Analyzing Aspects in Production Plans for Software Product Lines

Paul Noordhuizen

July 2006

Analyzing Aspects in Production Plans for Software Product Lines

Master of Science Thesis

Paul Noordhuizen

Enschede, July 2006

Graduation committee
prof. dr. ir. Mehmet Akşit
dr. ir. Bedir Tekinerdoğan (first supervisor)
dr. ir. Klaas van den Berg

Chair
Software Engineering

Departement
Electrical Engineering, Mathematics and Computer Science

University
University of Twente

Abstract

Software product line engineering aims to reduce the costs of manufacturing software products by exploiting the commonalities of a product family and managing the variabilities. Production plans define the process for producing software products from the available assets. However, it appears that product line engineering has not yet focused on crosscutting concerns in production plans.

We think that for coping with these crosscutting concerns aspects can be applied, as aspects are already used throughout the software development cycle to modularize crosscutting concerns and to provide composition mechanisms with other concerns.

Firstly, we analyze the problems with crosscutting concerns for production plans on two levels: the component level, that is crosscutting in the asset library from which products are built through the production plan, and the production plan level, that is scattering of variable features in the production plan itself. We identify these problems in a case study of a concrete product line and propose solutions for both levels.

On the component level, our approach is to modularize crosscutting concerns in separate, reusable aspect components and to provide configuration of the aspects to select the right variation of each aspect for the specific product that is to be produced. These configured aspects are then composed with the other selected assets through a separate pointcut specification to avoid context-specific references in the aspect implementation. This process is defined for the production plans.

On the production plan level, our approach is to modularize the variable features in the production plans by using XML-based feature models in the product line and the functional query language XQuery to select features by their type (common or variable) and/or name in stead of by their place in the feature hierarchy.

Our solutions on both levels are illustrated and demonstrated with the case as a running example.

Secondly, we explore the impact of aspect-orientation on the product line process. Product line engineering is moving more and more from production of software products by hand to automated generation of applications from the product line through some sort of specification. This goal obviously has implications for the structure and contents of production plans. However, the steps in a generative product line process – and thus in generative production plans – are unclear. We have analyzed the product line process for the popular generative technologies *XML-Based Feature Modeling* and *Pure::Variants* and investigated the impact of aspect-orientation on the generative production plans.

Contents

List of Figures

List of Tables

List of Abbreviations

AGM	Arcade Game Maker
AO	Aspect-Oriented
AOP	Aspect-Oriented Programming
AOSD	Aspect-Oriented Software Development
ATAM	Architecture Tradeoff Analysis Method
BOM	Bill of Materials
CBSE	Component-Based Software Development
CCM	Corba Component Model
CONOPS	Concept of Operations
COTS	Commercial Off-The-Shelf
DCOM	Distributed Component Object Model
EJB	Enterprise JavaBeans
FODA	Feature-Oriented Domain Analysis
MVC	Model-View-Controller
RUP	Rational Unified Process
SPLE	Software Product Line Engineering
UML	Universal Modeling Language

Foreword

This Master of Science thesis is the final report of my Computer Science studies at the University of Twente. It is the result of 12 months of work in which I have been supported by many people. I now have the opportunity to express my gratitude to all of them.

First of all, I would like to thank my supervisors Bedir Tekinerdoğan and Klaas van den Berg for their guidance, support, and patience. The dedication with which you have supported and encouraged me in all those meetings has motivated me to reach the result as lays before you today.

I also wish to thank the third member of the graduation committee, Mehmet Akşit, for his feedback and reserving time in a busy schedule to attend my final presentation.

Thanks go out to all of my friends who have supported me during this period. You gave me the opportunity to clear my head once in a while to keep me going. Special thanks go out to my girlfriend for her patience and encouragement.

Last, but not least I would like to thank my sister and parents who have been there for me during all these years and motivated me to finish my studies.

Chapter 1

Introduction

1.1 Context

The software engineering industry is becoming more and more demanding. It requires software of high quality to be developed in shorter periods of time against less cost. Achieving these goals using traditional development methods becomes an impossible task, as has been proved over and over again by software projects which exceed their time schedule and budget. Also, with technology persistently advancing and information technology demands constantly changing, the only certain factor in such an environment is that change will occur. It is therefore important to take these changes into consideration when developing software systems.

Focus needs to be shifted from developing single software products to product *lines* (also known as product *families*). A product line is a collection of software products sharing a common set of features that address the specific need of a defined domain. Product lines are created to effortlessly accommodate changes and to speed up the software production process. Product line engineering helps designing reusable components and a common architecture for a product line.

Reuse is a concept well-known to software engineers developing software families. Reuse-driven development is not only used by product line engineering, but also other development approaches such as generative programming and aspect-oriented software development. These approaches reuse various software engineering artefacts, from software architectures to code, for the development of software families. Reuse is achieved firstly, by determining the commonality and variability of the software products to be developed and secondly, by reusing those parts found to be common for all products. *Feature modeling* is used to specify the commonality and variability of a software family.

Product line engineering aims to reduce the costs of manufacturing of software products by exploiting these common properties and by managing the variabilities. Two key processes in product line engineering are domain engineering and application engineering. In domain engineering the scope of the product line is defined and the required components are implemented in reusable libraries. In application engineering these components are composed together in products using a so-called production plan.

1.2 Problem Statement

Obviously, the main goal of domain engineering is to capture the concerns in single components so that these can be made reusable. However, It appears that product line engineering has not yet focused on so-called crosscutting concerns for production plans, that is, concerns which cannot be easily localized in individual components and that tend to be scattered over multiple components. The lack of techniques for coping with crosscutting concerns for production plans in a product line engineering approach will reduce the reusability that was initially aimed for. Aspect-Oriented Software Development (AOSD) is an approach for coping with crosscutting concerns by providing explicit abstractions called aspects and composition mechanisms for composing the aspects with the components. Unfortunately, aspects have not yet been applied for production plans.

This thesis investigates two types of crosscutting concerns with respect to the production plans: crosscutting concerns on the component level, that is in the library of components from which end-products are composed through the production plan, and aspects in the production plan itself which aim to tackle scattering of variable features in the plan.

We study these issues through the following research questions:

1. Which problems arise with crosscutting concerns on the **component level** in a product line context?
2. Which problems arise with crosscutting concerns on the **production plan level** in a product line context?
3. How can aspects be applied to production plans to cope with these crosscutting issues?
4. Introducing aspect-orientation for production plans also affects the product line process. Product line engineering is moving more and more from production of software products by hand to automated **generation** of products from the product line through some sort of product specification. However, the steps in a **generative product line process** are unclear. This leads to three questions:
    - Which technologies are available for automation of production plans?
    - What are the product line processes for these technologies?
    - How do aspects on the component level and production plan level affect the product line process?

1.3 Approach

To answer these research questions we first investigate a number of key concepts of software product line engineering and aspect-oriented software development. Then we introduce a case study of a concrete product line to illustrate which activities and assets are involved in product line engineering and specifically how products are produced through the production plans.

The case study is then used to investigate the impact of crosscutting concerns in a product line context and which problems arise on both the component level and production plan level.

We then work out our approach to these issues on both levels by applying aspects to modularize the crosscutting concerns and we introduce configuration and composition techniques from AOSD to work with aspects as separate, reusable components in a product line context.

We finish this thesis with our approach to applying so-called *generative production plans*, which are production plans that produce software products in an automated way. The structure and contents of such production plans and the according product line process are investigated for two popular generative approaches.

1.4 Thesis Overview

A brief introduction to software product line engineering and aspect-oriented software development is given in **chapter 2**.

Chapter 3 presents a case study of the *Arcade Game Maker* product line to give the reader more feeling for the product line approach and the activities involved. This case is used as a running example in the report to illustrate abstract issues with concrete examples from the case.

In **chapter 4** we study the impact of crosscutting concerns in the case and explore and problems that arise due to these concerns on both the component level and production plan level.

Chapter 5 proposes a classification of product lines, based on the possible combinations of aspect-orientation in the *asset library* and the production plans and aspects are applied to the identified problems. We also introduce configuration and composition techniques from AOSD to work with aspects as separate, reusable components in a product line context.

Generative production plans and the generative product line process are explored in **chapter 6** with two generative technologies and an analysis of their processes. The impact of aspect-orientation for the product line process is then investigated.

Chapter 7 concludes this report by summarizing the answers found for our research questions, by making recommendations, and discussing ideas for future research.

Chapter 2

Background

2.1 Introduction

This chapter gives a brief introduction into the field of software product line engineering and aspect-oriented software development. Key concepts as *commonality and variability*, the *two life-cycle model*, *separation of concerns*, and *crosscutting concerns* are introduced. We also discuss the main product line activities of *core asset development* and *product development*.

2.2 Software Product Line Engineering

2.2.1 Introduction

A software product line is *a set of software-intensive systems that share a common, managed set of features satisfying 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* [14].

Software product line engineering is targeted at organizations concerned by the lack of predictable delivery and long-term return on their software investments. Product lines are becoming an important approach to reducing the costs and the time to market of complex software products. Some advantages of a product line of software systems when built from a common set of shared *assets* are an increase in profits, cost savings, system reliability, customer satisfaction, as well as a decrease in staffing requirements and time-to-market. Organizations using product lines for the development of software systems have a great competitive advantage.

In order to develop highly reusable *core assets*, product line engineering must have the ability to exploit commonality and manage variability among products in a certain *domain*. Systematic management of planned variations across a product line and exploiting commonalities is essential for successful software product line engineering. The commonality being exploited allows reuse of a number of shared assets, such as architectures, reusable components, schedules, budgets, test cases, performance modeling, training and documentation [14][41]. When developing a new product in a product line, composition and generation are encouraged rather than programming as in the traditional way. By reusing parts of previous systems to build new systems, their reliability is considerably increased.

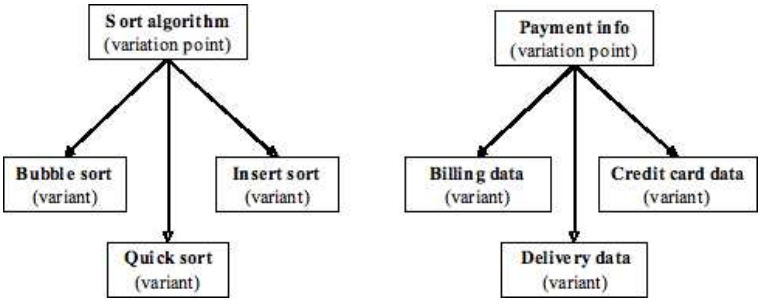

Figure 2.1: Examples of a variation point and its variants [15]

In this section we first further investigate the notion of commonality and variability. We then give an overview of the two main activities that take place in product line engineering: *core asset development* and *product development*. Then we introduce the *two life-cycle model* which shows the relations between domain engineering and application engineering.

2.2.2 Commonality and Variability

Let us now discuss two major concepts in software engineering in general and product line engineering: *commonality* and *variability*. Much research is available on the topics of commonality and variability analysis and management [15][28][31][37][54].

Considering a software product family, commonality represents the functionality uniformly occurring across all products in the family, whereas variability represents those characteristics only occurring in some, but not all of the products. For example, an attribute having the same value in all products is seen as commonality and when having different values for at least two of the products, as variability. Another example – moving away from product families – is looking at a set of figures, circles, triangles and squares. Each one of these figures is two-dimensional and has an area. Thus both of these features, 'two-dimensional' and 'has an area' are commonalities. Some of these shapes are differentiated by the number of sides they have and the formula of their area, thus 'number of sides' and 'formula of area' are variabilities.

At this stage the notion of a *variation point* is introduced. A variation point is any point in a product at which variation may occur. A variation point locates a variability and its bindings by describing several variants. Every variant is one way to realize that variability and bind it in a concrete way. The variation point itself locates the insertion point for the variants and determines the characteristics of the variability. An example of a variation point could be a sorting algorithm. The variants of this variation point would be bubble sort, quick sort or insert sort. Another example of a variation point in the case of an order process could be payment information. Its variants would be billing data, delivery data and credit card data. Figure 2.1 illustrates these examples.

Commonality and variability can also be expressed in terms of *features*, as we will get into in section 2.2.4. Features abstract from requirements which is important in product line engineering, because by combining a number of products in a family, the size and complexity of the requirements also increases, which severely hinders the detection of commonalities and variabilities in the family. *Traceability* is provided by mapping features in the *problem space* to implementation of the features in the *solution space*.

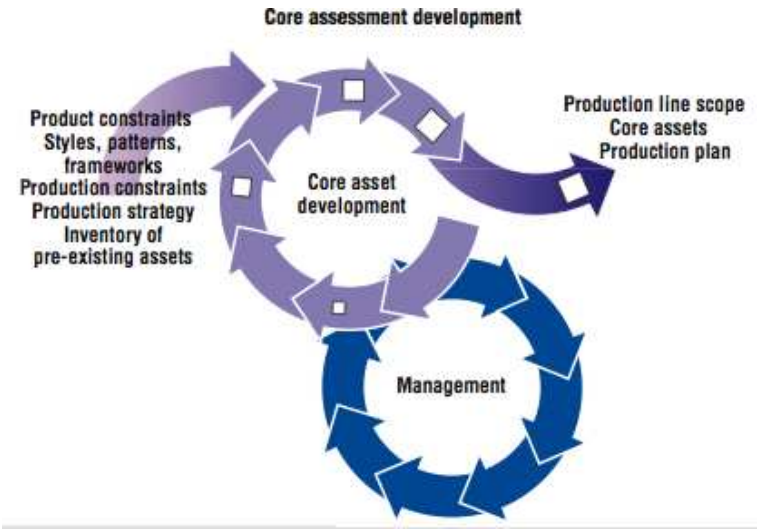

Figure 2.2: Core Asset Development activity [41]

Analyzing the commonality and variability of a domain provides a systematic way of thinking about and identifying the product family to be created and helps to analyze the economics of creating a family. Understanding the commonality and variability of software products allows functions, objects, and aspects – as we will see later on in this report – to be derived in a reusable form. Determining the commonality and variability of a product line is not always as straightforward as it seems. There are many different facets of any product family that need to be considered before distinguishing between its common and variable features. Once these features have been identified, representing them in a way useful to the developer is another problem not easily solved.

2.2.3 Product Line Activities

The Software Engineering Institute at Carnegie Mellon University has developed a framework for product line practice, which aids organizations with the activities of developing a product line [14]. It identifies and describes 29 practice areas, which are divided into software engineering practice areas, technical management practice areas and organizational management practice areas.

Product line practice consists of three main activities. They are the development of the core assets, the development of the products using the core assets, and the management of both these processes. The first two activities are described below.

Core Asset Development

The goal of this activity is to create product production capability. Figure 2.2 illustrates the core asset development activity along with its outputs and necessary inputs.

This activity is iterative and by the diagram it is evident that the inputs and outputs of this activity affect each other.

Three of the outputs of the core asset development activity are the product line scope, the core assets and the production plan. They are necessary for developing products from the product line.

The product line scope describes the products that will be included in the product line in terms of the commonalities and variabilities amongst the products. These might include, for example, features, their operations, their quality attributes such as performance and the platforms on which they run. The success of the product line depends on the careful definition of the scope. The scope may not be too large, or too small, and it must target the right products. For the product line to be kept current, the scope should evolve as market conditions change, as the organization's plans change, or as new opportunities arise.

The core assets are the basic building blocks of the product line. They include an architecture shared by the products in the product line and software components developed for systematic reuse. The software architecture plays an important role in that it must satisfy the general needs of the product line, as well as the individual products by explicitly admitting a set of variation points. Requirements specifications, domain models and commercial off-the-shelf (COTS) components are also classified as core assets. A core asset should always be associated with an attached process. The attached processes are step-by-step descriptions of how the core assets will be used in the development of the products in the product line. They are themselves core assets that are included in the production plan.

The production plan describes how products will be developed using the core assets. It is essentially a set of attached processes from each of the core assets. Every product in the product line has specific variations predefined by variation points. The production plan describes how these variation points can be accommodated. It also describes how the core assets are linked and how they are utilized effectively and within the product line constraints.

The three outputs of the core asset development activity are necessary for the product development activity, which in turn produces the products of the product line that fulfill a specific customer or market need.

Product Development

Increasing the productivity of product development is the main drive of product line engineering. The product development activity is responsible for assembling each product. It depends on the three outputs described above, as well as the requirements for each individual product. Figure 2.3 illustrates these relationships.

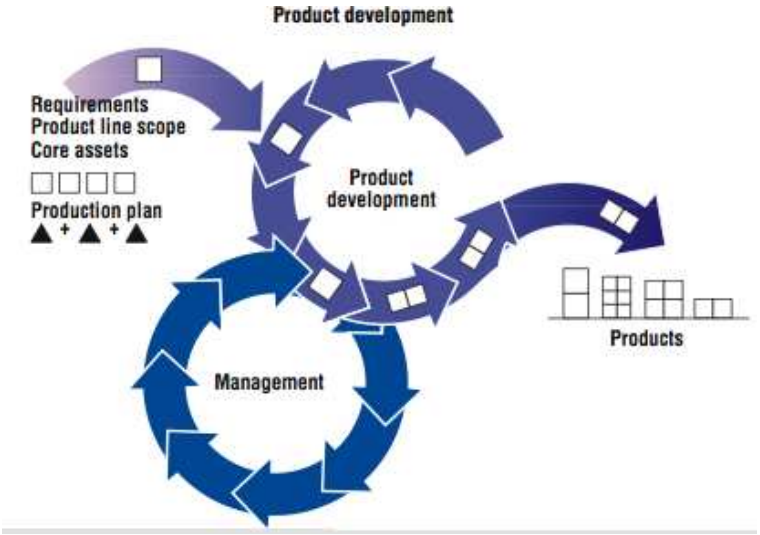

Figure 2.3: Product Development activity [41]

The requirements for a particular product are essentially the product features and are necessary for producing a specific product. The product line scope determines whether a certain product can be included in the product line or not. The core asset base is needed to build the products. The production plan describes how the core assets should be used in order to build a product.

From a simplified point of view, the product development activity consists of receiving requirements for a product that is in the product line scope, and then following the production plan so that the core assets can be correctly used to develop the product. The process is however not always that simple, as we will see later on in this report.

2.2.4 Two Life-Cycle Model

The product line activities can be grouped in two categories: *domain engineering*, and *application engineering*. Figure 2.4 shows the process and products of the overall domain engineering activity, and shows the relationships and interfaces of domain engineering to the application engineering process. This has come to be known as the *two life-cycle model* [9][15][57].

We start in this section with a discussion of the domain engineering processes.

There are different meanings assigned to the term *domain* in different disciplines and communities. In this report we use the definition adopted from Czarnecki & Eisenecker [15]:

> Domain: An area of knowledge:
>
> - Scoped to maximize the satisfaction of the requirements of its stakeholders.
> - Includes a set of concepts and terminology understood by practitioners in that area.
> - Includes the knowledge of how to build software systems (or parts of software systems) in that area.

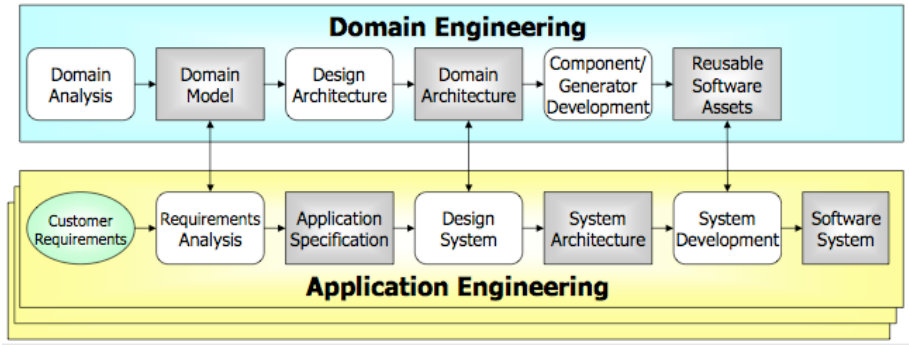

Figure 2.4: Two Life-Cycle Model [57]

Domain engineering then is *the activity of collecting, organizing, and storing past experience in building systems or parts of systems in a particular domain in the form of reusable assets (i.e., reusable work products), as well as providing an adequate means for reusing these assets (i.e., retrieval, qualification, dissemination, adaptation, and so on) when building new systems* [15].

Domain engineering consists of *Domain analysis*, *Domain design*, and *Domain implementation*. The results of domain engineering are reused during application engineering, that is, the process of producing concrete systems using the reusable assets developed during domain engineering.

Domain analysis

The purpose of *domain analysis* is to [15]:

- Select and define the domain of focus
- Collect the relevant domain information and integrate it into a coherent *domain model*

The sources of domain information include existing systems in the domain, domain experts, system handbooks, textbooks, prototypes, experiments, already known requirements on future systems, current or potential customers, standards, market studies, technology forecasts, and so on.

It is important to note that domain analysis does not only involve recording the existing domain knowledge. The systematic organization of the existing knowledge enables and encourages to actually extend it in creative ways [15].

A *domain model* is an explicit representation of the *common* and *variable* properties of the systems in a domain, the semantics of the properties and domain concepts, and the *dependencies* between the variable properties. In general, a domain model consists of the following components [15]:

- *Domain definition*: Defines the scope of a domain and characterizes its contents by giving examples of existing systems in the domain, counterexamples, and generic rules of inclusion and exclusion.
- *Domain lexicon*: Defines the domain vocabulary.

- *Concept models*: Describe the concepts in a domain in some appropriate modeling formalism and informal text.
- *Feature models*: Define a set of reusable and configurable requirements for specifying the systems in a domain. A feature model prescribes which feature combinations are meaningful, which of them are preferred under which conditions and why. See section 2.2.4 for more information.

Domain design

The purpose of *domain design* is to develop an *architecture* for the family of systems in the domain and to devise a production plan.

Buschmann et al. give a definition of a *software architecture* [10]:

> A *software architecture* is a description of the subsystems and components of a software system and the relationships between them. Subsystems and components are typically specified in different views to show the relevant functional and nonfunctional properties of a software system. The software architecture of a system is an artifact. It is a result of the software development activity.

