to just a few characteristics, such as cardinality. This in turn allowed me to demonstrate that the assets functionality could be provided using a ”ware” module, which could then be adapted to the allowed program contexts using related packaging modules. This allowed a variation point to be implemented by assembling packaging and ware modules rather than by the copying and modifying approach that has been identified as current practice. My research is the first place to apply the ware and packaging approach to providing SPL variation points.

Managing the asset base makes it easier to find the assets needed for product derivation. My research successfully demonstrated that better organization and fewer assets are an achievable goal.

Bibliography

A, C. 1964. *Notes on the synthesis of form.* Cambridge, Harvard University Press.

A, M. 2004. Personalized cost-efficient product line implementation. Tech. Rep. IESE-Report 056.04/E, Franhofer Institute.

A, M. G, C. 2001. Implementing product line variabilities. In *Symposium on Software Reusability*. 109 –117.

A, G., D, M., J, R., P, J. S., W, A. 1997. Reuse research and development: is it on the right track? In *Symposium on Software Reusability*, M. Z. (moderator), Ed. 212 – 216. Panel Discussion.

A, M., D, N., H, D., S, H., W, D. 2000. Software product lines: a case study. *Software Practice and Experience 30,* 7 (June), 825–847.

A, T., M, T., S, T. 2004. Using a configurator for modelling and configuring software product lines based on feature models. In *Software Variability Management for Product Derivation - Towards Tool Support at International Workshop of SPLC.*

A, C., B, J., B, C., K, E., L, O., L, R., M, D., P, B., W, J., Z, J. 2001. *Component-Based Product-Line Engineering with UML.* Addison-Wesley.

B, L., H, J., H, P. 2000. *Software Architectures: Advances and Applications.* Springer.

B, D., S, V., S, M. 1993. Scalable software libraries. In *1st ACM SIGSOFT symposium on Foundations of software engineering*. Los Angeles, 191–199.

B, A. 1998. An internet-base information system for cooperative software reuse. In *International Conference on Software Reuse*. IEEE, 236–245.

B, L. M. 1990. When objects collide experiences with using multiple class hierarchies. In *OOPSLA*. 181–193.

B, R., M, A., N, J. 2003. No name: just notes on software reuse. In *ACM SIGPLAN Notices*. Vol. 38. ACM Press New York, NY, USA, 76 – 96. COLUMN: OOPSLA onward! track.

B S, T. 1994. The library scaling problem and the limits of concrete component reuse. In *Third International Conference on Software Reuse*. IEEE Computer Society, 102109.

B, J. 2001. *Effective Java - Programming Language Guide*. Addison-Wesley, Boston.

B, J. 1999. Product-line architectures in industry: a case study. In *International Conference on Software Engineering*. 544–554.

B, J. 2000. *Design and Use of Software Architectures: Adopting and evolving a product-line approach.* Addison-Wesley.

B, J., F, G. G, D., K, J., O, H., P, K. 2001. Variability issues in software product lines. In *Proceedings of the Fourth International Workshop on Product Family Engineering (PFE-4)*. 1119.

B, G., D, P., K, W. 1994. *Product Design for Manufacture and Assembly*. Marcel Dekke, Inc, New York.

B , B. D , B. 2001. Real-time convergence of ada and java. In *SIGAda '01: Proceedings of the 2001 annual ACM SIGAda international conference on Ada*. ACM Press, New York, NY, USA, 11–26.

B , T. J., S , I., K , P., C , D. 2002. Adaptable components for software product line engineering. In *Software Product Line Conference*, G. J. C. ed., Ed. LNCS, vol. 2379. Springer, 154–175.

B , S. V. M , J. W. 1997. Reuse library interoperability and the world wide web. In *International Conference on Software Engineering (ICSE) Joint session Symposium on Software Reuse (SSR)*. ACM Press, 684–691.

C , J. 1993. Software packaging. Ph.D. thesis, University of Maryland.

C , G. M G , J. D. 2002. Guidelines for developing a product line production plan. Tech. Rep. CMU/SEI-2002-TR-006, Carnegie Mellon University, Software Engineering Institute.

C , P. N , L. 2002. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston.

C , J. O. 1999. *Multi-Paradigm Design for C++*. Addison-Wesley.