The elements in a software architecture and their connection patterns are designed to satisfy the requirements on the system (or systems) to be built. When developing a software architecture, one has to consider not only functional requirements, but also requirements for quality properties, such as performance, robustness, failure tolerance, throughput, adaptability, extendibility, reusability, and so on [15].

Domain implementation

Domain design is followed by domain implementation, which involves implementing the architecture, the components, and the production plan using appropriate technologies.

The product-building process is described in the production plan using the process definition style used for other processes in the organization. The *Product Builder Pattern* described by Clements specifies a set of product line practice areas that are used in building a product in a product line organization [14]. Each practice is a body of work or a collection of activities that an organization must master to carry out the essential work of a product line. Figure 2.5 shows the practice areas used in the pattern and the interactions between them.

For a specific product line, the practice areas necessary to build a product depend on how well the organization has institutionalized the product line practices, the maturity of the market, and the degree of product development automation.

Feature Modeling

As discussed earlier, feature models define a set of reusable and configurable requirements for specifying the systems in a domain. A feature model prescribes which feature combinations are meaningful, which of them are preferred under which conditions and why.

Generally, a feature is *"a property of a domain concept, which is relevant to some domain stakeholder and is used to discriminate between concept instances"* [15].

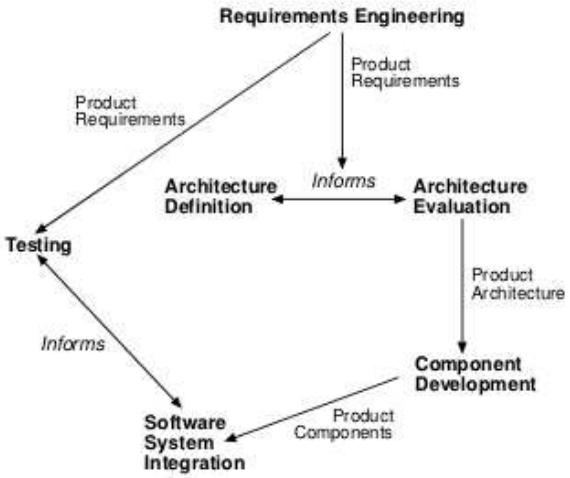

Figure 2.5: Structure of the Product Builder Pattern [12]

Reusable software contains inherently more variability than concrete applications and feature modeling is the key technique for identifying and capturing variability [15]. It is therefore important to have a better understanding of this technique and its benefits for the software product line approach.

A feature model consists of a feature diagram and some additional information, such as short semantic descriptions of each feature, rationales for each feature, stakeholders and (potential) customers interested in each feature, examples of systems with a given feature, default dependency rules, availability sites (i.e., where, when, and to whom a feature is available), binding sites (i.e., dynamic or static binding), open/closed attributes (i.e., whether new subfeatures are expected), and priorities (i.e., how important a feature is) [15].

Feature Diagrams
The most common representation of feature models is through FODA-style feature diagrams [15][28]. *Feature-Oriented Domain Analysis (FODA)* is a domain analysis method developed at the Software Engineering Institute (SEI) and is known for the introduction of feature models and feature modeling. A feature diagram consists of a set of nodes, a set of directed edges, and a set of edge decorations. The nodes and the edges form a tree. The edge decorations are drawn as arcs connecting subsets or all of the edges originating from the same node. The root of a feature diagram is a concept and the remaining nodes in a feature diagram represent features. Consider figure 2.6 for an example feature diagram of a control system which can contain one or more sensors, one or more actuators and one processor. A sensor can be a position sensor *or* a speed sensor and can optionally contain a self test. An actuator is a position actuator and can optionally contain a self test. The processor has a certain internal memory size.

The parent node of a feature node is either the concept node or another feature node. As one can see in the example feature diagram there are different types of features: *mandatory*, *alternative*, *optional*, and *or*-features [15]:

- A *mandatory* feature is included in the description of a concept instance if and only if its parent is included in the description of the instance.

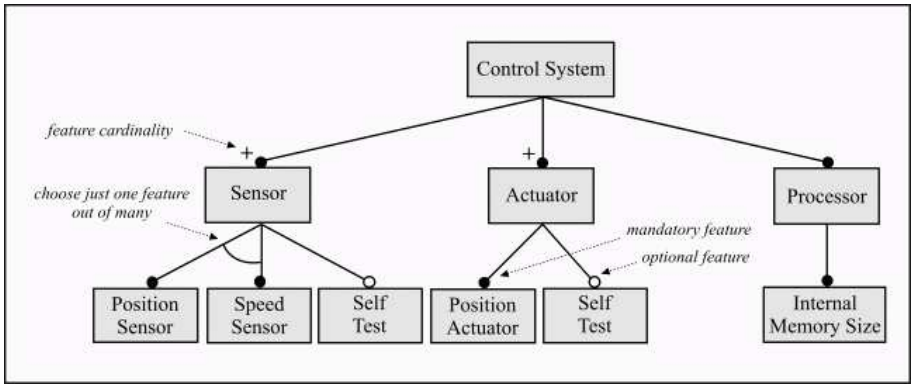

Figure 2.6: Example Feature Diagram [18]

- An *optional* feature may be included in the description of a concept instance if and only if its parent is included in the description.
- If the parent of a set of *alternative* features is included in the description of a concept instance, then exactly one feature from this set of alternative features is included in the description; otherwise none are included.
- If the parent of a set of *or*-features is included in the description of a concept instance, then any nonempty subset from the set of or-features is included in the description; otherwise, none are included.

Feature diagrams allow us to represent concepts in a way that makes the commonalities and variabilities among their instances explicit. For identification of commonalities there are two important notions: *common features* and *common subfeatures*. A feature is a *common feature* of a concept if it is a mandatory feature and either a direct feature of the concept or there is a path of mandatory features connecting the feature and the concept. A subfeature is a *common subfeature* of a feature, which is present in all instances of a concept that also have the feature. Thus, all direct mandatory subfeatures of the feature are common subfeatures. Also, a subfeature is common if it is mandatory and there is a path of mandatory features connecting the subfeature and the feature [15].

Variability in feature diagrams is expressed using optional, alternative, optional alternative, and or-features; these features are called *variable features*. The nodes to which variable features are attached are referred to as the earlier-mentioned *variation points* [15].

Application Engineering

Application Engineering is the process of building systems based on the results of Domain Engineering (see figure 2.4).

During the requirements analysis for a new concrete application, one can take advantage of the existing domain model and describe customer needs using the features (i.e., reusable requirements) from the domain model. This process can be supported by appropriate application ordering tools. New customer requirements not found in the domain model require custom development. The new requirements should also be fed back to domain engineering to refine and extend the reusable assets. In the

software product line approach one then either manually assemble the application from the existing reusable components and the custom-developed components or use generators to produce it automatically [15]. We investigate generative software development in section 6.2.2 and apply this approach to product line engineering in chapter 6.

2.3 Aspect-Oriented Software Development

2.3.1 Introduction

Aspect-oriented software development (AOSD) is *a set of emerging technologies that seeks new modularizations of software systems. AOSD allows multiple* concerns *to be separately expressed but nevertheless be automatically unified into working systems* [20].

An important principle in software engineering is *separation of concerns*, that is to decompose a system into separate *concerns* as a mechanisms for improving flexibility and comprehensibility and thus making software systems easier to write, understand, reuse, and modify.

Related to this principe is the problem of *crosscutting concerns*. Crosscutting is usually described in terms of *scattering* and *tangling*, e.g. crosscutting is the scattering and tangling of concerns arising due to poor support for their modularization.

Aspect-oriented programming (AOP) was introduced to tackle these crosscutting concerns on the code level by using an *aspect* as a module for a crosscutting concern. When this approach to programming started to gain interest in the community, the concepts in AOP stabilized and the notion of *early aspects* came into the picture to support aspect-orientation in earlier phases of the lifecycle.

In this section we investigate the need for aspect-oriented software development by discussing separation of concerns and crosscutting concerns. The reader is referred to sections 4.2 and 5.2.2 for more background information on the important concepts in aspect-oriented programming and how aspects can be *configured* when used in a product line context.

2.3.2 Separation of Concerns

The AOSD research area has its own terminology and themes. Looking at the definition of aspect-oriented software development one can see that modularization and unification of sofware systems are the main topics of interest. Connected to these topics is the notion of *separation of concerns*.

We define a concern as *an interest, which pertains to the systems's development, its operation or any other matters that are critical or otherwise important to one or more stakeholders* [6].

Separation of concerns then is defined as *an in depth study and realization of concerns in isolation for the sake of their own consistency* [6]. The concerns identified during this process are usually described in *concern models*. The reader is referred to appendix A for an overview of concern modeling approaches.

Separation of concerns is a long-established principle in software engineering and is described in many publications [20][25][44]. The general principle of separation of concerns is to decompose a system into separate concerns as a mechanisms for

improving flexibility and comprehensibility and thus making software systems easier to write, understand, reuse, and modify.

At the conceptual level, the separation of concerns needs to address two issues:

1. Provide a clear definition and conceptual identification of each concern that distinguishes it from the others.
2. Ensure that the individual concepts are primitive in the sense that they are not compositions of several concepts.

At the implementation level, the separation of concerns needs to provide an adequate organization that isolates the concerns. The goal at this level is to separate the blocks of code which address the different concerns, and provide for a loose coupling of them.

Applying separation of concerns at *both* the conceptual and the implementation level has a number of benefits [25]:

- Separating concerns results in a higher level of abstraction since one can reason about individual concerns in isolation.
- Separated concerns are easier to understand since their code is not cluttered with the code for other concerns.
- A separation of concerns results in a weak coupling of the concerns and so satisfies the demand for increased flexibility and reusability of single concerns.

Separation of concerns has received widespread attention in modern programming languages, with constructs such as modules, packages, classes, and interfaces, which support properties such as abstraction, encapsulation, and information hiding [51]. Also software architecture and design use separation of concerns, with techniques such as composition filters [1] and design patterns [21].

2.3.3 Crosscutting Concerns

Related to the principle of separation of concerns is the problem of *crosscutting concerns*. Crosscutting is usually described in terms of *scattering* and *tangling*, e.g. crosscutting is the scattering and tangling of concerns arising due to poor support for their modularization.

However, the distinction between these three concepts is vague, sometimes leading to ambigious statements and confusion. Van den Berg & Conejero describe a conceptual framework with precise definitions of scattering, tangling and crosscutting [7]. Their proposition is a *crosscutting pattern* where the three concepts are defined in terms of a *source* with respect to a *target*, and elements in the source are related to elements in the target. The terms crosscutting, tangling and scattering are defined as special cases of these mappings.

Based on this pattern, scattering can be defined as *when, in a mapping between source and target, a source element is related to multiple target elements* [7], and tangling as *when, in a mapping between source and target, a target element is related to multiple source elements* [7]. This means that two source elements are tangled if these elements are mapped onto the same target element.

Crosscutting is a specific combination of scattering and tangling *when, in a mapping between source and target, a source element is scattered over target elements and*

dependency matrix

		Target				
		t[1]	t[2]	t[3]	t[4]	
source	s[1]	1	0	1	1	S
	s[2]	0	1	0	0	NS
	s[3]	0	0	1	0	NS
		NT	NT	T	NT	

crosscutting matrix

		source		
		s[1]	s[2]	s[3]
Source	s[1]	0	0	1
	s[2]	0	0	0
	s[3]	0	0	0

Figure 2.7: Example dependency and crosscutting matrix [7]

where in at least one of these target elements, some other source elements are tangled [7]. This means that a source element s1 crosscuts source element s2 if s1 is scattered over target elements, and in at least one of these target elements, s1 is tangled with source element s2. Following from these definitions, tangling and scattering are necessary but not sufficient conditions for crosscutting.

To facilitate identification of crosscutting during the development cycle Van den Berg & Conejero propose a representation of crosscutting in matrices: a *dependency* matrix, a *crosscutting* matrix, and optionally a *scattering* and a *tangling* matrix. A dependence matrix (source x target) represents the dependency relation between source elements and target elements (inter-level relationship). A cell with 1 denotes that the source element is mapped to to the target element. Scattering and tangling can be easily visualised in this matrix: see figure 2.7 for an example. A matrix cell involved in both tangling and scattering is called a *crosscutpoint*. If there are one or more crosscutpoints then crosscutting occurs.

A crosscutting matrix (source x source) represents the crosscutting relation between source elements, for a given source to target mapping (represented in a dependency matrix). This is also visualised in figure 2.7. A crosscutting relation isn't necessarily *symmetric*: it is possible that source element s1 crosscuts source element s2, while not vice versa because s2 is not scattered.

Based on a dependency matrix one can define some auxiliary matrices: the scattering matrix (source x target) with just scattering, and the tangling matrix (target x source) which just tangling. Then the *crosscutting product matrix* (source x source) can be obtained by the matrix multiplication of the scattering matrix and the tangling matrix. The crosscutting product matrix represents the *frequency* of crosscutting relations between source elements, for a given source to target mapping. These extra matrices can be calculated automatically by using the dependency matrix [7].

From the attained crosscutting model, it is in some cases possible to avoid tangling, scattering and crosscutting by choosing another decomposition of source and target. The possibilities are determined by the expressive power of the languages in which the source and target are expressed. In case where limitations in the expressive power of the languages are the cause of tangling, scattering and/or crosscutting Van den Berg & Conejero use the terms *intrinsic tangling*, *intrinsic scattering*, and

intrinsic crosscutting [7].

The extension of a language with new constructs and new composition operators – such as aspects or composition filters – may change the (de)composition of source and target.

2.4 Summary

In this chapter we gave a short introduction into important concepts of software product line engineering and aspect-oriented software development.

We have defined what a software product line is, what the goals of product line engineering are, and how commonality and variability analysis is an essential part of product line engineering. We have also discussed the main product line activities of core asset development and product development and how these two activities are related.

The two life-cycle model has been introduced with the key product line processes of domain engineering and application engineering and the different steps in these processes.

Separation of concerns was discussed as an important principle in software engineering. Related to separation of concerns is the notion of crosscutting concerns. How crosscutting occurs in terms of scattering and tangling has been explored. AOSD was introduced as a set of emerging technologies to cope with these crosscutting concerns and support their modularization.

Chapter 3

Case Study: Arcade Game Maker Product Line

3.1 Introduction

Much literature is available on software product line engineering and the related research areas, as we have summarized in the background chapter. However, most studies discuss software product lines in an abstract way, while the application of the product line engineering approach is at least as interesting. This is why we present a case study of a concrete product line here to give the reader more feeling for the product line approach and the activities and assets involved.

The case study is based on the *Arcade Game Maker Product Line* which is a pedagogical product line introduced by John D. McGregor of Clemson University and the Software Engineering Institute (SEI) [35]. The case is used as a running example in this report to illustrate abstract issues with concrete examples from the case.

We first give an overview of the context and the assets in this example product line. Then we discuss the assets for the different software life-cycle phases in more detail. After this we focus on the production plans in the case.

3.2 Case Overview

The *Arcade Game Maker* (AGM) product line is a case of the fictitious company AGM that develops arcade games for desktop pc's and wireless devices (mobile phone, PDA, etc.). The company strategy is to initially develop three arcade games, named *Brickles*, *Pong* and *Bowling* and to offer these products in three stages:

- Stage 1: Distribute the products as freeware games for the desktop to market the company brand.
- Stage 2: Sell the games for wireless devices (mobile phone, PDA, etc.).
- Stage 3: Sell the games as convention giveaways and incorporate a company's message in the games.

The goal of the *Brickles* game is to break all the bricks in the playing field (see figure 3.1). The player has three pucks to use to break all of the bricks in the brickpile. Every time a puck hits the floor it is removed from play. If the player manages to break all the bricks, the player wins. If the player runs out of pucks while playing, the player loses. The player can also control the speed of the puck by moving a slider to the left (slower) or right (faster).

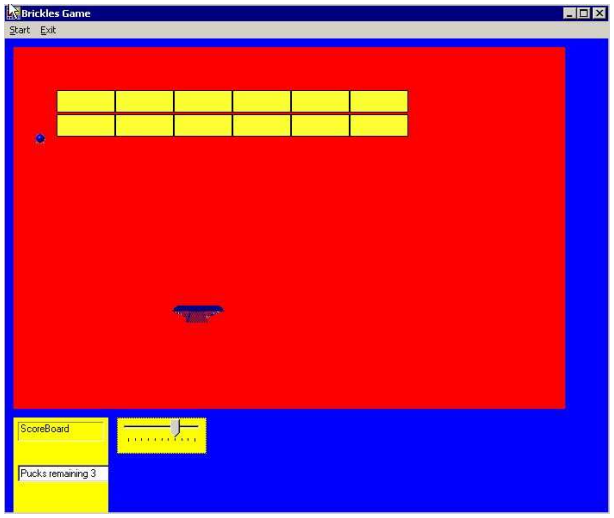

Figure 3.1: The Brickles Playing Field [35]

The *Pong* game is well-known to most readers: its concept is a simplified kind of tennis (see figure 3.2). The AGM version of the game is single player, where the objective is to keep the puck in play as long as possible. The player keeps the puck in play by moving the mouse which controls the paddles on either end of the playing field. The puck is absorbed by the left and right border of the playing field. The player controls both paddles but only one at a time. As the mouse moves across the center line, the paddle on the same side of the center line begins responding to mouse movement.

Figure 3.2: The Pong Playing Field [35]

The *Bowling* game rules are conform the real-life bowling rules, specifically:

1. The player attempts to knock down as many pins as possible in the given number of tries (frames).
2. Knocking down all the pins on the first try in a frame is termed a strike.
3. Knocking down all the pins on the two tries in a frame is termed a spare.
4. The score for a frame is the total number of pins knocked down by the two throws.
5. The score for a frame in which a spare is made is 10 plus the number of pins knocked down by the first ball of the next frame.
6. The score for a frame in which a strike is made is 10 plus the number of pins knocked down by the next two balls.
7. In the 10^{th} frame, if a strike or spare is achieved, a third ball is thrown and the number of pins is added to the score for that frame.

The Bowling interface is depicted in figure 3.3.

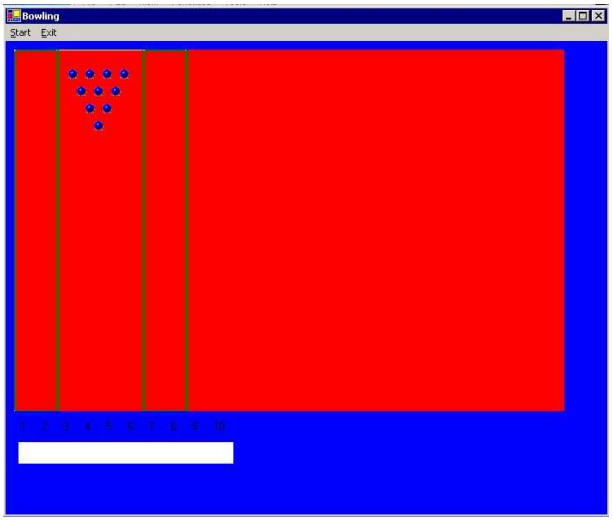

Figure 3.3: The Bowling Playing Field [35]

Depending on the success of these games AGM will extend the product portfolio later on.

AGM has just recently started to use the product line approach to software development as a way to achieve the strategic objectives:

- **Market Position**: AGM wants to be the market leader. The market is sensitive to how rapidly new technologies are introduced into products and the scope of the feature set. Therefore, AGM has decided to become an 'early adopter' of software product lines.
- **Time to Market**: With the product line approach AGM will be able to produce products at an increasingly rapid rate. Many game ideas have a very short life span, so AGM must be quick to develop and deploy a game.
- **Productivity**: To remain competitive, AGM must reduce the cost of building the games. Software makes up roughly 90 percent of the content of current products, so the productivity increase of the product line approach can be substantial.
- **Mass Customization**: AGM sees an opportunity in the area of convention giveaway products. They would like to be able to add a company's logo and other advertising marks to a game and sell it to that company as a marketing handout at conventions. The product line approach can support such variability.

3.3 Product Line Assets Overview

The AGM product line contains a number of assets. Each asset is now discussed briefly in the context of the case.

3.3.1 Business Case

The purpose of a business case is to analyze options for achieving the company's product production goals and to justify whatever approach is selected for achieving those objectives.

For the product line approach different strategies can be applied, categorized as *totally proactive* (the assets are built before the products), *totally reactive* (the assets are built as the products are built), or *incremental* (the product line is divided into sets of products and the assets of a set are built before any products are).

AGM chose the incremental approach, because of the staged release of the game products.

3.3.2 Scope

The scope document defines the boundaries of the product line and the design and implementation decisions that have been made to address the full scope of the product line but with no concern for any characteristics outside the product line.

The AGM product line scope roughly is arcade games where each game is a one-player game in which the player controls, to some degree, the moving objects and the objective is to score points by hitting stationary obstacles. The games range from low obstacle count to high.

3.3.3 Concept of Operations

The Concept of Operations (CONOPS) document is used to capture how the organization will make decisions and how they will manage the production of products in the product line.

AGM divided the product line organization into one permanent core asset team, responsible for domain engineering, and a varying number of temporary product teams, responsible for application engineering. As technical considerations, AGM has decided to use the Unified Modeling Language (UML 2.0) for the architecture and other design assets, Java for the code assets, and the Rational Unified Process (RUP) as the development process.

3.3.4 Requirements

The purpose of the requirements is to provide the specifications for the products that will be built as part of the product line.

In the AGM product line, the requirements consist of a use case model, a domain model, a commonality and variability analysis, a feature model, and (non-functional) quality attributes.

3.3.5 Architecture

The architecture should attain the qualities prescribed by the requirements. For AGM this means games that operate sufficiently fast to be pleasing and realistic so that the action of the game appears realistic to the player.

The architecture provides detailed models of the architectural structures for the game products. Typically, the architecture is represented as multiple views which

combined form the complete architecture. AGM uses the System Deployment View and Module Decomposition View. As architectural pattern a variant of the Model-View-Controller (MVC) pattern is used.

The resulting architecture has been evaluated using the Architecture Tradeoff Analysis Method (ATAM).

Further discussion of the architecture can be found in section 3.4.3.

3.3.6 Production Plans

A production plan describes the production strategy and provides the means of coordinating the processes attached to each of the core assets. The plan provides an overview of the core assets that are available for product building and how products are built from these assets.

The AGM product line has a generic production plan, and a product-specific production plan for each game, which is attained from the generic production plan by following the attached process.

AGM's production strategy is domain-based design and manual construction and specialization of core assets to form a product. As stated earlier, the core assets are built incrementally. In the early increments more emphasis is laid on identification of candidates for core assets for the later increments.

The production plan also provides management information about scheduling of the product production, the production resources needed (personnel and tools), the *Bill of Materials* (BOM) which specifies the cost per component needed for the product production, product-specific details such as the rules of the game to produce, and software metrics, i.e. "Unique Lines of Code".

See section 3.5 for an in-depth discussion of the production plans.

3.3.7 Test Plans

The AGM product line has a system test plan to describe how the product teams should test a new product and a unit test plan which specifies how core components are tested using automated unit tests. These unit tests include generation of different input values following from the tested component and checking whether the output of the component is conform specification.

The test plan is specialized for each specific game product to only incorporate the components which are used for the product. The results of the tests are documented in a test report, which is input to the asset/product developers for solving bugs.

3.3.8 User Manual

For each game a user manual is provided in the AGM product line. The manual has a fixed layout and is specialized for the product by documenting the specific rules of the game and the expected output of the game.

Also a general installation guide is provided in the manual.

3.3.9 Code Assets

Last but not least, the AGM product line includes the code assets for the three products: Brickles, Pong, and Bowling. The games have actually been implemented and can be downloaded from [35].

See section 3.4.5 for an in-depth discussion of the code assets.

3.4 Software Life-Cycle Phases

This section describes each life-cycle phase from the earlier discussed *two life-cycle model* relevant to the case and how each phase is filled in by the AGM product line. See also section 2.2.4 for the background information.

3.4.1 Domain Analysis

We now focus on the products of this phase in the AGM case: a use case model, a domain model, a commonality and variability analysis, a feature model, and (non-functional) quality attributes.

Use Case Model

The requirements document in the case enumerates 13 use cases divided into general and product-specific cases [35]:

- AGM001: Play the Game
- AGM002: Exit the Game
- AGM003: Change Case: Save the Game
- AGM004: Change Case: Save Score
- AGM005: Change Case: Check Previous Best Score
- AGM006: Play Brickles
- AGM007: Play Pong
- AGM008: Play Bowling
- AGM009: Initialization
- AGM010: Animation Loop
- AGM011: Install Game
- AGM012: Uninstall Game
- AGM013: Set the Speed of Play

Cases 3, 4, and 5 are so-called change cases which describe functionality that has not been implemented yet, but needs to be taken into account. Cases 6 till 8 are clearly product-specific use cases.

The use cases have been worked out in the case. An example use case is included in appendix E.

Feature Model

The case contains a FODA-style feature diagram. The diagram is split up in three parts as can be seen in figures 3.4, 3.5, and 3.6. The hierarchy of features beneath the action and services feature is shown in the second and third diagram respectively.

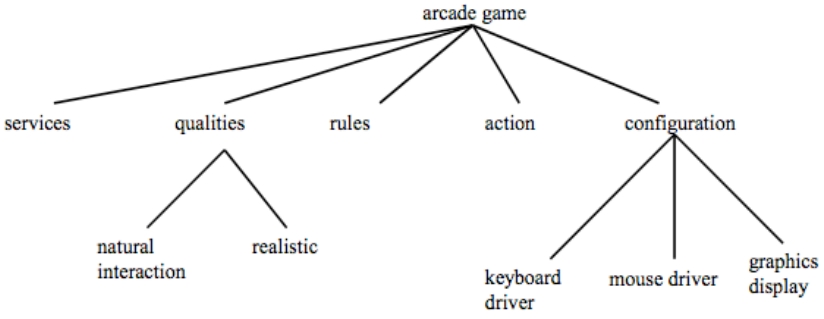

Figure 3.4: AGM Feature Model, part I [35]

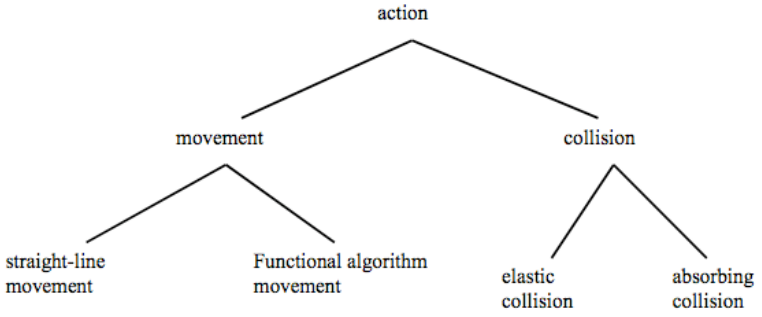

Figure 3.5: AGM Feature Model, part II [35]

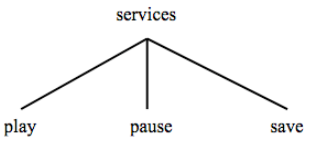

Figure 3.6: AGM Feature Model, part III [35]

3.4.2 Requirements Analysis

The customer requirements are limited in the AGM case to which of the three available products the customer wants to buy/use. The production plan however does foresee in the addition of products to the product line. When a product is added to the product line later on the commonalities and variations with the three existing products need to be analyzed for the identification of existing features that must change for the new game.

For more information on this process the reader is referred to section 3.5.

3.4.3 Architecture Design

The AGM case contains multiple architectures: a *common* architecture for the product line and *product-specific* architectures with the distinct architectural components for each game. This section explores these architectures and their relations.

Architectural Considerations

AGM has examined a number of factors for the architectures which have also been documented for the product line [35]. The underlying development paradigm is object-oriented and therefore the architecture has both *horizontal* and *vertical* dimensions. The vertical dimension corresponds to the specialization relation; the AGM architectures have several inheritance hierarchies as we will see later on. The horizontal dimension corresponds to the association relation which is used for the main structure of each product. Messages follow the association relations and, optionally, exceptions flow back over those links.

In the case a modified version of the Model-View-Controller architecture is used which allows for separation of the state of the system from the logic that presents some of that state to the user. The *controller* provides input to the system. The input is routed either to the *view* or the *model* as appropriate. For example, in the desktop versions of the AGM games, the keyboard and mouse serve as controllers.

The *view*s present information to the system user in a variety of forms [35]:

> In Brickles, the graphical interface of the game has several fixed items on the screen and two movable ones, the puck and the paddle. The mouse controller allows the user to move the paddle while the system determines the movement of the puck. The view has the responsibility to ask the Model for the data it needs to build its presentation. Each view maintains a copy of the game state that is needed for its particular requirements and it contains the graphical data to place the game data on the screen.

A downside of the MVC architecture is that much messaging occurs at every update and that the state is replicated among all views and the model. This can be conflicting with the AGM strategy to be able to run the games on minimal hardware with good performance. Additionally, the need to be able to add extra views to an existing game has a low priority. For these reasons AGM has adopted a variant of the MVC architecture in which the state of the game and the state of the views are blended in various components. The knowledge of the graphics elements is distributed across the system's entities. Each object knows how to participate in the game and how to draw itself on the screen. The container for all the game elements maintains overall control and sequences the updating of the screen.

Another consideration for the game performance is the distinction between moving and non-moving game items, which can greatly reduce the number of items that need to be checked and updated every time step. The AGM team has decided to use a combination of generalization/specialization and parameterization approaches in the game design. As the reader will see shortly, the graphical game items – and their classification as stationary or movable – are defined by specializing existing abstract definitions. Parameterization is used for the instantiation of the container of game components to, among others, pass an event handler.

Common Architecture

The common architecture is product-line-wide and is generic enough to model the basic structure of the game products. In the AGM case, the architecture is discussed from different angles with several *views* [35] as is common in architecture design. We focus on the chosen module decomposition and the generalization/specialization relations.

Figure 3.7 shows the architectural components for the common architecture.

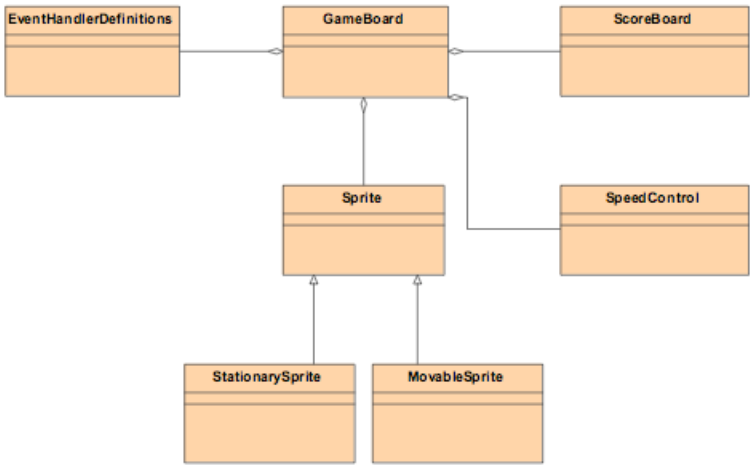

Figure 3.7: AGM Common Architecture

A game product consists of a `GameBoard` which is the earlier-mentioned container of all the other components. Each game in the AGM Product Line consists of so-called *sprites* which are the graphical building blocks of the game. Examples of sprites are the *puck* in each game and the *paddle* in the Brickles and the Pong game. Sprites come in two types: **stationary** and **movable**. A *brick* in the Brickles game is an example of a stationary sprite, the earlier mentioned *puck* is a movable sprite.

Each game contains a `Scoreboard`, for administration of the highscores for the players, and a `SpeedControl` for setting the frequency of *ticks* in the game. Furthermore, each game needs to handle input from the player, such as mouse and keyboard actions, by means of the `EventHandlerDefinitions`.

3.4.4 System Design

Brickles Architecture

As stated earlier, each product line product has its specific architecture based on the common architecture. We start off with the product-specific architecture for the Brickles game. The role of of the production plan in the translation from the common architecture to the product-specific architecture is discussed in section 3.5.

Figure 3.8 shows the product-specific architecture for Brickles. The dark colored items are the architectural components from the common architecture, the lighter colored items are the specialized components for the Brickles game.

The diagram clearly shows that the main variations for a concrete game are in the sprites area: for Brickles, the different graphical elements are either a specialization

of the `StationarySprite` component or of the `MovableSprite` component. The only other variation is the event handling specific to the Brickles game represented by the `BricklesEventDefinitions` component.

The `ScoreBoard` is not specialized for Brickles (and Pong); the score is maintained as a string within the `ScoreBoard` component and consequently allows different games to have different formats for their scores and even to have a different number of scores. For example, Brickles has a single score, Pong has two scores simultaneously, while Bowling must display scores for all 10 frames. Each game is responsible for converting their score to a string. For Brickles, this is handled by the `Puck` component. Whenever a puck collides with a brick and as a result the brick is deleted from the gameboard, the `Puck` raises the score and sends a message with the updated score string to the `ScoreBoard` component which then repaints itself. The more complex scoring rules in Bowling should also be reflected in the scoreboard. As we will see shortly the Bowling game specializes the `ScoreBoard` component for this purpose.

The `SpeedControl` component is general enough to be used in all games, because it only controls the frequency with which it sends *ticks* to the `GameBoard` component for an update of the game display.

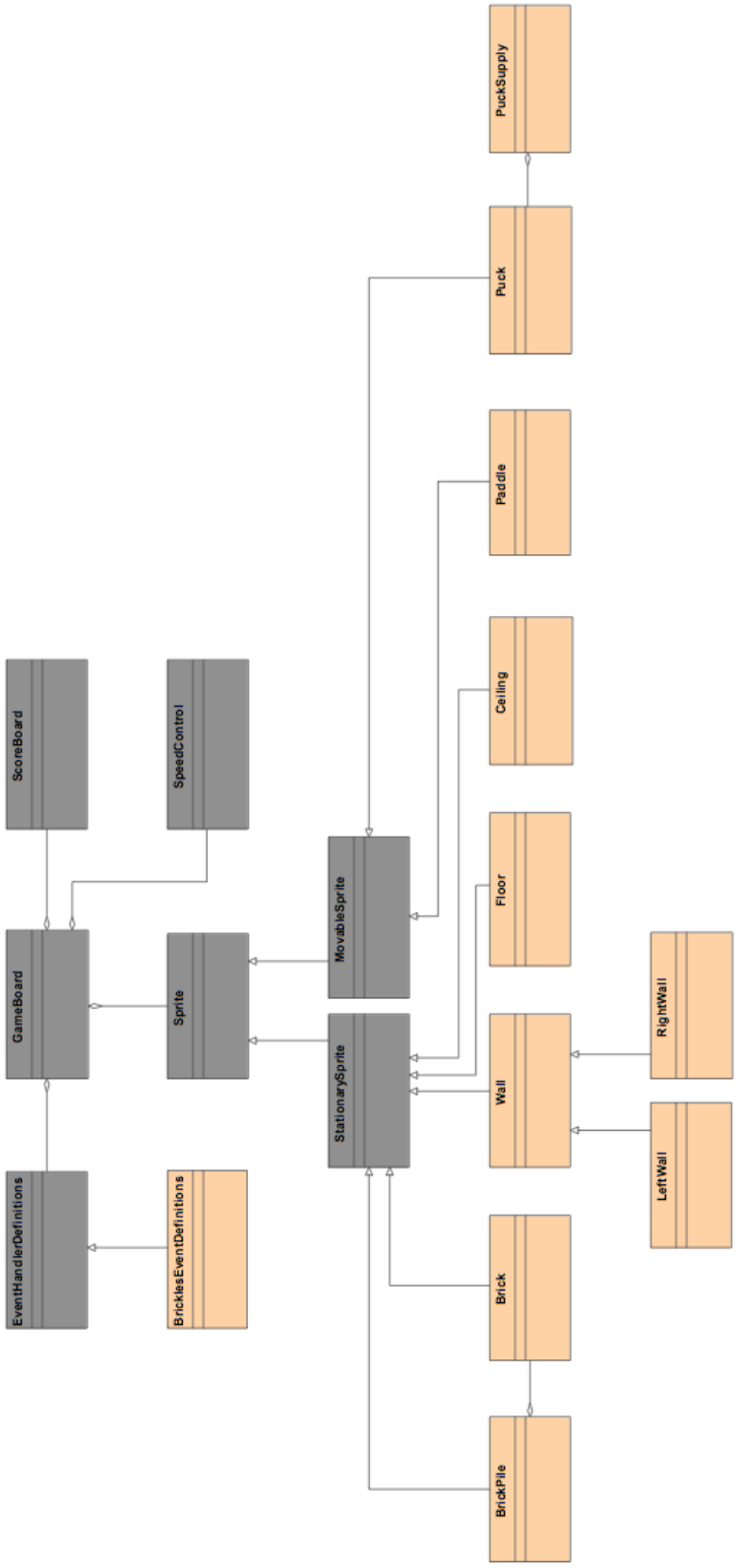

Figure 3.8: Brickles Architecture

Pong Architecture

As can be seen in figure 3.9 the Pong architecture is very similar to the Brickles architecture with that difference that the brick and brickpile sprites have disappeared and the `DividingLine`, `LeftPaddle`, and `RightPaddle` components have been added.

Moreover, the `PongEventDefinitions` component specializes the event handling for the Pong game.

The different game sprites are further discussed in section 3.4.5.

Bowling Architecture

The architecture for the Bowling game is depicted in figure 3.10. The sprites specific to the game are again displayed in the diagram and the event handling is the responsibility of the `BowlingEventDefinitions` component. There are two interesting points in the Bowling architecture, namely the fact that the bowlingball sprite is a specialization of the puck from the other games, and the specialization of the `ScoreBoard` component for the more complex Bowling rules, as discussed earlier.

The `BowlingBall` component looks similar to the puck in the other games, but has a more complex movement algorithm and therefore is a specialization of the `Puck` component from Brickles and Pong.

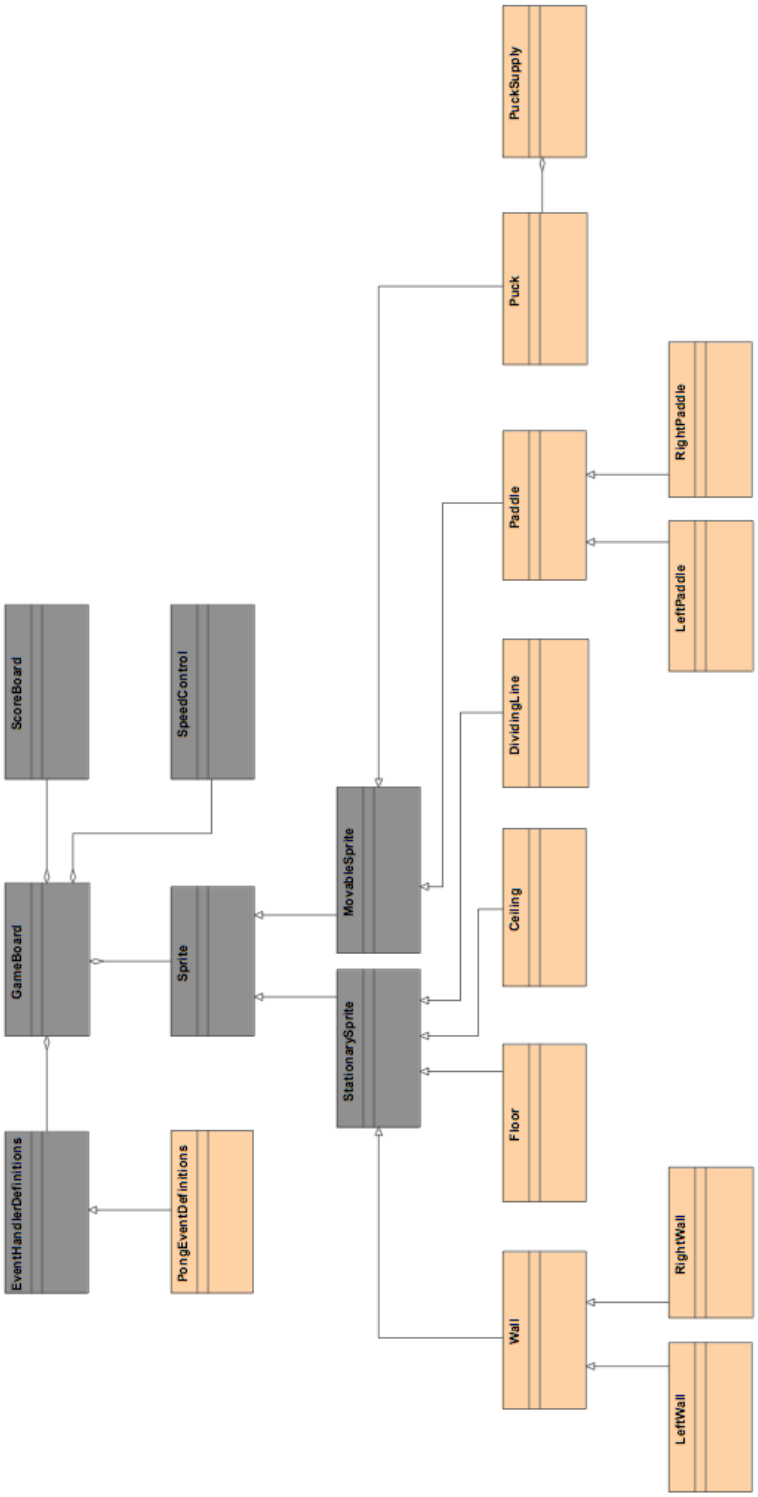

Figure 3.9: Pong Architecture

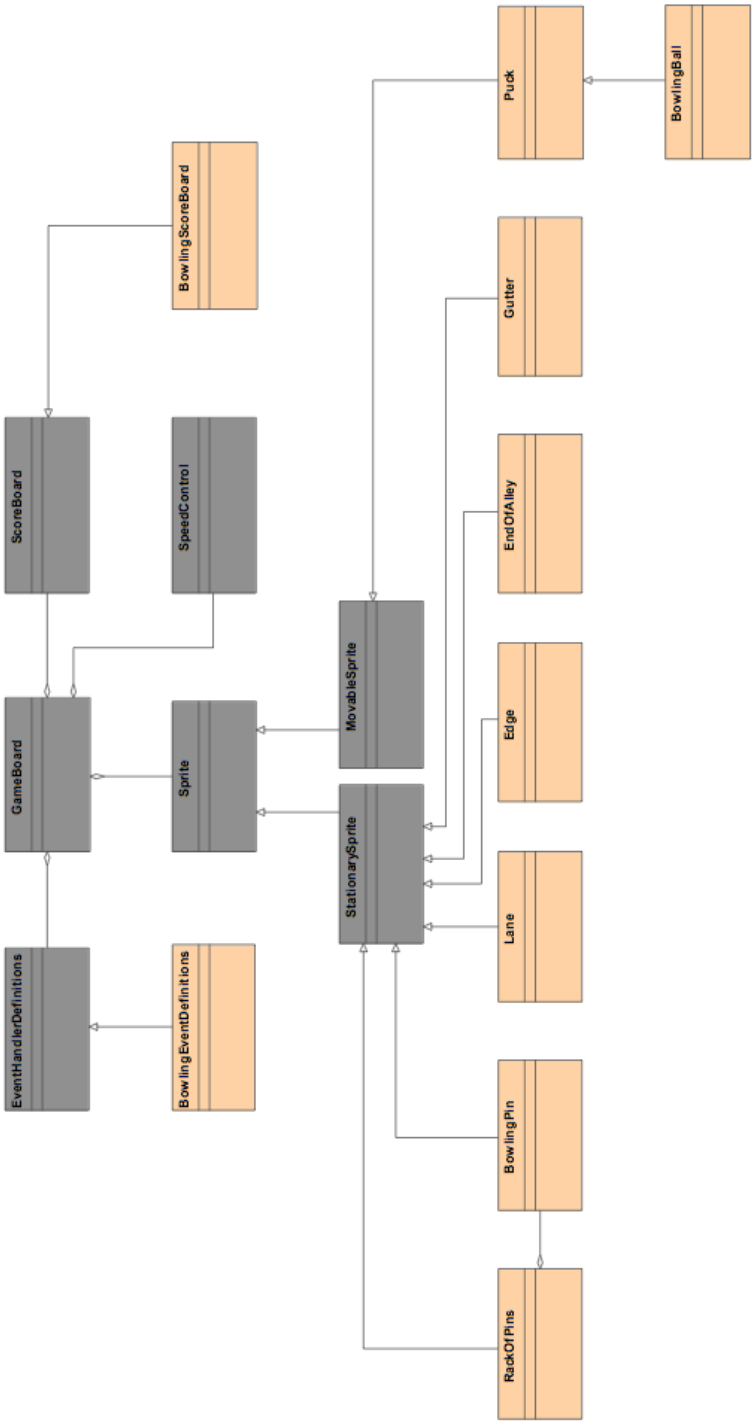

Figure 3.10: Bowling Architecture

3.4.5 Component/Generator Development

After the discussion of the architectures for the AGM case, we now proceed with the code assets for the three games that are divided into a number of packages of which the relations can be seen in figure 3.11.