C , K. E , U. W. 2000. *Generative Programming Methods, Tools, and Applications*. Addison-Wesley.

C , K., H , S., E , U. 2005. Formalizing cardinality-based feature models and their specialization. *Software Process Improvement and Practice 10,* 1, 7–29.

D , S., S , M., B , J. 2004. Experiences in software product families: Problems and issues during product derivation. In *Software Product Lines: Third International Conference, SPLC*, R. L. Nord, Ed. Number 3154 in LNCS. Springer-Verlag, 165–182.

D , S., S , M., B , J. 2005. Product derivation in software product families: a case study. *Journal of Systems and Software 74,* 2 (January), 173–194.

D , R. 1999. Resolving packaging mismatch. Ph.D. thesis, Carnegie-Mellon University.

D , R. 2001. Avoiding packaging mismatch with flexible packaging. *IEEE Transactions on Software Engineering 27,* 2 (Febuary), 124–143.

D , D., K , D., O , S., L , W., W , J. 1997. Applying software product-line architecture. *IEEE Computer 17*, 49–55.

D , E. M. 1992. Reuse is not done in a vacuum. In *5th annual Workshop on Software Reuse (WISR)*, L. Latour, Ed.

D , L. K , J. 1995. Reuse dimensions. In *SSR '95: Proceedings of the 1995 Symposium on Software reusability*. ACM Press, New York, NY, USA, 137–149.

F P . 1994. An empirical study of representation methods for reusable software components. *IEEE Transaction on Software Engineering 20,* 8 (August), 617–630.

F , W. B. F , C. J. 1995. Sixteen questions about software reuse. *Communications of the ACM 38,* 6 (June), 75–87.

G , C. 1995. Exploiting domain architectures in software reuse. In *Symposium on Software reusability*. ACM SIGSOFT Software Engineering Notes, vol. 20. 229 – 232.

G , C. 1998. Detecting architectural mismatches during systems composition. Ph.D. thesis, University of Southern California.

G , E., H , R., J , R., V , J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

G , D. 2000. Software architecture: a roadmap. In *International Conference on Software Engineering*. 91 – 101.

G , D., A , R., O , J. 1995. Architectural mismatch or why its hard to build systems out of existing parts. In *International Conference Software Engineering*. 179–185.

G , J. S , K. 2004. *Software Factories Assembling Applications with Patterns, Models, Frameworks and Tools*. Wiley Publishing Inc.

G , M. 2000. Implementing product-line features with component reuse. In *International Confernce on Software Reuse*. Springer-Verlag, 137–152.

H , W. E. 1952. On the rate of gain of information. *The Quarterly journal of experimental psychology 4*, 11–26.

H , E. H , J. 2004. Advice weaving in aspectj. In *Proceedings of the 3rd international conference on Aspect-oriented software development*. Lancaster, 26–35.

H , J. M. 2006. Organizing the asset base for product derivation. In *Software Product LIne Conference (SPLC)*. Baltimore, MD.

H , J. M. M G , J. D. 2006. A series of choices: Variability in the development process. In *ACM Southeast Regional Conference*. Melbourne, FL.

H , J. M. S , M. 2004. Enhancements enabling flexible feature and implementation selection. In *International Conference on Software Reuse (ICSR)*. Madrid, 86–100.

H , R. 1953. Stimulus information as a determinant of reaction time. *Journal of Experimental Psychology 65*, 3946.

IEEE. 1995. Ieee standard for information technology - software reuse - data model for reuse library interoperability: Basic interoperability data model (bidm). Standard 1420.1, IEEE.

J , M., K , R. L., B , J. 2004. Representing variability in a family of mri scanners. *Software Practice and Experience 34,* 1, 69–100.

J , S., B , P., Z , H., Z , W. 2003. Xvcl: Xml-based variant configuration language. In *Proc. Int. Conf. on Software Engineering, ICSE03*. Portland, OR, 810–811.

J , C. 1986. *Programming productivity*. Mcgraw-Hill.

J , C. 1996. Programming languages table, release 8.2.

K , A. K. S , S. 2000. *Product Design for Modularity*. Kluwer Academic Publishers, Boston.

K , K. C., C , S. G., H , J. A., N , W. E., P , A. S. 1990. Feature-oriented domain analysis (foda) feasibility study. Tech. Rep. (CMU/SEI-90-TR-021), Software Engineering Institute, Carnegie Mellon University, Pittsburgh.