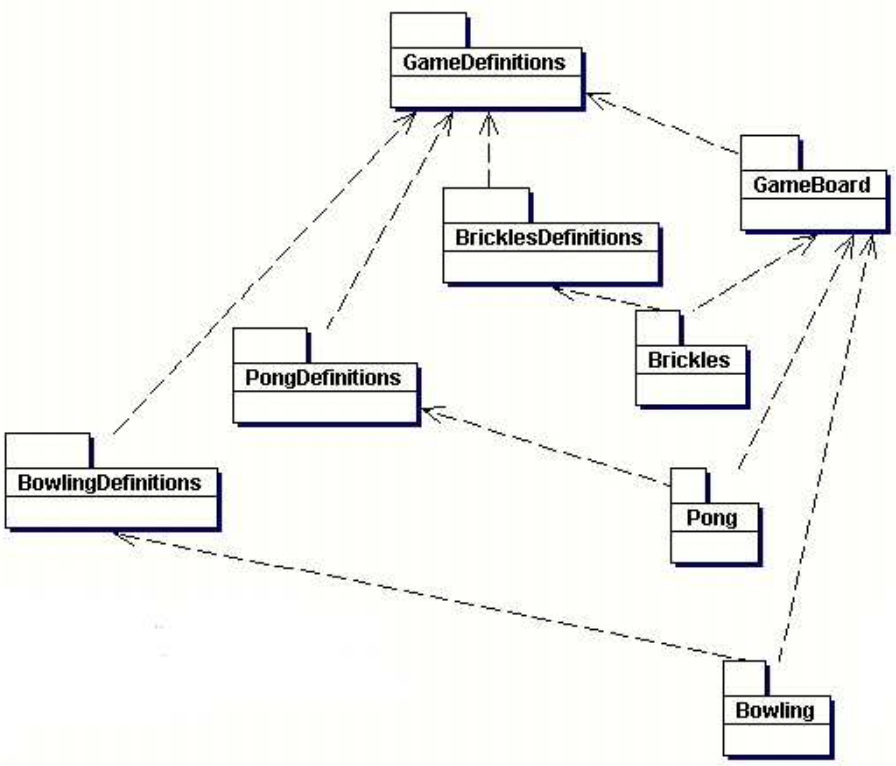

Figure 3.11: Overview of Code Packages [35]

The class diagram for each of these packages has been worked out in the appendix; see chapter B. As one can see, the architectural components are mapped to classes in the implementation.

The main packages are `GameDefinitions` and `GameBoard` which respectively provide functionality for the main game building blocks (like `Sprite`) and the environment in which the games operate (the `Board`). The rules are defined per game by extending the `GameDefinitions` and `GameBoard` package with for example the `BricklesDefinitions` package which provides functionality specific to the Brickles game. Each game further has a main class which couples the components to form the end-program, for example the `BricklesGame` class in the `Brickles` package.

GameDefinitions package

We now discuss the `GameDefinitions` package in detail. For an overview of this package the reader is referred to the class diagram in section B.1 in the appendix.

As described earlier in the architectures, each game in the AGM Product Line consists of *sprites* of two types: **stationary** and **movable**. Stationary as well as movable sprites can be composed to form collections of sprites, for example the

brickpile in the Brickles game. Moving sprites involves direction and speed and the combination of the two is termed *velocity* in the AGM case.

The purpose of the `ContainerEmptyException` class is not clear at the time of writing. The class is however mentioned shortly in the code documentation and it therefore appears in the class diagram. Its purpose might be to throw an exception when a composition of sprites is empty, for example when all the bricks in a brickpile have been broken.

GameBoard package

An overview of the `GameBoard` package is depicted in figure B.2 in the appendix.

The `Board` manages the stationary and movable sprites in the playing field, moves the game pieces (for example, the puck) per *tick* in the game, and detects collisions of game pieces (for example, the puck with a brick). When a collision is detected a `CollisionException` is thrown to provide the necessary behavior.

The package also provides the `ScoreBoard` and the `EventHandlerDefinitions` which is implemented in the different games.

3.4.6 System Development

BricklesDefinitions package

This package provides the functionality for the Brickles game by specifying the sprites in the game, namely `Brick`, `BrickPile`, `Puck`, `PuckSupply`, and `Paddle`. The brickpile and pucksupply are compositions of respectively brick and puck sprites. The boundaries of the playing field are defined in terms of `Wall`, specifically `LeftWall` and `RightWall`, `Ceiling`, and `Floor` (see also figure B.3 in the appendix).

The event handling for the Brickles game is done by the `BricklesEventDefinitions` class, which implements the `EventHandlerDefinitions` interface from the `GameBoard` package.

Three types of exceptions are defined, namely `OutOfBricksException` when the player has broken all bricks in the playing field (and therefore wins the game), `OutOfPucksException` when the player has no more pucks left (and therefore loses the game), and `PuckDeletedException` when a puck is absorbed by a boundary of the playing field, that is if this behavior is activated for the specific boundary (wall, floor or ceiling).

Brickles package

As stated earlier, each game has a main class which ties together all the different components needed to form the actual game. For Brickles this class is the `BricklesGame` class. When starting Brickles all necessary classes are initialized and stored internally in private attributes. The `BricklesGame` class further provides methods for the different interface parts and event handling.

PongDefinitions package

The `PongDefinitions` package is very straightforward. For the Pong game the `BricklesDefinitions` are reused and only a specialization of the Brickles paddle is defined with the `PongPaddle` class.

The event handling implementation for Pong is provided by the `PongEventHandlerDefinitions` class.

Pong package

The main class for the Pong game is `Pong` as can be seen in figure B.6. In the same way as the earlier mentioned `BricklesGame` class it combines the components needed for the game.

This package also includes classes `DividingLine` and `Paddle` for the according game concepts. However, one would expect these in the `PongDefinitions` package.

BowlingDefinitions package

In the same way as the `BricklesDefinitions` package, this package provides the functionality for the Bowling game by specifying the sprites in the game, namely `BowlingBall` (that is the *puck*), `Edge`, `EndOfAlley`, `Gutter`, `Lane`, `BowlingPin`, and `RackOfPins` which is a collection of `BowlingPin` sprites.

The event handling for the Bowling game is done by the `BowlingEventHandlerDefinitions` class, which implements the `EventHandlerDefinitions` interface from the `GameBoard` package.

Two types of exceptions are defined, namely an `EndOfAlleyException` when the bowling ball reaches the end of the alley and possibly pins are knocked down, and an `EndOfGameException` when the 10 frames have been played and the game ends.

For the Bowling game the default score board from the `GameBoard` package is extended by the more complex `BowlingScoreBoard` which implements the scoring rules as discussed earlier.

Bowling package

This package provides the main class `Bowling` for the game. The different components are again initialized and stored in private attributes. To implement the bowling rules also some state information is stored in the class, with for example the `workingOnSpare` attribute.

3.5 Production Plans In-Depth

After the treatment of the code assets we now discuss the information the production plans in the AGM Product Line offer to facilitate the product developers in the process of building end-products through the composition of code assets.

As discussed earlier, the AGM case consists of four production plans: (1) a generic production plan that describes the available core assets, the production process, and how the production plan can be tailored to the product-specific production plan, and (2) this product-specific production plan for each of the three games currently in the product line.

The following sections treat the contents of production plans and how these are filled in by this case. To get an impression of a concrete production plan, the relevant pages of the generic production plan are included in appendix C.

3.5.1 Product Line Scope

When the product developers want to build a product it is important to know if the product *fits* in the product line and it therefore is possible to build the game from the available assets. Implicit to the product line approach is that only a limited number of variations are possible, so there is a limit to the range of products that can be produced.

The AGM production plan states that the product line is intended for a series of arcade games ranging from low obstacle count to high with a range of interaction effects, with similar content as the current games (Brickles, Pong, and Bowling), and availability for different platforms (i.e. PC and handheld devices). These limits are still rather vague and it is therefore left to the product developers in dialogue with the core asset developers to assess if a product is allowed in the product line.

The production plan does specify two possible variations for the different games:

- Variation in behavior when a movable sprite collides with a stationary sprite. The movable sprite is then **absorbed** by the stationary sprite and deleted from the game (for example, in Brickles when the puck collides with the floor), or it is **reflected** by the stationary sprite according to the law of physics.
- Variation in **event handling** for each game by the means of implementing the keyboard and/or mouse events specific to the game.

Furthermore, the product line products should have certain qualities [35]:

> The products must be enjoyable to play in order to be a success. This requires both a colorful display and realistic action.
>
> - Any new game elements should add to the quality of the display. It should be colorful and representative of the item it represents.
> - The action of the game must proceed sufficiently fast to demand the player's attention. When constructing a game, if the number of elements slows the game, alternatives must be investigated.
> - The action of the game must look like what the player expects. The motion and reactions of movable elements must be realistic. As elements are added to the game, their actions and their boundaries must be correctly set through parameters so that collisions appear real.

3.5.2 Assets for Product Production

To produce a product, the product developer should have an overview of the available core assets in the product line. These assets are divided into analysis, design, and implementation assets. The AGM production plan enumerates the *domain analysis model*, the *feature model*, and the *use case model* as analysis-level assets. For more information on these assets the product developer is referred to the requirements document, which is also available in the AGM product line. On the

design-level, the case contains a description of the software architecture which has already been discussed earlier.

On the implementation-level the production plan mentions the code assets (i.e. the C# components which provide the implementation of the architecture components, and the test cases which are divided into *unit tests* (in the DotUnit testing framework) for individual classes, *integration tests* for the combination of classes which form a component, and *system tests* where a set of use cases is tested for the game as a whole. The test cases are documented in a test plan per product.

3.5.3 Production Process

A production plan should also describe the production process to come to the end-products. In the AGM product line the production process for a new product consists of five steps:

1. **Product definition and identification**: AGM has identified three products, which are single games (i.e. Brickles, Pong, and Bowling), to be implemented. For each product the rules of the game are defined.
2. **Incremental analysis**: when a product is added to the product line later on the commonalities and variations with the three existing products need to be analyzed for the identification of existing features that must change for the new game.
3. **Product design**: because of the high skill level of the product developers the design of a new game is described rather limited as [35]:

   (a) Plan how to provide those features from existing components.

   (b) Plan how to provide the remaining features from new assets.

   (c) Design the new implementation of the EventHandlerDefinitions interface.

   As explored earlier in the common and product-specific architectures, the architecture for the new product is based on the common architecture with specializations of the stationary and movable sprites specific to the game, and the above mentioned implementation of the event handling in the game.
4. **Code building**: when writing code for the new game the following steps should be taken into account [35]:

   (a) Start new ClassLibrary in a Visual Studio Project using {game name} Definitions as its name Use this for new classes other than the game definition itself.

   (b) Start new Windows Application in a Visual Studio Project using the name of the game.

   (c) Copy the Form1.cs file from a previous product line project.

   (d) Change the namespace name to the new game.

   (e) Configure the new gameboard.

   (f) Copy data.txt from a previous games working directory. This is the resource file for the game.

   (g) Edit data.txt to reflect the new game.

   (h) Compile the resource file.

(i) Copy the compiled resource file to the Debug directory.

5. **Product testing**: each core asset is tested as it is created or revised using a DotUnit test class per code asset. The initial generic game test set is revised for each new game. In addition a game specific test set is created. The system tests are maintained as text documents and are applied by hand.

3.5.4 Management of the Production Process

As discussed earlier, the production plan provides management information about scheduling of the product production, the production resources needed (personnel and tools), the *Bill of Materials* (BOM) which specifies the cost per component needed for the product production, product-specific details such as the rules of the game to produce, and software metrics, i.e. "Unique Lines of Code".

As can be seen on page 12 of the production plan in appendix C the schedule specifies the tasks for product production per process step, who is responsible for a task, and what is the time estimate on the realization of the task.

The AGM production plan states that the primary resources – next to personnel – needed for product production are the Visual Studio .Net environment and an UML modeling tool. The generic BOM consists of four components: the product-specific Game component, the generic GameBoard, the required Sprites for the game, and implementation of the EventHandlerDefinitions interface. These components are all developed in-house. The rules of the game, which are the most unique parts of the product, are distributed across the Sprites, the EventHandlerDefinitions, and the Game component.

The production plan defines two metrics for the product production process: the number of new lines of code and the number of unique lines of code. The first metric describes the number of lines of new code that has to be written for the new product. The second metric describes the percentage of the product that is code not used in any other product.

3.5.5 Product-Specific Production Plans

The information in the production plan is, in general, very generic and applies to all products built using the current asset base. Some sections of the plan need to be modified for a specific product, specifically the schedule and the bill of materials. In the AGM case, following from the initial manual product production approach, the schedule defines which personnel are needed and when. A copy of the management information provided in the Brickles production plan is included in appendix D.

The case only provides a product-specific production plan for the Brickles game. In this plan the schedule has been taken directly from the generic production plan with no alteration, because the schedule is generic enough to be applied to all products currently in the product line. The bill of materials has been worked out for the Brickles game as can be seen on page 13 of appendix D. It shows which code assets are needed for the realization of the game, what the source is of the code assets (in this case all in-house), and the cost of per code asset in terms of lines of new code.

Testing of the code assets specific to the Brickles game is described in the Brickles test plan which is also available in the case [35].

3.6 Summary

The Arcade Game Maker case illustrates what a concrete product line looks like and how product line engineering aims at reusing different types of software artefacts as assets.

We have investigated the phases in a product line process and how these phases are related. Each phase has been made concrete for the AGM product line.

We finished the case study with a discussion of the different elements of a typical production plan and how products are produced through this plan. Again, the different elements have been illustrated with the production plans from the case and their contents.

Chapter 4

Impact of Crosscutting Concerns

4.1 Introduction

Now that we have an overview of a concrete product line we investigate the impact of crosscutting concerns in this chapter. Before we are going to work with a concrete crosscutting concern we first discuss the main concepts of *aspect-oriented programming* in the background section.

We then introduce the crosscutting concern *Replay Actions* to the case and analyze its functionality, how we could implement it in the 'traditional' object-oriented manner, and which concrete problems occur. Then we abstract from these issues and identify problems with crosscutting concerns for the product line context on both the component level and production plan level.

We finish this chapter with an aspect-oriented implementation of the Replay Actions feature. In the next chapter we explore modularization of crosscutting concerns on both the component level and production plan level.

4.2 Background

Aspect-oriented programming (AOP) modularizes crosscutting concerns on the code level with the concept of an *aspect*, that is *a unit for modularizing an otherwise crosscutting concern* [6]. After modularizing the (crosscutting) concerns using AOP the languages should also support a mechanism to *compose* the concerns. We define *composition* as *the integration of multiple modular artefacts into a coherent whole* [6]. Composition of aspects is sometimes referred to as *weaving*. Van den Berg et al. have created a glossary of common AOSD concepts [6]. Appendix F contains a list of important concepts and terms.

Aspect-oriented programming revolves around three important elements: a *join point model*, a way of *identifying* join points, and a way of *affecting implementation* at join points [30]. In this report we define a *join point* as *a point of interest in some artefact in the software lifecycle through which two or more concerns may be composed* [6]. The *join point model* in an AOP language defines the kinds of join points available and how they are accessed and used.

Identification of a join point is generally related to the concept of a *pointcut*; this is a predicate that matches join points. More precisely, a pointcut is a relationship from "join point" to boolean, where the domain of the relationship is all possible join points.

4.2.1 AspectJ

AspectJ is the de-facto standard for aspect-oriented programming, which is an extension of the popular programming language Java. In AspectJ pointcut designators identify particular join points by filtering out a subset of all the join points in the program flow. The following examples are adopted from [30]. The pointcut designator:

```
call(void Point.setX(int))        ||
call(void Point.setY(int))
```

identifies any call to either the `setX` or `setY` methods defined by `Point`. Syntactically, this code consists of two `call` pointcut designators composed with '*or*'. The syntax of `call` is based on that of Java method signatures. Programmers can define *named* pointcut designators, and pointcut designators can identify join points from many different classes. The following code defines a pointcut named `move` that designates any method call that moves figure elements:

```
pointcut move():
  call(void FigureElement.moveBy(int, int)) ||
  call(void Point.setX(int)                 ||
  call(void Point.setY(int)                 ||
  call(void Line.setP1(Point)               ||
  call(void Line.setP2(Point);
```

The previous pointcut designators are based on explicit enumeration of a set of method signatures; this is called *name-based crosscutting*. AspectJ also allows specification of methods rather than their exact name. This is called *property-based crosscutting*. The simplest of these involve using wild cards in certain fields of the method signature. Others use control flow or other properties to identify join points. Consider:

```
call(void Figure.make*(..))
call(public * Display.*(..))
cflowbelow(move())
```

The first designates any call to methods defined on `Figure`, for which the name begins with 'make' and which take any number of parameters. The second identifies any call to a public method defined on `Display`. The third uses the `cflowbelow` primitive pointcut designator and idenitifies all join point that occur during the execution of methods that move figure elements.

In AspectJ, *advice declarations* are used to define additional code that runs at join points. *Before* advice runs at the moment a join point is reached, or in other words just before the method begins running. *After* advice runs at the moment control returns through the join point, or just after the method has run (and before control is returned to the caller). *Around* advice runs when the join point is reached, and has explicit control over whether the method itself is allowed to run at all. This fater advice adds some code to any join point matched by the `move` pointcut designator.

```
after(): move() {
  <code to be added>
}
```

This concludes our short discussion of the AspectJ language concepts. When written as an aspect the structure of a crosscutting concern is explicit and easy to reason about. Aspects are also modular, making it possible to develop plug-and-play implementations of crosscutting functionality. This is an interesting property of aspects for use in software product line engineering as we will see later on in this report.

4.2.2 CaesarJ

CaesarJ [2] is an aspect-oriented language with a strong support for reusability. It combines the aspect-oriented constructs, pointcut and advice, with advanced object-oriented modularization mechanisms. From an aspect-oriented point of view, this combination of features is targeted at making large-scale aspects reusable – one can say, it enables aspect components. From a component-oriented view, on the other hand, CaesarJ is addressing the problem of integrating independent components into an application without modifying the component to be integrated or the application. Aspect-oriented component models are further discussed in section 5.2.1.

CaesarJ has been developed to solve several structural problems found in AspectJ. These are discussed in [36]. The well-known observer design pattern is used here to clarify the explanation. The AspectJ implementation of all '*Gang of Four*' design patterns can be found in [23].

CaesarJ expresses an aspect as a set of collaborating abstractions. The core of an aspect in CaesarJ is its *collaboration interface* that defines the interaction between the aspect *implementation* and the aspect *binding* to application specific classes and functions. Furthermore, CaesarJ provides runtime aspect deployment on an application.

For example, the collaboration interface for the observer protocol is:

```
public collaboration interface ObserverProtocol {
  public interface Subject {
    public provided void addObserver(Observer o);
    public provided void removeObserver(Observer o);
    public provided void changed();
    public expected String getState();
  }

  public interface Observer {
    public expected void notify(Subject s);
  }
}
```

In the collaboration interface for the observer protocol two interfaces are defined. These interfaces define the layout of the entire protocol which consists of subjects and observers. The `provided` methods are part of the aspect implementation and the `expected` methods are part of the aspect binding.

The aspect implementation of the observer protocol looks like this:

```
public class ObserverProtocolImpl provides ObserverProtocol {
  public class Subject {
    private List observers = new LinkedList();
    public void addObserver(Observer o) { observers.add(o); }
    public void removeObserver(Observer o) { observers.remove(o); }
    public void changed() { ... Notify all Observers ... }
  }

  public class Observer {}
}
```

As one can see, the provided methods from the collaboration interface are implemented here.

The aspect binding gives the implementation that is *application specific* to the usage of the aspect:

```
public class ColorObserver binds ObserverProtocol {
  public class LineSubject binds Subject wraps Line {
    public String getState() {
      return "Line colored "+wrappee.getColor();
    }
  }

  public class ScreenObserver binds Observer wraps Screen { ... }

  after(Line l): (call(void Line.setColor(Color)) && target(l)) {
    LineSubject(l).changed();
  }
}
```

There is one subject `LineSubject` and one observer `ScreenObserver`. It is possible to bind more subjects and observers to the observer protocol in a single binding.

Finally, aspects need to be deployed on the application in question. This is done as follows:

```
public class CO extends
  ObserverProtocol(ObserverProtocolImpl,ColorObserver) {}

public class Test {
  private deployed static final CO co = new CO();
  ...
}
```

Here the aspect implementation `ObserverProtocolImpl` and aspect binding `ColorObserver` are combined and deployed on the application.

Concluding this short discussion, CaesarJ's main goal is to improve aspect reusability by strictly separating the crosscutting feature that an aspect implements and the binding of the aspect to a concrete application context. This solves the problem of hard-coded references in an aspect to the context in which it is deployed.

4.2.3 Composition Filters

The composition filters model is an extension of the conventional object-oriented model through the addition of object composition filters [1][8]. Filters are first class

objects and thus are instances of filter classes. The purpose of filters is to manage and affect message sends and receives. In particular, a filter specifies conditions for message acceptance or rejection, and determines the appropriate resulting action. Filters are programmable on a per class basis. The system makes sure that a message is processed by the filters before the corresponding method is executed: once a message is received, it has to pass through a set of input filters, and before a message is sent, it has to pass through a set of output filters.

Separation of concerns is achieved by defining a filter class for each concern. Each filter class is responsible for handling all aspects of its associated concern. The filter mechanism gives programmers a chance to trap both message receives and sends, and to perform certain actions before the code of the method is actually executed. The resulting code is thus nicely separated into the special purpose concern (in the filter) and basic concern (in the method).

In order to add crosscuttting concerns to one or more objects, the composition filters model provides the *superimposition mechanism*. Superimposition is expressed by a superimposition specification, which specifies how the concerns crosscut each other.

Compose* is an implementation of the composition filters model. Figure 4.1 shows an example tracing concern which is superimposed on two classes on which the feature needs to be applied.

```
1  concern Tracing {
2
3    filtermodule tracingModule {
4      externals
5        log : Log;
6     inputfilters
7        logIn : Meta = (isEnabled => [*.*] log.traceMessage);
8      outputfilters
9        logOut : Meta = (isEnabled => [*.*] log.traceMessage);
10   };
11
12   superimposition {
13     selectors
14       withTracing = { C | i sClassWithNameInList (C, [  Pacman  ,  Ghost ] ) };
15     filtermodules
16       withTracing <- tracingModule;
17   };
18
19   implementation begin in Java;
20
21   public class Log {
22
23     public void traceMessage (Message m) {
24       // tracing functionality here
25       // ..
26       // continue evaluating this message
27       m.fire( );
28     }
29   }
30   end;
31
32 };
```

Figure 4.1: Example of superimposed concern in Compose*

This example shows a concern that specifies a filtermodule `tracingModule` that filters every incoming and outgoing message, reifies it and passes it to an external

of type `Log`, which will log the incoming or outgoing message (the logging part itself is unspecified here).

The superimposition clause specifies on which instances of classes this filtermodule is superimposed. In this case, the filtermodule `tracingModule` is superimposed on all instances of classes `Pacman` and `Ghost`.

Summarizing, the composition filters approach is a different way to think about join points, pointcuts, and deploying aspectual behavior on an application. The underlying aspect-oriented concepts however are similar.

4.3 Analysis

4.3.1 Crosscutting Concern Replay Actions

As we have seen earlier, the AGM product line consists of three games: **Brickles**, **Pong** and **Bowling**. Now let's say we want to introduce the optional feature *Replay Actions* for the games, that is the state of the game is saved regularly during the game and the player can rollback the state of the game by a certain amount of time to replay his actions from that point. Such a feature is strongly related to the well-known crosscutting issue of *Persistence*.

The player can use the *Replay Actions* feature by clicking a button in the game interface when he has made an insensible move and wants to correct his last actions. We leave in the middle if this feature makes the games more fun or not.

The feature requires that the *game state* is saved regularly. The game state for the games basically consists of:

- Current position of every relevant sprite in the game
- Current velocity of relevant movable sprites
- Current game score
- Current game speed

We will discuss the relevant data per game in a moment.

The game player can rollback the game a few actions at any moment, so we decide to save the game state at every collision of the puck with the paddle, for Brickles and Pong, or at the start of every new frame for the Bowling game.

When the player uses the replay feature the game state is emptied and the player isn't able to use the feature again directly; the replay button is therefore deactivated. After a few actions we have collected new game states and the replay button is then reactivated.

4.3.2 Implementing Replay Actions feature

The implementation of the Replay Actions feature consists of the following steps:

1. Collect the current game state when a collision of the puck with the paddle occurs, or a new bowling frame begins.

2. Write the current game state to a database; the database contains the last n saved game states, where n is configurable for the feature when producing the game-product.
3. Delete the oldest game state from the database table, if the database contains *n+1* game states; the collection of game states in the database is a *sliding window*.
4. When the player presses the Replay button, the first game state in the database is looked up and loaded into the game. The Replay button is then deactivated and the database table is emptied.
5. After n saved game states have passed the Replay button is reactivated.

We now focus on collecting the game state per game. See also section 3.4.5 and appendix B for more information on the class structure in the AGM case.

Brickles

For the Brickles game we want to save the following data to be able to restore the game state:

- The `visible` property for each stationary sprite `Brick` in the `BrickPile`. If the puck collides with a brick in the brickpile, the brick's visible property is set to false and hence the brick is 'deleted' from the game board.
- The return value of the `numleft` method of the `BrickPile` object which is the number of bricks left in the brickpile. If the `numleft` value reaches 0 an `OutofBricksException` is thrown and the player wins the game.
- The properties `MyLocation.X` and `MyLocation.Y` for the position of the movable sprites `Puck` and `Paddle`.
- The return values of `MyVelocity.speedX()` and `MyVelocity.speedY()` for the speed of the puck and the paddle.
- The return value of `MyVelocity.getDirection().getDirection()` for the direction of the puck and the paddle.
- The return value of `ps.numberLeft()` for the number of pucks left in the puck supply. The number of pucks left is the score in the Brickles game.
- The return value of `b.getSpeed()` for the game speed.

Pong

The game state for the Pong game is collected as follows:

- The properties `MyLocation.X` and `MyLocation.Y` for the position of the movable sprites `Puck` and `Paddle`.
- The return values of `MyVelocity.speedX()` and `MyVelocity.speedY()` for the speed of the puck and `MyVelocity.SpeedY()` for the two paddles.
- The return value of `MyVelocity.getDirection().getDirection()` for the direction of the active paddle.

- The return value of `ps.numberLeft()` for the number of pucks left in the puck supply.
- The property `whoseTurn` through which the game keeps track of which of the two paddles is active.
- The properties `scoreTrue` and `scoreFalse` which hold the scores for the left and right player respectively.
- The return value of `b.getSpeed()` for the game speed.

Bowling

For the Bowling game we want to save the following data to be able to restore the game state:

- The `visible` property for each stationary sprite `BowlingPin` in the `RackOfPins` stationary sprite. If the bowling ball collides with a bowling pin in the rack, the pin's visible property is set to false and hence the pin is 'deleted' from the game board.
- The return value of the `numleft` method of the `RackOfPins` object which contains the number of pins left in the rack.
- The properties `MyLocation.X` and `MyLocation.Y` for the position of the movable sprite `BowlingBall`.
- The return values of `MyVelocity.speedX()` and `MyVelocity.speedY()` for the speed of the bowling ball.
- The return value of `MyVelocity.getDirection().getDirection()` for the direction of the bowling ball.
- The `score` array which holds the score per frame.
- The `frame` property for the current frame number.
- The game speed cannot be controlled in the Bowling game and therefore doesn't need to be saved.
- The properties `workingOnSpare` and `workingOnStrike` for keeping track whether a spare or strike was thrown in the previous frame. In the next frame the first ball score or first and second ball score are then added to the previous frame score in case of a spare or strike respectively.

A comparison of the game state elements among the three games is depicted in table 4.1.

Game state element	Brickles	Pong	Bowling
`visible`	X		X
`numLeft()`	X		X
`ps.numberLeft()`		X	
`MyLocation.X` & `MyLocation.Y`	X	X	X
`MyVelocity.speedX()` & `MyVelocity.speedY()`	X	X	X
`MyVelocity.getDirection().getDirection()`	X	X	X

`b.getSpeed()`	X	X	
`whoseTurn`		X	
`scoreTrue` & `scoreFalse`		X	
`score`			X
`frame`			X
`workingOnSpare` & `workingOnStrike`			X

Table 4.1: Similarities and differences in game state elements

As you can see, some game state elements are shared among games and some are specific to a game.

If we now look **when** we are going to collect and save the game state this differs per game. As discussed earlier the game state is saved when the puck collides with the paddle, or when a new bowling frame starts. The methods that are called when these events occur are:

For **Brickles**: `Paddle.collideWith()`
For **Pong**: `Paddle.collideWith()` or `PongPaddle.collideWith()`
For **Bowling**: `Bowling.newFrame()`

To incorporate the *Replay Actions* feature in the product line games we can add the methods `saveGameState()` and `loadGameState()` to the main class of each game, and call the `saveGameState()` method from within the above mentioned methods. The `loadGameState()` method needs to be connected to a button in the game interface and sets a number of properties and calls various `set`-methods to restore the game state which is loaded from the database.

When a new game is introduced to the AGM product line and the *Replay Actions* feature needs to be implemented for the game, the developer can follow a number of steps:

- Specify the new game state elements.
- Adapt the `saveGameState()` and `loadGameState()` methods for the new game and add these to the main game class.
- Identify when the game state needs to saved or loaded and call `saveGameState()` and `loadGameState()` from the appropriate methods.

4.3.3 Problems with 'Traditional' Implementation

Now that we have an idea of the 'traditional' way of implementing the Replay Actions feature, we see that a number of problems occur:

- The `saveGameState()` method call is *tangled* with the code that handles collisions or starts a new frame in case of Bowling. This makes the code less readable for the programmer.
- For Pong the `saveGameState()` method call is *scattered* among the collision handling methods for both paddles. Because we work in a product line context and the Replay Actions feature is optional, the `saveGameState()` method call

needs to be deleted from or added to each collision handling method by the product developer each time a game product is built. This creates extra work for the product developer and is error-prone, because – especially when a concern is scattered over many other concerns – the addition or deletion of all method calls is easily forgotten.

- The game state elements *vary* per game product. This means that the `saveGameState()` and `loadGameState()` methods needs to be created for each game and thus multiple versions of the Replay Actions implementation need to be saved, that is one version per game. This leads to redundancy in the Replay Actions implementation and thus more difficult evolution, because when the Replay Actions feature needs to be modified all versions of the feature need to be changed. This again leads to extra work for the developers and is error-prone, because one version is easily overlooked in the evolution process.

Summarizing, these problems are caused by the fact that the Replay Actions feature is a crosscutting concern and that the functionality of the feature varies per game, namely which elements are needed in the game state.

Our intuition for dealing with crosscutting concerns is to modularize these in aspects. In the next section we generalize the problems with crosscutting concerns and focus on the product line context. Then we investigate the aspect-oriented implementation of the Replay Actions feature in section 4.3.6.

4.3.4 Crosscutting Concerns for Production Plans

Now that we have investigated the problems with crosscutting concerns on the component level for the concrete case of the Replay Actions feature, we abstract from these issues in this section and focus on the product line context. We also discuss which problems occur with crosscutting concerns on the production plan level.

In our discussion we will use the term *crosscutting feature*. This is a crosscutting concern that represents a reusable, configurable requirement in the context of domain engineering (see section 2.2.4). Recall that a software product line contains a predefined set of features which is implemented by the code assets in the product line.

Component Level

Modularization
On the component level, crosscutting concerns are mapped to multiple code assets / components which jointly provide the desired behavior. These components contain hard-coded references to each other, i.e. to do a method call or set a property. This strong coupling of components reduces *reusability*, because when a component is used in a different context (read: a different product) it still depends on the other components to provide the concern, while the other components may not be suitable for the different context.

Hard-coded references among components also reduce *ease of evolution*, because when the functionality of a crosscutting concern is changed, all components implementing the concern potentially need to be modified to cope with the changes.

Our intuition for dealing with crosscutting concerns on the component level is to implement crosscutting concerns as aspects in the asset library. An aspect then is

a separate asset in the product line and is selected for use in an application when this product should contain the crosscutting concern that the aspect implements.

Configuration
Each code asset has an *attached process* that describes how the asset should be used when developing a product line product (see section 2.2.3). Therefore the attached process for aspects needs to be considered. As any feature a crosscutting feature can have a number of variations, i.e. alternative subfeatures. These variations are implemented in the aspectual asset along with the common behavior.

Recall that when feature selections have been made for a product all variability has been removed, because for each variation point only a single variation is selected. This implicates that the selected variation for a crosscutting feature should be reflected by the aspect to which it is mapped.

An aspect which contains variability should therefore be *configured* for the specific context – the selected variations – in which it is used.

Composition
Another issue is the composition of aspects with other code assets. Because we work in a product line context the features selected for a product can differ strongly and consequently the selection of assets from the library. Aspects do not implement crosscutting behavior on their own: the aspect needs to be *bound* to the relevant assets to deploy the crosscutting feature.

As we know aspectual behaviour is bound to the specific context through a *pointcut specification*, which matches the relevant join points. In the product line approach these join points can differ per product. As a result, the pointcut specification needs to be defined for each aspect for the product line product. This is again the task of the product developer.

A different problem is when aspects depend on each other to provide a certain behavior. For example, every time someone wants to print a document the person needs to be granted access to the printer by providing the right credentials and the printer should then be locked during the print job. When the print job is finished the printer is unlocked and a new job can be accepted. When the access control and locking feature are both implemented as an aspect, these aspects should work together to provide the right behavior, i.e. the locking aspect cannot execute its behavior when the access control aspect has not run yet.

These dependencies among aspects need to be dealt with. Related to these dependencies is the concept of a *shared join point*, which is a join point one which two or more aspects are deployed. This leads to a number of other composition issues, which go beyond the scope of this report. For more information the reader is referred to [40].

Production Plan Level

The product line has a predefined set of features of which each product has a subset of these features. When talking about features we need to make a distinction between *common* features and *variable* features. Common features are mandatory features that all products have, variable features are optional, alternative or or-features that are selected for some products, but are not included in other products. This means that the commonality in the product line is represented by the *common* features and the variability is represented by the *variable* features.

Furthermore, a feature can contain a number of *subfeatures* which in turn can be mandatory, or, optional or alternative. An example of such a feature hierarchy in the AGM case is discussed in the next section.

The production plan specifies the production process which abstractly consists of the following steps:

1. Select all common features for the product and for each feature select the direct and indirect sub-features, if applicable.
2. Select the variable features for the product and for each selected feature select the direct and indirect sub-features, if applicable.
3. Compose the selected features to form the end-product.

This process is also illustrated in the next section.

This means that the product developer needs to 'configure' each selected feature by choosing the relevant direct and indirect sub-features for the product. The production process is worked out in more detail in the production plan. The above mentioned steps implicate that the variability in the product line, that is the variable (sub-)features, is *scattered* throughout the production plan. The variable features are specified in the context of their parent- and sub-features and the variability in the product line is therefore not modularized.

The scattering of variable features can become a problem when working with large product lines with hundreds or even thousands of features. It then becomes hard to keep track of the variability in the product line. Modularizing the variability in the production plan can tackle this problem. In section 5.3.2 we discuss how aspect-orientation can help in the production plan.

4.3.5 Case Example of Crosscutting on Production Plan Level

We now show a concrete case example of crosscutting on the production plan level. As we have seen, a feature can contain a number of *subfeatures* which in turn can be mandatory, or, optional or alternative. In the AGM case a mandatory feature is *collision handling* which takes care of the behavior when in the game the puck collides with another object. This feature has the sub-features *absorb* or *reflect* which are alternative. So each game has the *collision handling* feature, but the product developer needs to choose between the *absorb* and *reflect* features. Subfeatures can in their turn have subfeatures, which leads to a hierarchy of features.

Let us look at a part of the feature model for the AGM case as depicted in figure 4.2.

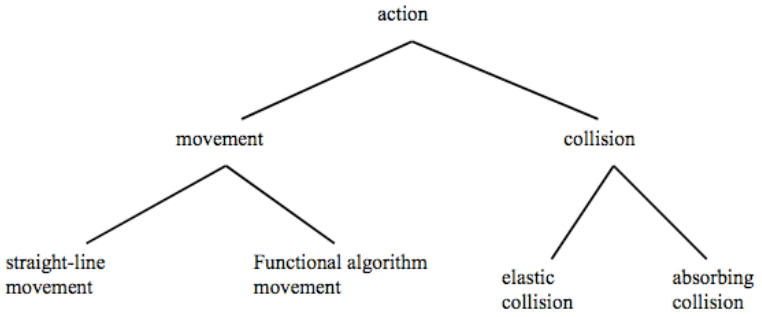

Figure 4.2: Part of AGM Feature Model [35]

A part of the concrete production process steps for the developer to follow would be:

1. Select the common feature *action*
2. (a) Select the common feature *movement*
   i. Select one variable feature from the alternative features *straight-line movement* and *functional algorithm movement*
   (b) Select the common feature *collision*
   i. Select one variable feature from the alternative features *elastic collision* and *absorbing collision*

It is clear from this example that the variable features (for *movement* and *collision*) are scattered among the production process steps. This can become a problem when working with large product lines with hundreds or even thousands of features. It then becomes hard to keep track of the variability in the product line. Modularizing the variability in the production plan can tackle this problem.

The Arcade Game Maker product line is an example of a 'traditional' product line. The games are built from C# classes and the variability in the production plan is scattered, because for example the variations in collision handling and game rules are not grouped in the production process. The AGM product line both has features that map to one code asset and features which are implemented by multiple assets working together to provide the desired behavior. For example, the *scoreboard* feature is implemented by the single class `ScoreBoard`, while the earlier mentioned *collision handling* feature is scattered amongst the classes `Board` (for collision *checking*) and all `Sprite` classes which realize the behavior when a collision has been detected.

The fact that the crosscutting feature *collision handling* is not modularized causes reduced evolution and reusability, which contradicts the goals of product line engineering. For example, introduction of a third variation for the collision handling feature affects all the classes involved in the implementation of this feature, which makes the evolution rather time-consuming.

4.3.6 Replay Actions feature as Aspect

As we have seen in section 4.3.2 better separation of concerns is needed for the crosscutting concern Replay Actions. With our background knowledge on aspect-oriented programming we now try to modularize the Replay Actions feature as an aspect. The pseudo code for the aspect is listed in figure 4.3. We use AspectJ syntax, because most readers will already be familiar with this, although the code assets in the AGM case are written in C#.

```
1  public aspect ReplayActions {
2
3    pointcut saveGameState():
4      call(void Paddle.collideWith())     ||
5      call(void PongPaddle.collideWith());
6
7    before(): saveGameState() {
8      // Game state collection and save code here
9      ...
10   }
11
```

```
12    pointcut loadGameState(ActionEvent e):
13      call(void ActionListener.actionPerformed(ActionEvent));
14
15    after(): loadGameState(e) {
16      if ( ((JButton) e.getSource()).getText() == "Replay Actions") {
17        // Game state restore code here
18        ...
19      }
20    }
21  }
```

Figure 4.3: Replay Actions Aspect

Line 1 defines the aspect `ReplayActions` which specifies the *pointcut* `saveGameState()` in lines 3 - 6 for the *join points* for the Pong game at which we want to add behavior. Line 8 - 11 is the *before advice* which adds the game state collection and save code at the join points, before the code of the method is executed. Lines 13 and 14 describe the *pointcut* `loadGameState()` to monitor the Replay Actions button. Line 16 - 21 specifies the *after advice* for the `loadGameState` pointcut, which checks whether it is the Replay Actions button that is clicked and adds the behavior for retrieving the game state from the database and restoring it in the game.

With this approach we see that the scattering and tangling problems don't occur any more, because the implementation is now cleanly modularized and the concerns on which the Replay Actions feature is deployed are *oblivious* of the crosscutting concern. This is an important property of aspect-orientation. The variation problem is still present, because the advices for the `saveGameState` and `loadGameState` pointcuts differ per game and thus multiple versions of the aspect are still needed.

4.4 Summary

We started this chapter with an introduction to the main concepts of aspect-oriented programming and how the AOP languages AspectJ, CaesarJ and Compose* have implemented these concepts.

We then introduced the crosscutting concern Replay Actions to the case, worked out the implementation of the crosscutting concern in the traditional object-oriented manner. We identified three concrete problems on the component level for this implementation: scattering of the Replay Actions concern over multiple concerns, tangling of the Replay Actions concern with other concerns, and that a specific implementation of the concern is needed for each game because the game state elements vary per game product.

After this discussion we identified the more general problems with crosscutting concerns that occur on both the component level and the production plan level in the product line context. On the component level these problems are modularization of a crosscutting concern, configuration of aspects for specific variations, and composition of aspects with other code assets without context-specific information in the aspect implementation. On the production plan level we have seen that variable features are scattered throughout the production plans and these need to be modularized. This problem is illustrated with a case example.

We finished this chapter with a modularization of the Replay Actions concern in an aspect-oriented implementation. The problem with variation of game state elements was still present in this implementation. In the next chapter we investigate solutions to the identified problems with crosscutting concerns in the product line context.

Chapter 5

Modularizing Crosscutting Concerns for Production Plans

5.1 Introduction

As we stated earlier in our problem statement, the goal of this report is to investigate problems that arise in product line production plans when dealing with crosscutting concerns and to apply aspects for modularization of these concerns. This chapter is our approach to the problems with crosscutting concerns on both the component level and production plan level as we have identified in the previous chapter.

We start with additional background information on component based engineering, aspect-oriented component models, and aspect *configuration*. We will use this background knowledge later on in this chapter in the discussion of applying aspects to the product line context.

To get more understanding about incorporating aspect-orientation in product lines we then investigate different combinations of aspect-orientation in the production plans and asset library.

After this discussion we explore solutions to the identified problems on both levels and then apply these solutions to our case study.

5.2 Background

5.2.1 Component Based Software Engineering

Strongly related to software product line engineering is the research area of *Component Based Software Development* (CBSE). This topic studies the specification of components in so-called *repositories* and the composition of components to form the required application.

As discussed earlier in the product line section, product line products are composed from the core assets and the product developer should somehow know which features and interfaces assets/components provide and how these components interact with each other. When we look at the goals of component based software engineering –

that is high quality and low cost software design and development through software reuse – more parallels can be drawn.

To understand more about the production of product line end-products from predefined, *plug&play* components we investigate the specification and composition of these components in more detail in this section. We also study a number of *aspect-oriented* component models.