K , B. M , M. 1999. Using patterns to model variability in product families. *IEEE Software 16,* 4 (July/August), 102–108.

K , G., H , E., H , J., K , M., P , J., G , W. 2001. Getting started with aspectj. *Communicatons of the ACM 44,* 10 (October), 59 65.

K , C. D. S , J. 1996. Dagar: a process for domain architecture definition and asset implementation. In *Proceedings of the conference on TRI-Ada '96: disciplined software development with Ada*. 231 – 245.

K , C. 2001. Easing the transition to software mass customization. In *Software Product Family Engineering: 4th International Workshop, PFE*. Bilbao, Spain.

K , C. W. 1992. Software reuse. *ACM Comput. Surv. 24,* 2, 131–183.

L , T. K. N , D. W. 1985. Selection from alphabetic and numeric menu trees using a touch screen: Breadth, depth, and width. In *Computer Human Interface (CHI)*. 73–78.

L , K. K , K. C. 2004. Feature dependency analysis for product line component design. In *International Conference on Software Reuse*. Number 3107 in LNCS. 69–85.

L , B. 2005. Big lever case study: Engenio. Tech. Rep. Tech Rpt 2005-06-14-1, Big Lever, Inc.

L , A., C , R., D , T. M., K , E. A., H , S. O., H , P. 1989. Change oriented versioning in a software engineering database. In *2nd International Workshop on Software Configuration Management*. Princeton, N.J., 5665.

L , M. I , N. 1993. Frameworks versus libraries: A dichotomy of reuse strategies. In *Proceedings 6th Annual Workshop on Software Reuse (WISR).*

M , M., B , J., F , M. E. 1999. Framework integration problems, causes, solutions. *Communications of the ACM 42,* 10 (October), 80–87.

M G , J. D. 2006. Arcade game maker product line. www.cs.clemson.edu/ johnmc/productLines/example/frontPage.htm.

M G , J. D. S , D. A. 1992. *Object-Oriented Software Development: Engineering Software for Reuse*. Van Norstrand Reinhold, New York.

M I , M. D. 1968. Mass produced software components. In *In Software Engineering; Report on a conference by the NATO Science Committee*, N. P and B. Randel, Eds. NATO Scientific Affairs Division, 138 –150.

M , A. H , G. T. 2002. Evolving legacy system features into fine-grained components. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*. ACM Press, New York, NY, USA, 417–427.

M -W . 2006. Merriam-webster online dictionary.

M , H., A -K , E., G , R., M ., H. 1997. Another nail to the coffin of faceted controlled-vocabulary component classification and retrieval. In *Symposium on Software Reusability (SSR)*. ACM Press, 89 – 98.

M , R., M , A., M , R. 1998. A survey of software reuse libraries. *Annals Software Engineering 5,* 2, 349–414.

M , G. C., L , A., W , R. J., R , M. P. 2001. Separating features in source code: An exploratory study. In *23rd International Conference on Software Engineering (ICSE'01).*

M , D. A , C. 2002. Model-driven product line architectures. In *Software Product Line Conference*. Number 2379 in LNCS. Springer, 110–129.

M , D. P , T. 2002. Generic implementation of product line components. In *Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*. Number 2591 in LNCS. Springer-Verlag, 313 – 329.

N , V. A. 1993. Moving beyond library-based reuseowego. In *6th annual workshop on software reuse (WISR)*.

O , P. H , J. 1994. Construction and testing of polynomials predicting software maintainability. *Journal of Systems and Software 24,* 3 (March), 251–266.

O , H., H , W., T , P. 2000. Software engineering tools and environments: a roadmap. In *International Conference on Software Engineering*. 261 – 277.

O . 2004. Function point analysis. In *A Dictionary of Computing.* Oxford Reference Online. Oxford University Press.

P , D., C , P., W , D. 1984. The modular structure of complex systems. In *International Conference on Software Engineering*. Orlando.

P , D. L. 1976. On the design and development of program families. *IEEE Transactions on Software Engineering SE-2,* 2 (March), 1–9.

P , D. E. 1998. Generic architecture descriptions for product lines. In *2nd ESPRIT ARES Workshop*. Number 1429 in LNCS. SpringerVerlag, 51–56.