Component Specification

Component specification aims to provide a basis for the development, management, and use of components. The specification of a component generally consists of four parts, all regarding the *interface* that the component provides [22][56]:

- an interface **signature** that comprises *properties*, *operations* and *events*. The properties are the externally visible structural elements of the component and are commonly used for customization and configuration of the component at the time of use. The operations capture the service/functionality that the component provides and are the means with which the outside world interacts with the component. Through its operations a component can be controlled *proactively* by the system; the alternative is *reactive* control through events that are generated by components from time to time, which other components in the system may choose to respond to.
- interface **constraints** that limit the properties and operations of a component interface, in addition to the constraints imposed by their associated types. Constraints are of one of two types: regarding individual elements (for example, range constraints on properties) or concerning the relationships among elements (for example, constraining operation invocation on the occurrence of a specific operation invocation before it).
- interface **packaging and configuration** to relate the component to a context of use, that is a *use scenario*, which defines the interaction between the component and related components and states which *interface configuration* is required for this use. A component may play different roles in a given context, and the component can be used in different types of contexts. The interaction between the component and other components can differ depending on the components and their related perspectives (for example, during interaction with a particular type of component, only certain properties are visible and operations available). These perspective/role-oriented interaction protocols are defined in packaging configurations for the component interface.
- **quality attributes** address the non-functional properties of the component, such as security, perfomance and reliability. Because components are often *blackboxes* to the product developer, these qualities are important to get an idea of the qualities of the resulting product. How these quality attributes should be characterized and how the impact on the enclosing system can be analyzed is beyond the scope of this report.

A component specification is described in some formalism, commonly called a *Component Definition Language* (for example, OMG IDL [42]). The specification is often an XML formatted document. See figure 5.1 for an example interface specification.

```
1 <IDL>
2   <comment> Account Example </comment>
```

```
3   <module name="Online_Account">
4     <exception name="OverDraft">
5       <member name="message" type="STRING"/>
6     </exception>
7
8     <interface name="SavingsAccount">
9       <inheritance ancestor="Account"/>
10      <attribute mode="readonly" name="balance" type="Money"/>
11      <signature mode="twoway" rtype="void" name="deposit" >
12        <argument mode="in" name="amount" type="Money"/>
13      </signature>
14      <signature mode="twoway" rtype="boolean" name="withdraw" >
15        <argument mode="in" name="amount" type="Money"/>
16         <raises>
17           <exception name="OverDraft" />
18         </raises>
19      </signature>
20    </interface>
21  </module>
22 </IDL>
```

Figure 5.1: An example XML specification of an account interface [27]

This example defines the interface for a `SavingsAccount` component, which is a subcomponent of the component `Account`. Two signatures for the component are defined for depositing and withdrawal of money to/from the account. In the case of withdrawal an exception can be thrown when not enough money is present on the account.

Component Composition

To (re)use a component in a system – that is in composition with other components – it often needs to be made suitable for the given context. Letting a component fit in its environment falls in the following three categories [56]:

- **Component Customization**: letting the component users choose from a fixed set of options that are already pre-packaged inside the software component.
- **Component Adaptation**: the component users adapt a software component by altering existing functionality. This implicates that the user needs to understand the complex behavior and functionality of its classes, so any change in either of them will not break the structure of the system specified by the component designer. However, most components are blackbox, so – without understanding of the inner workings of the component – the modification options are limited to the visible properties and operations.
- **Component Generation**: the component users generate a software component by accepting a configuration description and assemble the concrete components according to this description.

Aspect-Oriented Component Models

Different component models exist today for offering an architecture for the specification and composition of components. Well-known commercial component models

are the *Corba Component Model* (CCM) [42], *Enterprise JavaBeans* (EJB) [49] and the *Distributed Component Object Model* (DCOM) [38].

Recent work has been done on the combination of CBSE and AOSD to cope with the appearance of crosscutting when describing the necessary dependencies among components and the implementation of these dependencies using 'glue code' [13][45][52]. As discussed earlier, AOSD can resolve the crosscutting issue.

How Crosscutting Arises

We now first look a little bit deeper into how crosscutting arises in 'conventional' component models which causes reduced reusability and adaptability of components.

To build an application from *plug&play* components the product developer needs the earlier-mentioned interface signature and constraints to understand which interfaces each component *provides* and *requires*. Furthermore, the developer chooses one of the packaged roles for each component and configures the packaged properties for the context in which the component is deployed.

This seems a very elegant way to build an application. A downside however is that when a component A declares that it uses the services offered by a component B through its *required* interface, that declaration affects the implementation of A, because direct calls to B methods appear in the code and they are hard-coded. Consequently, changes in business rules involve updating both the specification and implementation of software components. In addition, specification changes affect the packaging and configuration of the components.

This means that component systems are not easily adaptable to new requirements, because the introductions of new requirements involve changes in the entire component specification (signature, constraints, packaging/configuration, and quality attributes) [13].

In the following sections we describe two approaches for aspect-oriented component models.

Aspect Component Based Software Engineering
Clemente & Hernández propose what they call *Aspect Component Based Software Engineering* to provide better flexibility, adaptability, and reusability for component-based applications [13].

They propose a different way to specify component dependencies avoiding the crosscutting at the implementation phase of a component. They classify component dependencies as *intrinsic* and *non-intrinsic* [13]:

- A dependency is *non-intrinsic* when its use depends on the framework or the context in which a component is to be used. That is, if we delete the dependency from the component description, the component maintains its initial functionality without those facilities that the deleted dependency provides.
- A dependency is *intrinsic* when its description and use is vital for the component itself. In other words, if this dependency is deleted then the component loses its meaning.

During component implementation only the interfaces it provides and its *intrinsic* dependencies should be implemented. This means that each component only implements the basic business rules and therefore crosscutting is not being introduced in the component implementation.

When the component is packaged its *non-intrinsic* dependencies are specified in an XML-structured file and implemented using aspects. When composing the end-system from the components, the component interfaces and the *intrinsic* and *non-intrinsic* dependencies are used to form the final product.

JasCo
Suvée et al. also studied the combination of aspect-orientation and component based software engineering (CBSE) and they introduce a novel aspect-oriented implementation language and an aspect-oriented component model based on enterprise java beans (EJB), both called *JasCo* [52].

They identify several problems with the integration of AOSD with CBSE [52]:

- The deployment of an aspect within a software-system is static in that the aspect looses its identity when it is weaved into the application code. This makes it difficult to extract an aspect from a composition and replace it afterwards with a different aspect.
- Aspects are often described with a specific context in mind, which makes it hard to reuse aspects.
- Communication between the components in an application is specific to the employed component model. JavaBeans makes use of the event-model and current AOSD technologies are not suited to deal with this kind of interaction.

They therefore introduce the *JasCo* language on top of Java with two new concepts: *aspect beans* and *connectors*. *Aspect beans* are used for describing some functionality that would normally crosscut several components from which the system is composed. An aspect bean normally holds one or more *hook-definitions* as a combined *join-point/advice* from AspectJ and a *hook* is used to specify when the execution of a component should be 'cut' and what extra behavior should be executed there.

Connectors are used to deploy aspects within an application and contains one or more hook-initializations, zero or more behavior method executions, and any number of regular Java constructs. A hook-initialization is identical to a Java class instantiation and takes one or more method signatures as input to convert abstract parameters from the hook-definition to concrete parameters. Connectors can also control how multiple aspects are deployed: simultaneous or sequentially, which means that the precedence of aspects is defined in the connector. For advanced aspect combinations *JasCo* offers so-called *combination strategies*.

The *JasCo* component model implements the 'aspect-enabled' EJB component model in which 'normal' beans can still operate by converting them to JasCo components. Aspects are deployed at run-time by the platform by using a connector registry and providing the ability to dynamically load and unload connectors. This improves reusability and adaptability of the components and component composition, because no recompiling is needed after adding or removing aspects, as opposed to static weaving at compile-time.

5.2.2 Aspect Configuration

As we will see in the next chapter we would like to view aspects as separate components in a software product line. Like with components, as we have just investigated, variability and configurability should be supported for aspects to offer the functionality specific to a product line product. For example, when we want to use the crosscutting feature *caching* in our product and *caching* is implemented by an

aspect, the functionality may vary in the size of the cache, the percentage of the cache to delete when the cache is full, the type of messages being cached (strings, documens, etc.), and so on. The aspect in question needs to be configured for the specific variation.

A technique for dealing with the configuration of aspects is called *framed aspects* [33][34]. Traditional approaches mainly focus on the classic categories of evolution namely, *corrective* (fixing of bugs), *adaptive* (adding a new feature), *perfective* (improving performance), and *preventive* (preventing problems before they occur). While this categorization is useful in showing the *type* of evolution to be performed, it does not demonstrate how the change affects the software architecture itself. In order to suppport this it can be more useful to think of *crosscutting* and *non-crosscutting* evolution.

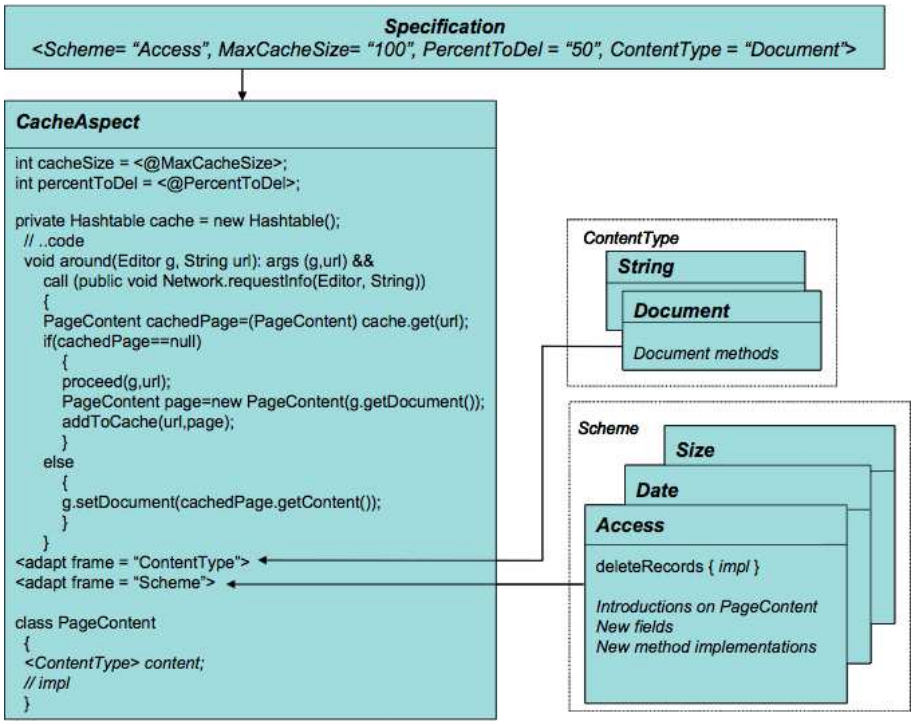

Figure 5.2: Parameterized < *adapt* > to Provide Variations in Cache Aspect [33]

A software product line can be subject to a variety of changes over its lifetime ranging fom addition, retraction, restructuring and replacement of a feature to introduction of a new product or an entirely new product line (in instances where variability becomes too large).

Frame technology is essentially a language independent textual pre-processor that creates software modules by using code templates and a specification from the developer. Examples of typical commands in frames are `<set>` (sets a variable), `<select>` (selects an option), `<adapt>` (refines a module with new functionality), and `<while>` (creates a loop around repeating code). While the framing solution helps to clearly identify a concern on the code-level, it is not a particularly elegant solution, bcause the code gets cluttered with tags which can make code difficult to read, understand and therefore evolve.

Aspect-oriented programming addresses problems of crosscutting concerns and code

tangling, but no parameterization support is available. This issue can be partly solved by using abstract aspects and concrete aspects derived from these for the particular variants required by a particular product. However, this solution can lead to *inheritance anomalies* [39] in deeper inheritance structures and also requires that the developer or maintainer possesses an understanding of the operations encapsulated by the abstract aspect.

The framed aspects approach states that, while both techniques have their strenghts and weaknesses, a hybrid of the two approaches can provide essentially all the combined benefits thus increasing configurability, modularity, reusability, evolvability, and longevity of product line assets. The method is based on using aspects to encapsulate otherwise tangled features in the product line and use frames to provide parameterization and reconfiguration support for the feature aspects. The balance of aspect-oriented programming and frames reduces the template code clutter induced by frames alone and at the same time give the ability to create meta variables and options which can be bound to a specification from the developer when the frame processor is executed.

Let us now look at a frame for our crosscutting *cache* feature. The cache aspect in figure 5.2 contains code that is common to all variant forms of the generic cache. In the first two lines of the `CacheAspect` frame two configuration variables are declared for the cache size and the percentage of the cache to delete when the cache is full.

To enhance reusability and flexibility the framed aspect approach is used to parameterize the cache aspect with variants for the type of content which is cached and the scheme for deleting items in a full cache based on their size, date or access. These variations are specified by an `<adapt>` frame.

The specification frame as can be seen at the top of the figure specifies the configuration values and variation points for the aspect. All the frames together are then parsed and merged to result in the final aspect without any variation left in it. The configured aspect is then ready to be used in a product line product.

Loughran et al. have developed a methodology which allows a feature diagram using FODA (see also section 2.2.4 and [28]) for a given reusable aspect component to be created and mapped directly to framed aspects [34].

5.3 Analysis

5.3.1 Classification of Product Lines

Aspect-orientation can be incorporated in product line engineering in different ways: in the collection of assets in the product line (the asset *library*) for modularization of otherwise crosscutting concerns on the code level, and/or in the production plan to tackle scattering of variable features in the production plan itself. This leads to four different types of product lines, which are each briefly discussed in this section. Table 5.1 illustrates the different types.

Asset Library	Production Plan
Not AO	Not AO
Not AO	AO
AO	Not AO
AO	AO

Table 5.1: Aspect-Orientation in Product Lines

After classification of the product lines, we investigate AOSD techniques for solving crosscutting on both the implementation and production plan level in the next section. In this report we assume aspect-orientation in the production plans as well as the asset library, because we would like to modularize crosscutting on both levels.

Non-AO Assets, Non-AO Production Plan

In our exploration of the different types of product lines we start off with the 'traditional' product line, that is aspect-orientation is not used in the asset library nor in the production plan. This type of product line is depicted in figure 5.3.

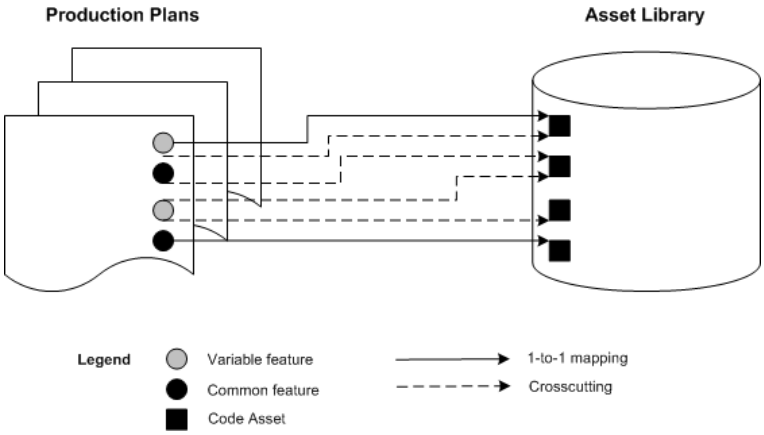

Figure 5.3: 'Traditional' Product Lines

As discussed earlier, during the production process specified in the production plans (one generic plan and the product-specific plans) the common features and a number of variable features are selected for the product. Features map to code assets in the product line library. This mapping can be 1-to-1, a feature is then implemented by 1 asset, or crosscutting where a feature is implemented by multiple assets.

Non-AO Assets, AO Production Plan

Figure 5.4 illustrates a product line where the production plans are aspect-oriented and the asset library consists of standard object-oriented code.

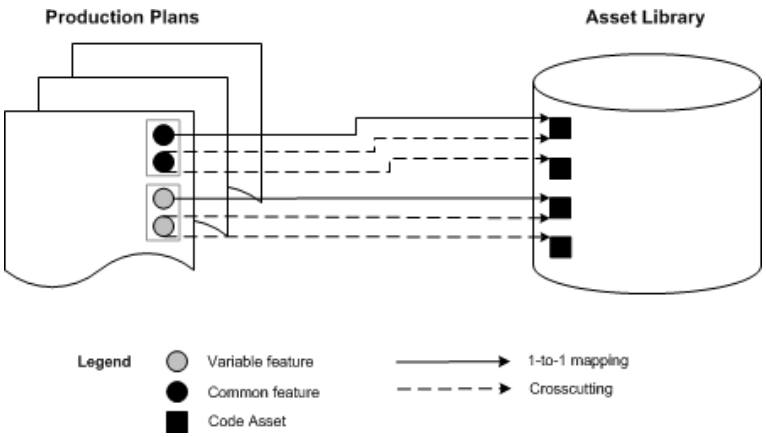

Figure 5.4: Non-AO Assets, AO Production Plan

The common and variable features are now grouped in the production plans and the production process within. Scattering and tangling however still occur in the asset ilbrary, when a feature maps to multiple assets and/or multiple features are implemented by a single asset.

AO Assets, non-AO Production Plan

In this case the asset library uses AOSD techniques to modularize crosscutting concerns as aspects in the library. The production plan is not aspect-oriented here. See figure 5.5 for a schematic overview.

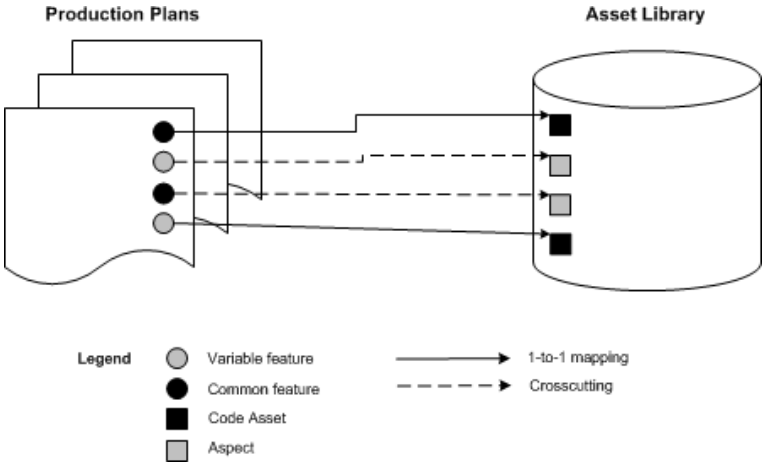

Figure 5.5: AO Assets, Non-AO Production Plan

Now crosscutting features cleanly map to single assets, namely aspects. These aspects are composed with the other code assets to form the end-product.

AO Assets, AO Production Plan

When both the asset library and the production plans use aspect-orientation the situation is as in figure 5.6.

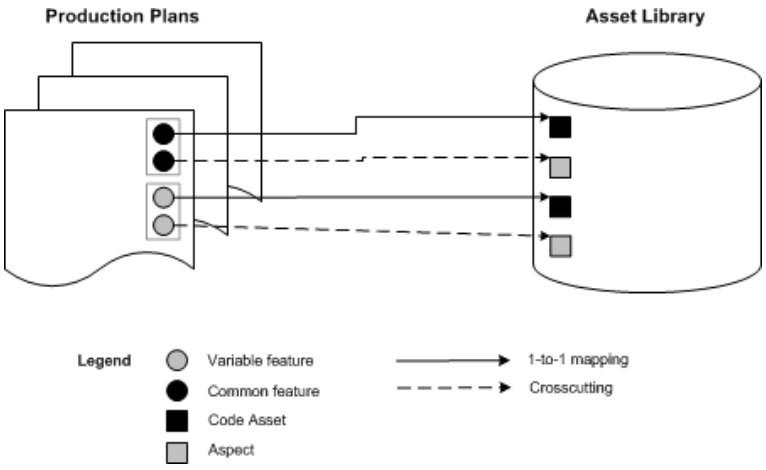

Figure 5.6: AO Assets, AO Production Plan

One can see that the variability is modularized in the production plan. 'Regular' as well as crosscutting features are implemented by single components in the asset library.

5.3.2 Approach to Identified Problems

We now propose solutions to the problems with crosscutting concerns in the product line context as we have identified in the previous chapter. Again we make a distinction between the component level, that is the asset library from which products are built using the production plans, and the production plan level, that is the production plans themselves and the features they contain.

These solutions are each explored with a concrete case example in the next section.

Component Level

Modularization
As we have discussed earlier in sections 2.3 and 5.2.2 aspects modularize otherwise crosscutting concerns. Any AOP language offers concepts to express a crosscutting concern as a single module. We have seen in the previous chapter how the crosscutting concern Replay Actions was modularized with the AOP language AspectJ.

Configuration
In this chapter we earlier explored the *framed aspects* approach to aspect configuration. When using this approach the product developer creates (or generates) a *specification frame* that selects the right variations for the aspect in question. The specification frame and the aspectual asset are then input to the *frame processor* which produces the configured aspect.

We apply the framed aspects approach to our case study in section 5.4.

Composition
We have seen that in a product line context it is important to separate the aspect implementation from its context, that is its binding with other code assets. When we use the earlier-discussed *JasCo* language aspect beans are used to modularize a crosscutting concern and define hooks as abstract pointcuts. Connectors are then used to define a pointcut specification which makes the hooks concrete for the given context.

In this approach *combination strategies* can be defined for the correct handling of dependent aspectual behavior.

The reader is again referred to section 5.4 for a concrete example of this solution.

Production Plan Level

As we previously investigated, scattering of variable features in the production plan is a problem for the product developer to keep track of the variations possible for the products and make valid feature selections accordingly.

We want to modularize this variability in the production plan in some way by making feature selections not based on the *hierarchy* of features, but on the *type* of features, that is *common* or *variable*. Common features are available in all products and therefore can be selected automatically during the feature selection process. The product developer should only consider the specific variations for a product.

To modularize variable features in the production plan we propose an approach based on research on *traceability of concerns* [5]. As we have seen in the previous chapter, feature models are often represented as XML-documents which specify the hierarchy and type of features as illustrated in feature diagrams. With the feature type we mean here mandatory, or, alternative, or optional. Whether a feature is common or variable can be extracted from the hierarchy of features.

When we store the XML documents for feature models in an XML database with support for the functional query language *XQuery*, like *eXist* [19], we have a powerful mechanism to extract the information we need from the feature models. Essentially, we want a production plan like the pseudo code listed in figure 5.7.

```
1  aspect ProductComposition {
2    selectCommonFeatures();
3
4    pointcut selectVariableFeatures() { // selection by enumeration of features
5      select x;
6      select y;
7      select z;
8    }
9
10   pointcut selectVariableFeaturesByName() { // wildcard selection of features
11     select Pong*;
12   }
13
14   advice: selectVariableFeatures() {
15     // configuration
16     x.setProperty(property, value);
17     x.setProperty(property, value);
18
19     y.setProperty(property, value);
20
21     // create product
22     compose();
23   }
24 }
```

Figure 5.7: Pseudo code for an Aspect-Oriented Production Plan

Here we define an aspect for the production process in which the common features for the product are selected, the variable features are selected (by enumeration or by name), the features are configured, and the product is composed.

We can develop xqueries to implement the function and pointcuts in the pseudo code above. In the next chapter we explore generative technologies which use XML-based feature models. In the following examples we assume the feature model from the so-called *Pure::Variants* approach. However, the xqueries can be modified to work with any other XML-based feature model.

To begin with the selection of common features, the according xquery is listed in figure 5.8.

```
1  declare namespace cm="http://www.pure-systems.com/consul/model";
2  declare namespace ps="http://www.pure-systems.com/";
3  declare namespace f="http://my-own/xquery/";
4
5  declare function f:selectCommonFeatures(){
6          let $mandatoryFeatureIDs := //cm:relation[@cm:type="ps:mandatory"]/cm:target
7
8          for $target in $mandatoryFeatureIDs
9          let $featureID := fn:substring(data($target),3)
10         let $feature := //cm:element[@cm:id=$featureID]
11         return $feature//cm:vname/data(cm:mimedesc)
12 };
13
14 f:selectCommonFeatures()
```

Figure 5.8: XQuery for Selection of Common Features

To select the common features we select all features from the feature model that have a mandatory relation with its parent feature and – in this example – we output for each mandatory feature its name.

A second example xquery is shown in figure 5.9.

```
1  declare namespace cm="http://www.pure-systems.com/consul/model";
2  declare namespace ps="http://www.pure-systems.com/";
3  declare namespace f="http://my-own/xquery/";
4
5  declare function f:selectVariableFeature($name){
6          let $feature := //cm:element[@cm:name=$name and @cm:type="ps:feature"]
7          let $featureID := data($feature/@cm:id)
8          let $featureRelation := //cm:relation[@cm:type="ps:alternative" or
9          @cm:type="ps:optional" or @cm:type="ps:or"]/data(cm:target)=concat('./',
10         $featureID)
11
12         return $featureRelation
13 };
14
15 f:selectVariableFeature("AbsorbingCollision")
```

Figure 5.9: XQuery for Selection of Variable Features by Name

With the xquery function `selectVariableFeature` we select a variable feature by its name, i.e. `AbsorbingCollision`. The example query checks whether the feature

exists in the model and indeed is variable. If the check is positive the function returns `true` or else `false`.

As we will see in the next chapter, the product developer can be supported by tools to simplify the production process. We may not want to require from the product developer to be proficient in XQuery. Therefore, we propose that support for modularization of variability in the production plan is incorporated in tooling. This means that feature selections can be made not only by traversing the hierarchy of features, but the product developer can also select variable features by providing an enumeration of feature names or select features by using wildcards. A graphical user interface should be added to the tooling to offer this functionality.

5.4 Application of Solutions to Case

Now that we have explored solutions to our previously identified problems with crosscutting concerns, we apply these topics to the AGM product line.

5.4.1 Component Level

Modularization

On the component level, we can modularize crosscutting features with aspects as we have earlier investigated with the *Replay Actions* feature in section 4.3.6.

Configuration

When using the Replay Actions feature in an AGM product the product developer might want to configure the maximum number of game states that are saved before the oldest game state is deleted. When applying the framed aspects approach we could define a configuration variable for the aspect as follows:

```
1  public aspect ReplayActions {
2    int maxGameStates = <@MaxGameStates>;
3
4    // ... code
5
6    if (numberOfGameStates == maxGameStates) {
7      deleteOldestGameState();
8    }
9
10   // ... code
11 }
```

Here we define an aspect variable `maxGameStates` which gets the value of the frame variable `MaxGameStates`. The aspect variable is used to check whether the number of saved game states has reached the maximum number of game states. If this is the case the oldest game state is deleted.

The product developer can now configure the Replay Actions aspect with a specification frame, for example:

```
1  <MaxGameStates = "10">
```

Here we configure the Replay Actions aspect to keep track of a maximum of 10 game states.

Composition

We have seen that in a product line context it is important to separate the aspect implementation from its context, that is its binding with other code assets. When we use the earlier-discussed *JasCo* language, we could define an *aspect bean* for the Replay Actions feature as follows:

```
class ReplayActions {

  hook GameStateSave {
    GameStateSave(method(..args)) {
      execute(method);
    }

    before() {
      // Game state collection and save code here
      ...
  }

  hook GameStateRestore {
    GameStateRestore(method(..args)) {
      execute(method);
    }

    after() {
      // Game state restore code here
      ...
    }
  }
}
```

Here we define two *hooks* for saving and restoring the game state.

Deployment of the Replay Actions aspect bean on the Brickles game can then be done with the following *connectors*:

```
connector BricklesGameStateSave {
  ReplayActions.GameStateSave gss =
    new ReplayActions.GameStateSave(
    * Paddle.collideWith(*) );

   gss.before();
}

connector BricklesGameStateRestore {
  ReplayActions.GameStateRestore gsr =
    new ReplayActions.GameStateRestore(
      * Handler.ReplayActionsClick(*) );

  gsr.after();
}

```

Here we deploy the save and restore hooks at the selected join points.

5.4.2 Production Plan Level

On the production plan level, we can modularize variability in the plan by using XML-based feature models and XQuery to extract features from the model.

For example, the code for the production of the Brickles game would be:

```
1  // selection of common features
2  common = selectCommonFeatures();
3
4  // selection of specific variations
5  movement = selectVariableFeature("StraightLineMovement");
6  collision = selectVariableFeature("AbsorbingCollision");
7  replay = selectVariableFeature("ReplayActions");
8
9  // configuration of selected assets (i.e. use specification frame)
10 // ... code
11
12 // composition of assets to form end-product (i.e. use connectors)
13 // ... code
```

Here the previously defined xquery functions `selectCommonFeatures` and `selectVariableFeature` are used to extract the XML representation of the selected variations from the model. We select the specific variations for the Brickles game for the features `movement` and `collision handling`.

As we will see in the next chapter, next to the features their mapping to code assets is modeled as well. This means that when we select features we indirectly also select the right code assets. These assets can then be configured and composed together to produce the game product.

In the next chapter we also discuss the earlier-mentioned tool support for making feature selections and transforming these to a concrete product.

5.5 Summary

In this chapter we started with additional background on component based software engineering and aspect configuration. We then explored different combinations of aspect-oriented and non-aspect-oriented production plans and asset libraries to get a better understanding of aspect-orientation in product lines. In this report we assume aspect-orientation in the production plans as well as the asset library, because we would like to modularize crosscutting on both levels.

We then applied aspects to cope with the problems with crosscutting concerns on both the component level and production plan level as we have identified in the previous chapter. On the component level we use aspect-oriented programming to modularize the implementation of crosscutting concerns. We have introduced the concept of *framed aspects* to configure aspects for a specific product and we have illustrated composition of aspects with other assets for a specific context with the aspect-oriented *JasCo* language.

On the production plan level, we have proposed XML-based feature models in combination with the functional query language *XQuery* to modularize scattering of variable features in the production plans.

We have applied the proposed solutions to our case study. We have used the earlier-introduced *Replay Actions* feature to illustrate how aspect configuration and composition can be dealt with. We finished the application to the case with a discussion of an aspect-oriented production plan for the Brickles game.

Chapter 6

Applying Generative Production Plans

6.1 Introduction

Now that we have investigated problems with crosscutting concerns for production plans and our approach to modularizing these using aspects, we discuss the impact of aspect-orientation for the product line process. We introduce the concept of *generative production plans* in this chapter: production plans that to some degree support automation of the production process. Product line engineering is moving more and more from production of software products by hand to automated *generation* of applications from the product line through some sort of specification. This goal obviously has implications for the structure and contents of production plans. However, the steps in a *generative product line process* are unclear.

We first discuss more background information on different levels of automation of production plans and the ideas behind generative software development.

Then we study the requirements for generative production plans and two generative technologies that can provide automation of the production process. We work out the product line process for each of these technologies. These results are used at the end of this chapter to assess the impact of incorporating aspect-orientation in the product line process.

6.2 Background

6.2.1 Automation of Production Plans

Before we go deeper into the production plans, we distinguish between three types of production plans based on the level of automation of the production process:

- **manual**
- **semi-automatic**
- **automatic**

A manual production plan provides the guidelines to the product developer to instantiate products from the product family, but offers no automation of the production process. The developer builds the product by hand from the available assets.

When using a semi-automatic production plan, the product developer needs to select the feature set for the product and then the production plan automatically selects the right assets, but the developer still needs – to some degree – configure and compose the end-product from these assets.

In a fully automatic production process the product developer only needs to select the features for the product and the production plan then automatically selects, configures and composes the assets to form the end-product. Some product lines have a fixed set of family products with specific features per product. In this case the product developer only needs to select the product to be generated.

6.2.2 Generative Software Development

Generative software development [15] is about automating the creation of software products, such as components and applications. It combines domain engineering with appropriate technologies for implementing the elementary components of a product family and for their automatic assembly. This requires modeling product families, ordering products, providing implementation components to assemble the products from, specifying the mapping from product specifications to concrete assemblies of implementation components, and implementing this mapping using generators.

The first step towards achieving this is *generative domain modeling*, which enables automatic generation of a system in a family based on its specification. A generative domain model consists of a *problem space*, a *solution space* and the *configuration knowledge* mapping between them. This is shown schematically in figure 6.1.

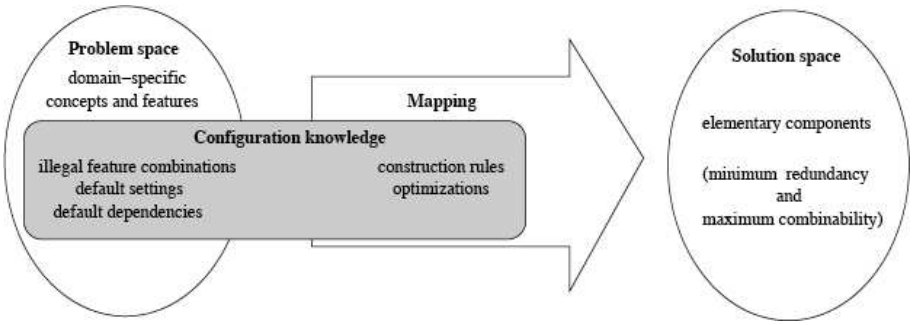

Figure 6.1: Elements of a Generative Domain Model [16]

The problem space consists of the application oriented concepts and features that are necessary for the development of an application. There are different kinds of features: *concrete features*, *crosscutting features* and *abstract features* [15].

A concrete feature is directly mapped to a component, which can possibly be a parameterized component. An example would be implementing sorting by a sorting component. A crosscutting feature is mapped to an aspect as we have investigated in the previous chapter. An abstract feature, for example a performance requirement, does not have any specific mapping. It is implemented by an appropriate combination of components and aspects.

The configuration knowledge specifies certain combinations of features that may not be allowed, as well as, if some features are not specified, their default settings and dependencies that need to be assumed. It also specifies the construction and optimization rules.

The solution space consists of the implementation components with all their likely combinations. The implementation components are designed so that they can be combined in as many ways as possible, while maximizing reuse, and minimizing code duplication. Section 5.2.1 gave an overview of the research area of *component-based software engineering*.

The development and implementation of a generative domain model for a family of systems is one of the essential objectives of generative software development. The main development steps of generative software development are:

- Domain scoping
- Feature modeling and concept modeling
- Designing a common architecture and identifying the implementation components
- Specifying a domain-specific notation for the ordering of systems
- Implementing the implementation components
- Implementing the domain-specific notations
- Implementing the configuration knowledge using generators.

These steps need to be done iteratively and incrementally during the analysis, design and implementation phases. Clearly, looking at the generative software development process, many parallels can be drawn with the domain engineering and application engineering activities from the two life-cycle model (see section 2.2.4).

6.3 Analysis

6.3.1 Generative Production Plans

We now explore the requirements for a generative production plan for generation of products through some sort of specification, as far as the product line context allows this (maturity of the product market, stability of product line scope, etc.). What we want for the production plan is:

- A formalism to express a product family specification for the common and variable features in the product line and the hierarchy of features.
- Feature constraint checking: specification of valid combinations of features and a mechanism to check this.
- A formalism to express a product specification: feature selection and configuration.
- Tool support to create a product specification from the family specification and perform constraint checking.

- Transformation of product specification to asset selection and configuration.
- Composition of selected assets to yield the product.

We now explore two *technologies* for generative product line engineering and work out the process for both approaches. Then the two technologies are compared and one of them is selected for further investigation.

6.3.2 XML-Based Feature Modeling Process

An interesting, generative approach is *XML-Based Feature modeling* [11] which introduces the concepts of a *family model* and an *application model* that are both feature models and describe the software assets behind a product family and the features selected for an application respectively. The feature models are expressed in an XML-based format and this gives handles for constraint checking (check that only valid combinations of features are possible) and tool support (a standard XML environment with XML schema and XSL support suffices).

We have worked out the UML activity diagram for the entire product line process for this approach is depicted in figure 6.2.

The diagram is divided into the domain engineering and application engineering activities. Every relevant element in the diagram is numbered and now step-wise discussed.

Domain Engineering

[1] Family Meta-Model

The XML-Based Feature Modeling approach defines a so-called *family meta-model* which is an XML Schema that specifies which elements and attributes can be used in the family model which in turn is a feature model.

[2] Family Meta-Model (as Input)

The family meta-model is an input for the definition of the family model by a domain engineer. See also numbers [3] and [4].

[3] Definition of Family Model

The domain engineer defines the family model for a specific product line. This feature diagram contains all features available in the product line with their relations. This results in a hierarchy of features, where each feature is of the type mandatory, or, alternative, or optional.

The domain engineer can use a standard XML tool to create the XML specification of the family model.

[4] Validation of Family Model

The family meta-model validates any family model created for a product line through a standard XML tool with support for XML Schema and therefore offers an environment for the domain to create a valid family model.

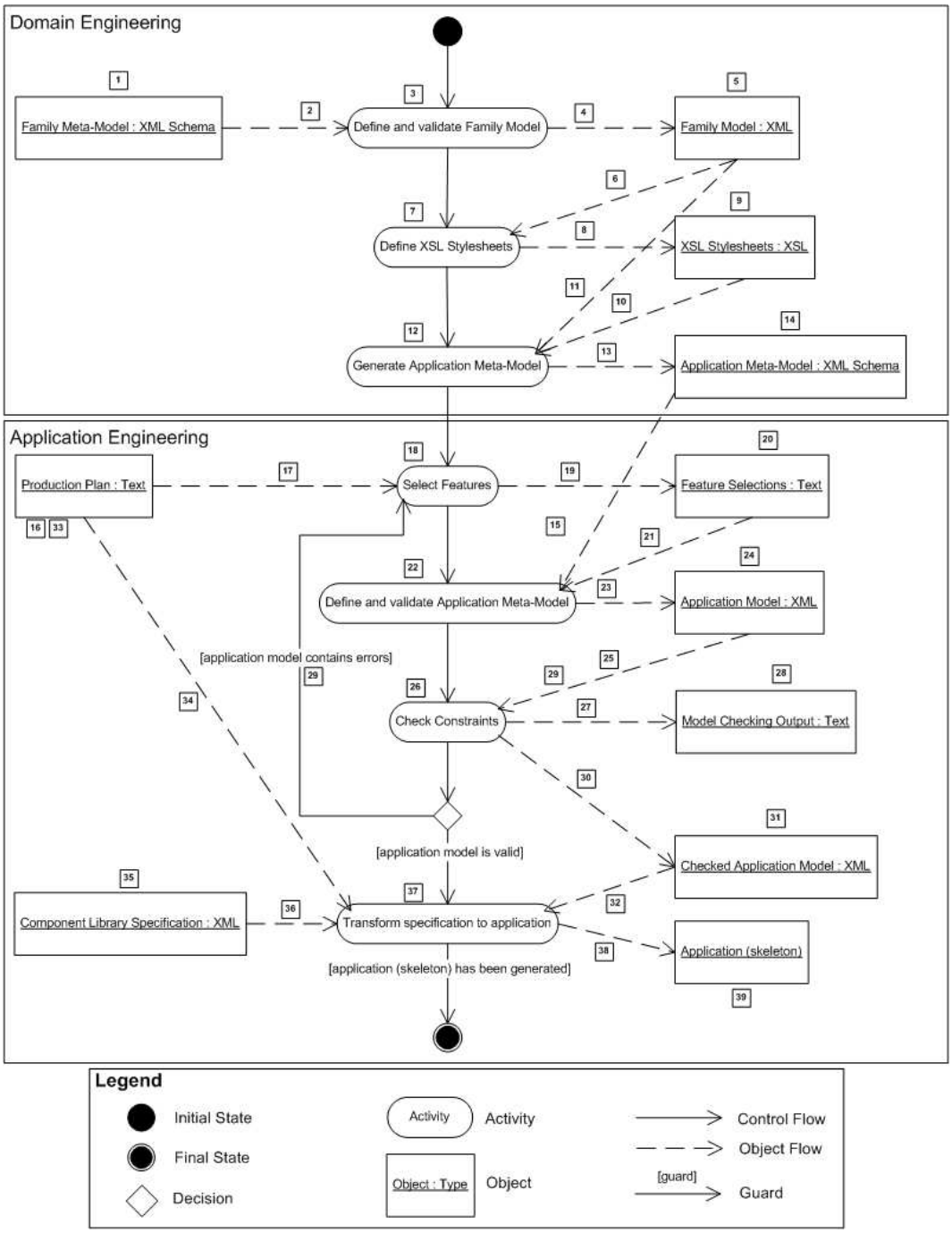

Figure 6.2: Product Line Process for *XML-Based Feature modeling*

[5] Family Model

The first deliverable in the domain engineering process in then a validated family model with the specification of all product line features.

[6] Family Model (as Input)

In the next step, the family model is used as an input for the definition and execution of an XSL Stylesheet. The reader is referred to [7] and [8].

[7] Definition of XSL Stylesheets

The family model describes the features available in the product line. This model however doesn't contain information about the mapping of these features to code assets and constraints on features. This means that information needs to be added by the domain engineer in dialogue with the product developer by translating the family model to – what we will see in a moment – the *application meta-model*.

Cechticky et al. found it to be better to also have an application meta-model in stead of using only the family meta-model, because this allows for fine-tuning of the application model for the specific family. This degree of fine-tuning follows from the case studies done in the approach where the number of elements in the application meta-model is between one and two orders of magnitude larger than the number of elements in the family meta-model.

[8] [9] XSL Stylesheets

The XSL stylesheets thus add information to each feature specification. The XSL transformation defines how the original model should be converted to the new model.

[10] XSL Stylesheet (as Input)

The XSL stylesheet is input to an XSLT engine that parses the stylesheet and executes it on the original model which is the second input as we will see in a moment. Most standard XML tools offer functionality for executing XSL transformations.

[11] Family Model (as Input)

The second input for the transformation process is the family model itself.

[12] [13] Generation of Application Meta-Model

The Application Meta-Model is generated in this step by executing the XSL stylesheet on the family model.

[14] Application Meta-Model

The application meta-model is an XML schema which defines the structure for the application models.

[15] Application Meta-Model (as Input)

The application meta-model validates any application model created in the product line. The validation again can be done by a standard XML tool.

Application Engineering

16 Production Plan

The production plan describes the features available in the product line and their interrelations. These interrelations obviously follow from the hierarchy of mandatory, or, alternative, and optional features, but can also be extra constraints that do not result from the feature hierarchy. For example, 'feature A depends on the presence of feature B' or 'feature A excludes feature B'. Different types of so-called *composition constraints* can be specified in this approach on both the family model and application model level.

The production plan also plays a vital role in the transformation process from specification to application which we will discuss later on.

17 Production Plan (as Input)

The information from the production plan is important for the product developer to make a valid selection of features for the product.

18 19 Select Features

When the product developer wants to produce an application from the product line (s)he needs to select the features for the product or – if the product line only produces specific products with a predefined feature set – the product which needs to be created.

As we have just discussed, these feature selections need to be correct, i.e. all mandatory features are selected, exactly 1 feature is selected from multiple alternative features, the feature constraints are satisfied, etc.

20 Feature Selections

The feature selections need to be saved in some format: as a list of features in natural language, or as a partial family model in which only the selected features are specified.

21 Feature Selections (as Input)

The feature selections are used in the process of creating an application model.

22 23 Creation of Application Model

When the product developer wants to create the application model for the product to be produced, (s)he takes the feature selections and translates these to an application model. This translation depends on the format of the feature selections. If the feature selections are specified in natural language, the developer needs to specify each selected feature as an XML fragment for the application model. If the feature selections are expressed as a partial family model the developer can use a variant of the earlier defined XSL stylesheets to generate the application model.

In both cases the application model is validated by the application meta-model (XML Schema).

24 25 Application Model (as Input)

The resulting application model is syntactically valid, but can still violate certain constraints. Therefore the application model is input to a constraint checking XSL

stylesheet.

[26] [27] [28] Constraint Checking

The application model is checked with the constraints specified in the application meta-model. This might be done through an XSL stylesheet that defines the constraints that need to be checked and how these checks should be outputted. The XSL stylesheet is then executed, evaluates the application model and outputs for each constraint whether the constraint is satisfied or not.

[29] Constraint Errors

In case of constraint errors the output is redirected to the Select Features process with which the product developer can correct the unsatisfied constraints by modifying the feature selections. This cycle continues until the application model is valid and satisfies all composition constraints.

[30] [31] [32] Checked Application Model

If all composition constraints are fulfilled the application model is ready to be transformed to a concrete application. The XML specification of the application model is input to the transformation process.