P . 1999. *Phaedrus*. Number 1636. Project Gutenberg Etext.

P , J. S. 1994. Balancing the need for large corporate and small domain-specific reuse libraries. In *ACM symposium on Applied computing*. 88–93.

P , J. S. 1999. Technical opinion: reuse: been there, done that. *Communications of the ACM 42,* 5 (May), 98–100.

S , K. 2002. A comprehensive product line scoping approach and its validation. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*. ACM Press, New York, NY, USA, 593–603.

SEI. 2005. About software product lines. Software Engineering Institute Web Site.

S , L. M., P , J., L , K. J. 1996. Evolution of object behavior using context relations. In *4th ACM SIGSOFT symposium on Foundations of software engineering*. 46 – 57.

S , C. E. 1948. A mathematical theory of communication. *The Bell System Technical Journal 27*, 379423, 623656.

S , M. 1995. Architectural issues in software reuse: Its not just the functionality, its the packaging. In *Symposium on Software Reusability*. ACM Press New York, 3–6.

S , M. A. 1995. Organization domain modeling (odm): formalizing the core domain modeling life cycle. In *Symposium on Software reusability*. 196 – 205.

S , M., G , O., B , J. 2004. Tool support for covamof. In *Proceedings of the Workshop on Software Variability Management for Product Derivation - Towards Tool Support.*

S , M., D , S., N , J., B , J. 2004. Managing variability in software product families. In *2nd Groningen Workshop on Software Variability Management*.

S , M., T , C., B , B., M , A., P , O., S , W., F , S. 2004. Introducing pla at bosch gasoline systems: Experiences and practices. In *Software Product Line Conference*. Number 3154 in LNCS. Springer.

S , M., G , J., B , J. 2005. A taxonomy of variability realization techniques. *Software Practice & Experience 35,* 8 (July), 705 – 754.

T , S. H , A. 2002. Systematic integration of variability into product line architecture design. In *Software Product Line Conference*. Number 2379 in LNCS. 110–129.

T , W. 1991. A conceptual model for megaprogramming. *ACM SIGSOFT Software Engineering Notes 16,* 3 (July), 36–45.

T , W. 1994. Domain-specific software architecture frequently asked questions. *ACM software engineering notes 19,* 2 (April), 52 – 56.

T , D. 1989. Object-oriented development for open systems. In *IFIP Congress 1989 Information Processing 1989*. North Holland, 1033–1040.

H , A. 2004. Design-time product line architectures for any-time variability. *Science of Computer Programming Special Issue Software Variability Management 53,* 3 (December), 285–304.

D , A. K , P. 2001. Domain-specific language design requires feature descriptions. *Journal of computing and Information Technology 10,* 1, 1–24. http://www.cis.uab.edu/cs693/DeursenFDL.pdf.

G , J., B , J., S , M. 2001. On the notion of variability in software product lines. In *Working IEEE/IFIP Conference on Software Architecture (WISCA'01)*. Amsterdam, The Netherlands, 45 – 54.

O , R. B , J. 2002. Widening the scope of software product lines from variation to composition. In *Software Product Line Conference*. Springer-Verlag, 328–347.

O , R., L , F., K , J., M , J. 2000. The koala component model for consumer electronics software. *IEEE Computer 33,* 3 (March), 78–85.

W , R. J. M , G. C. 2000. Implicit context: easing software evolution and reuse. In *8th ACM SIGSOFT international symposium on Foundations of software engineering: twenty-first century applications*. 69 – 78.

W , D. L. G , H. 2004. Modeling variability in software product lines with the variation point model. *Software Variability Management - Science of Computer Programming Special Issue 53,* 3 (December), 305–331.

W , J. G. 2000. Supporting diversity with component frameworks as architectural elements. In *International Conference on Software Engineering*. 51–60.

W , C. 2006. Pattern language. *Wikipedia, The Free Encyclopedia*.

W , N. 1971. Program development by stepwise refinement. *Communications of the ACM 14,* 4 (April), 221–227.

Y , R. K. 1989. *Case Study Research: Design and Methods*. Applied Social Research Methods Series, vol. 5. Sage Publications.

Z , D. C , G. 2003. Measures for software product lines software engineering and analysis initiative. Tech. rep., Software Engineering Institute. October.