[33] [34] Production Plan (as Input)

The transformation process in this approach is not clear at the time of writing, because the so-called *generative environment* to actually create end-products is 'future work'. Therefore the role of the production plan in the transformation is also unclear. However, as we will see in the second approach, the production plan will extend a standard transformation process to compose the actual application (or application skeleton).

[35] [36] Component Library Specification (as Input)

The code assets available in the product line are usually stored in a component library (see also section 5.2.1). Among other things, the interfaces of the components and the way in which they can work together to provide functionality are specified for the library.

This specification is input to the transformation process in that the code assets referred to by the application model can be selected, configured and composed to form the end-product.

[37] [38] Transformation from Specification to Application (Skeleton)

Again, the transformation process for this approach is not yet worked out at this moment. Based on our knowledge from the second approach which we will discuss in the next section, we can assume that there is a standard transformation process which can be extended by the product developer through the production plan.

[39] Application (Skeleton)

The result of the entire product line process is the application or application skeleton. In the latter case, the product line creates a framework of the product which the product developer needs to configure and modify by hand to finish the product.

We now continue with a discussion of a second approach (*Pure::Variants*) and look at the process which it proposes. Then we compare both approaches with each other.

6.3.3 Pure::Variants Process

A similar approach for product specification and production is *Pure::Variants* [46]: an Eclipse plug-in for creating feature models and family models, and to transform these to concrete applications. *Feature relations* can be used to define valid selections of combinations of features for a domain.

Family models describe how products in the product line will be assembled or generated from code and other artefacts. Each family model is made up of components, parts and source elements. Components are organized into a hierarchy that can be of any depth. Each component is further decomposed into parts that in turn are built from source elements, as can be seen in figure 6.3.

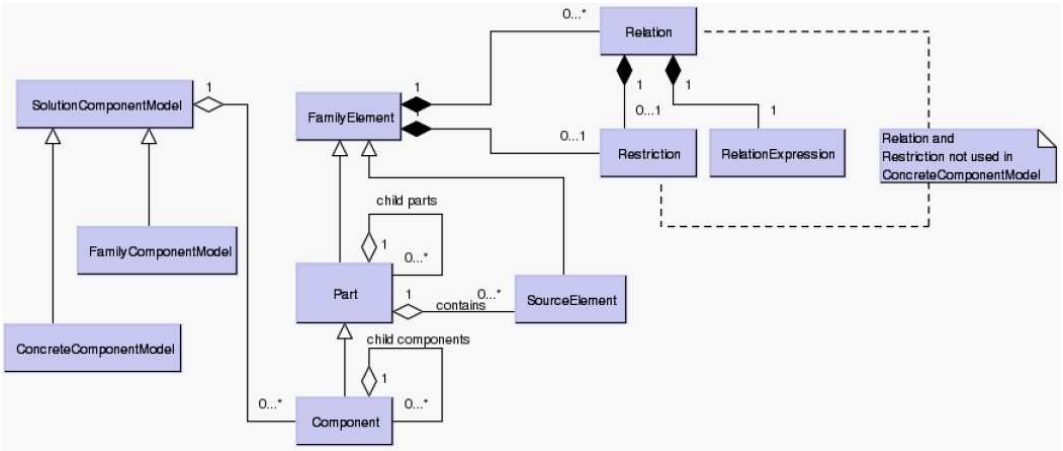

Figure 6.3: Pure::Variants Component Model [47]

The Pure::Variants tool can be used to automatically generate individual products. The product line process for Pure::Variants is depicted in figure 6.4. We again stepwise discuss the elements in the diagram.

Domain Engineering

[1] Feature Meta-Model

The Pure::Variants approach offers a meta-model for feature models, which again is an XML Schema.

[2] Feature Meta-Model (as Input)

The domain engineer normally uses the Eclipse tool to create the feature model for a specific product line / domain. The resulting feature diagram is then automatically validated against the meta-model. Optionally, one could also define or modify the feature model by hand and then validate the model through a standard XML tool.

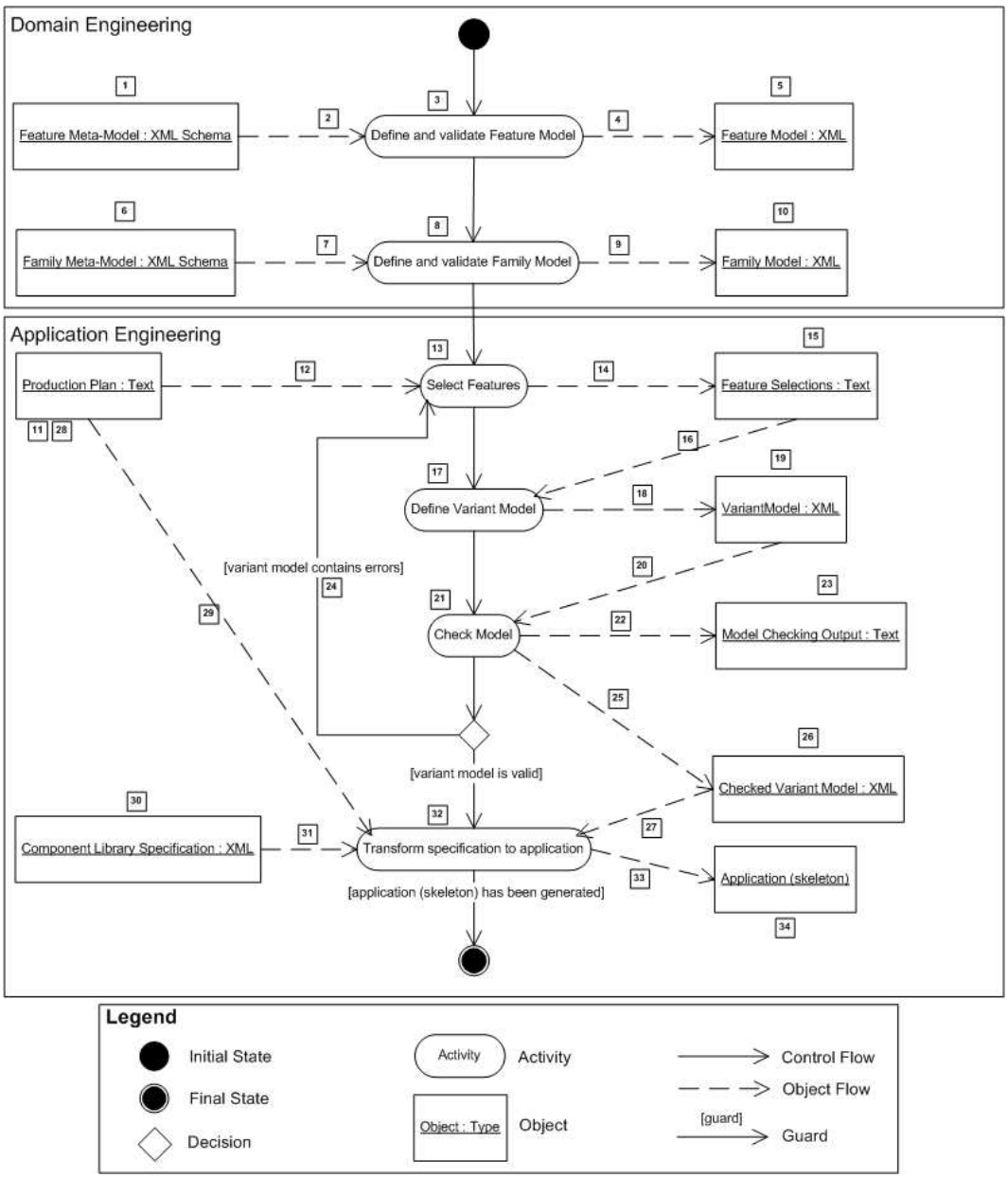

Figure 6.4: Product Line Process for *Pure::Variants*

3 4 Definition of Feature Model

The domain engineer defines the hierarchy of features in the feature model in the Eclipse environment through a graphical user interface (GUI). See section 6.4 for a more in-depth discussion of the Pure::Variants Eclipse tool.

5 Feature Model

The feature model for the product line contains all features and relations between features. These feature selections obviously result from the hierarchy of features (for example, alternative subfeatures for a feature), but Pure::Variants also offers so-called *feature restrictions* for specification of more complex relations between features. A restriction rule may contain arbitrary pvProlog statements (a dialect of Prolog). Table 6.1 shows some examples of the specification of feature restrictions.

Restriction	**pvProlog statement**
feature A requires feature B	`requiresFeature('B')`
feature A excludes feature B	`conflictsFeature('B')`
feature A requires feature B or C	`requiresFeature('B','C')`

Table 6.1: Feature Restrictions in Pure::Variants

Also logical operators such as `and`, `or`, and `not` can be used to define more complex restrictions. The Pure::Variants approach also provides pvProlog statements to restrict features in a more low-level way by for example checking attribute values.

6 7 Family Meta-Model

As with the feature model the family model is also validated by an XML Schema: the family meta-model. This meta-model describes the elements and attributes allowed in the family model to specify the code assets in the product line.

8 9 Definition of Family Model

The Eclipse plugin for Pure::Variants also provides a user interface for defining the family model. This interface is also discussed further in section 6.4.

10 Family Model

Relations between features (*problem space*) and components, parts, and source elements (*solution space*) are defined in the family model which also specifies the instantiation of the component model for the product family with a hierarchy of the code assets available in the product line and the mapping of features to implementation. As with features relations and restrictions these can also be specified for family elements. Some examples of these restrictions are given in table 6.2.

Restriction	**pvProlog statement**
Class A implements Feature B	`hasFeature('B')`
Component A requires Component B	`requiresComponent('B')`
Class A provides function B	`hasPart('B')`

Table 6.2: Restrictions on Family Elements in Pure::Variants

Application Engineering

The task of the domain engineer is now finished and the product developer takes over by defining a so-called *variant model* which we will discuss in a moment. The feature and family model resulting from the domain engineering activity are used for this variant model.

[11] Production Plan

The Pure::Variants offers extended tool support for specification of the feature, family, and variant models as we will see in the next section. Functionality for creating feature selections for a to be produced product and checking whether these selections are valid is available in the tool. Therefore the role of the production plan in the selection of features is limited to documentation of the available features and their relations.

[12] Production Plan (as Input)

The production plan is thus used as a reference for the product developer when selecting a feature set for an application.

[13] [14] Select Features

The product developer defines a product by its feature set. (S)he needs to select all mandatory features in the product line some variable features depending on the feature type (or, alternative, or optional). The feature selections are made using the Pure::Variants tool (again see section 6.4).

[15] [16] Feature Selections

When the product developer thinks that the feature set is complete and correct for the product, the feature selections are input for defining the variant model.

[17] [18] Definition of Variant Model

The variant model is generated by the Eclipse tool using the family model, feature model, and feature selections. For each selected feature model the according part of the family and feature model are selected and placed into the variant model. After evaluation of all selected features the variant model has been populated with both a partial family model and feature model.

[19] Variant Model

The variant model is represented in XML format, while working with the model is done through the GUI. This representation opens the road to powerful checking and querying mechanisms (such as XML Schema, XPath, XQuery, etc.) which are used by the Eclipse tool.

[20] Variant Model (as Input)

When the product developer has made a variant model using the tool the model needs to be checked for validity and whether all restrictions placed on features and family elements are fulfilled.

[21] Model Checking

The checks performed on a variant model are threefold:

1. The variant model is compared against the feature model to check whether all mandatory features are selected and the variable features are selected correctly, for example exactly one alternative from multiple alternative subfeatures is selected. The tool can automatically resolve some conflicts by for example selecting all mandatory features if the product developer has forgotten to activate one or more for the product.
2. For each feature in the variant model the restrictions (if applicable) are examined. If a restriction isn't met the tool gives back an error for this restriction in the *Problem View* in Eclipse.
3. Also for each family element in the variant model the restrictions are investigated. Again, when a restriction conflicts with the current feature selection, the tool returns an error for the specific restriction.

[22] [23] Model Checking Output

The model checking done by the tool delivers feedback to the product developer for correcting the feature selection, if necessary.

[24] Errors in the Model

When at least one error has occurred while checking the model, the product developer needs to revise his feature selections. This revision cycle continues until the variant model for the application is valid.

[25] Correct Model

When no errors are found during the model checking, the model is valid and the production process can continue.

[26] [27] Checked Variant Model

Now that the variant model is finished it acts as input to the transformation to a concrete application (skeleton).

[28] Production Plan

This variant model is an abstract (XML) description of the solution in terms of components (or modules). This description is used to control a transformation process that in turn generates the source code and other artefacts of the finished product variant. Pure::Variants provides a standard transformation process, but in most cases the product developer will customize and extend the transformation process for the specific product line.

This is where the production plan comes into action. The production plan defines the transformation process for the product line as an XSLT script with Pure::Variants XSLT extension functions, which transforms the nodes of the variant model to a format the component library understands for selection of components.

[29] Production Plan (as Input)

In the Pure::Variants tool the XSLT script defined by the Production Plan is selected for the transformation process.

[30] [31] Component Library specification

As we have seen earlier, the family model specifies the components, parts, and source elements in the product line. However, this family model's structure is conform the XML Schema defined specifically for Pure::Variants, while component libraries generally have their own specification format e.g. OMG IDL (see also section 5.2.1). To prevent the product line developers from doing double work it is possible to translate the component library specification to a family model using e.g. XSLT.

The component library specification is input to the transformation process for selection of the right components from the asset library.

[32] [33] Transformation from specification to application

When generating a product, a component (with its parts and source elements) is only included in the resulting product configuration when its parent is included and when any further restrictions on the components are fulfilled e.g. the selection of specific features, or limits on attribute values associated with features.

The standard transformation process assumes that the family elements map to files or file parts in the filesystem. When transforming the specification to a product the relevant files are selected and copied to the folder in which the application resides.

For our custom transformation process the XSLT script defined by the production plan is executed on the variant model to result in a formalism (e.g. XML) that the asset library understands to fetch the right components. The transformation process then copies the retrieved assets to the application folder and configures the assets as specified in the variant model.

[34] Application (Skeleton)

In the ideal situation the transformation process generates a complete and working application. More realistically, an application skeleton is generated in which all the right assets are available, but some configuration by hand is still needed to finish the product. This is then the task of the product developer.

6.3.4 Comparison of Both Approaches

If we compare the product line process for both approaches (figures 6.2 and 6.4), similarities as well as differences can be seen. Both approaches model the features, assets and applications in XML-based models and use these models in a transformation process to come to an end-product. Both approaches have a mechanism to express constraints on (combinations of) features and assets, but the Pure::Variants approach offers more powerful expression of constraints through Prolog-like statements.

The approaches differ in the level of tool support and the state of the transformation process: the XML-based Feature Modeling approach uses standard XML tools, while Pure::Variants has developed a more complete tool set in the Eclipse environment; the transformation process is 'future work' in the first approach, while

Pure::Variants already offers a working standard transformation process which can be customized and extended by the product developer.

The similarities and differences between both approaches are summarized in table 6.3.

Technology requirements for product production	**XBFM**	**P::V**
Formalism to express product line features	+	+
Feature constraints and checking	+	++
Formalism to express a product specification	+	++
Tool support	+	++
Transformation from specification to asset selection	-	+
Composition of selected assets to yield the product	-	+

Table 6.3: Comparison of Technologies

In this report, we prefer Pure::Variants above the XML-based Feature modeling approach, because Pure::Variants has all the requirements listed in the table above, while the second approach is more focused on the modeling of features and does not yet offer a transformation process from feature specification to implementation. In the next section we go deeper into the workings of the Pure::Variants approach.

6.3.5 Aspect-Oriented Product Line Process

Aspect-orientation in software product line engineering affects the product line process from specification to application. In this section we investigate the impact of aspect-orientation by example of the Pure::Variants product line process and discuss the issues involved.

Each process step on which aspect-orientation has impact is now discussed. The Pure::Variants product line process is again depicted in figure 6.5.

Family Meta-Model

Aspects are implementations of crosscutting features and thus need to be specified in the family model. This means that the family meta-model needs to be extended to incorporate aspects. The Pure::Variants family meta-model already contains the part type *ps:aspect*, but there is no support for aspect configuration nor specification of pointcuts for the composition with other assets.

We propose the addition of configuration variables and hooks to the family model (and thus the family meta-model) for specification of how an aspect can be *fitted* in a specific product context.

A configuration variable should be specified as an element of an aspect and denotes a variation for the aspect in question. A configuration variable has a name, a data type (integer, string, enumeration, etc.) and optionally value constraints. For example, In case of an enumeration type the value constraints specify the set of values from which the product developer can choose.

A hook should also be specified as an element of an aspect and is an abstract pointcut as we have seen in the *JasCo* approach (see section 5.2.1). A hook has a name and an advice type, that is before, after or replace. During product production each relevant hook needs to adapted to the product context by specifying a connector per hook, again as we have seen previously in the *JasCo* language.

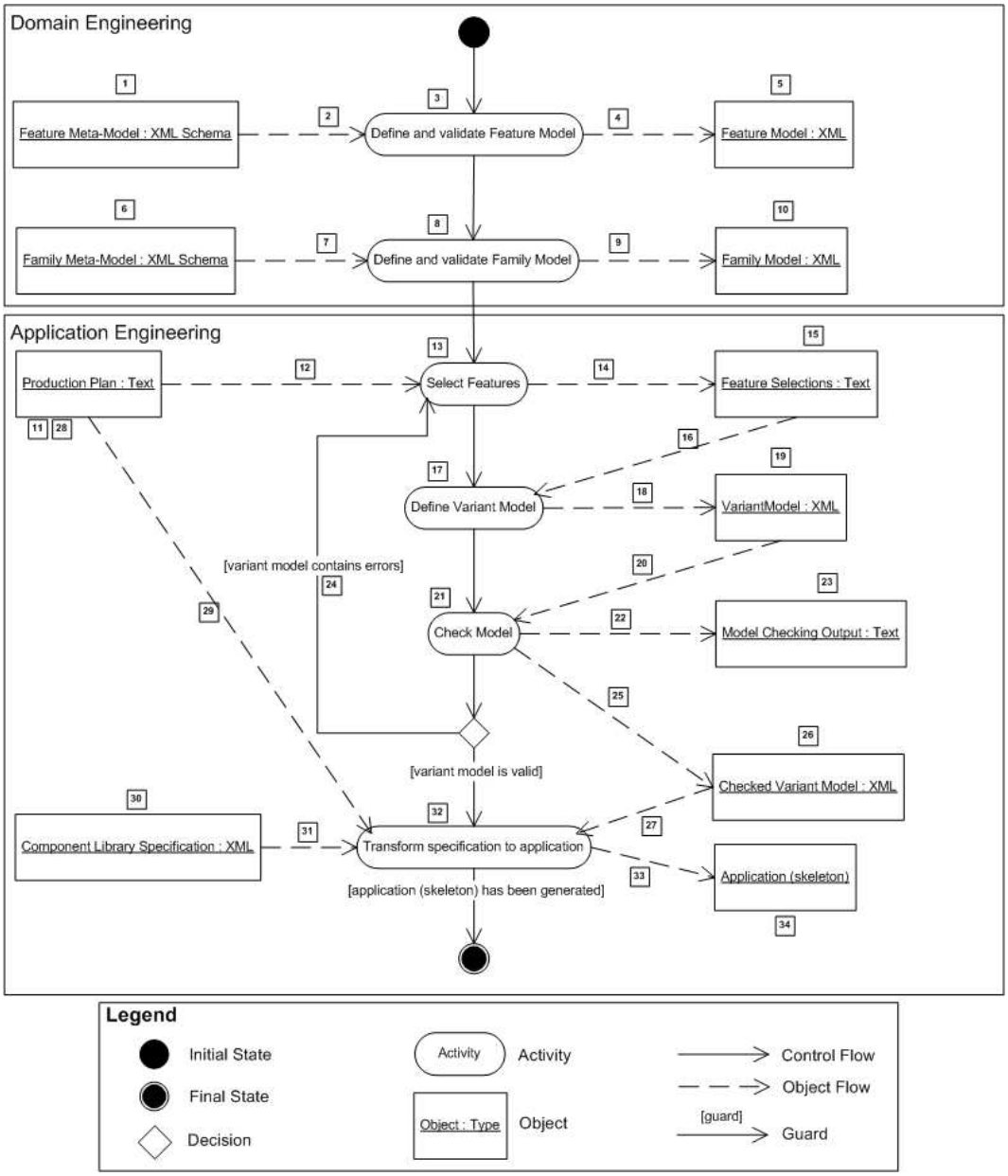

Figure 6.5: Pure::Variants Product Line Process

Define Family Model

While defining the family model the domain engineer specifies the regular family assets as well as the aspects available in the product line. For each aspect the configuration variables and hooks need to be defined.

The resulting family model contains all family assets and the mapping of regular as well as crosscutting features to their implementation.

Production Plan

The production plan provides information to the product developer to make a valid feature selection and specifies how the specification should be transformed to the application (skeleton).

Selection of features can be done by traversing the hierarchy of features or – as we have proposed in the previous chapter – or by selecting variable features by their name. Wildcards should be allowed in name-based selections to select multiple features at once. The tool needs to offer an interface for this alternative method of feature selection with for example an inline search box in which the product developer can type a (part of a) feature name and all matches in the feature model are shown in a drop down box.

Secondly, the transformation of aspects needs to be defined in the production plan by using the configuration variables and hooks for each aspect. We propose a transformation process in which for each aspect a specification frame is provided by the product developer in which all configuration values are specified and this frame in combination with the aspect is input to the frame processor which returns the configured aspect as the result.

Then for each aspect hook a connector is defined by the product developer to bind the aspect to the other selected assets for the product. The configured aspect in combination with the connectors provides the selected behavior at the selected join points.

Component Library specification

The component library should also support aspectual components as separate assets. The library needs to store the framed aspects with their configuration variables and the hooks for each aspect.

Transform specification to application

In the transformation process aspects specified in the variant model are selected from the component library. For each aspect, the aspect implementation, specification frame, and connectors are used as result in the correct binding process.

6.4 Application to Case

To get a better impression of the Pure::Variants approach we discuss the Eclipse tools and their underlying models with the AGM case as example.

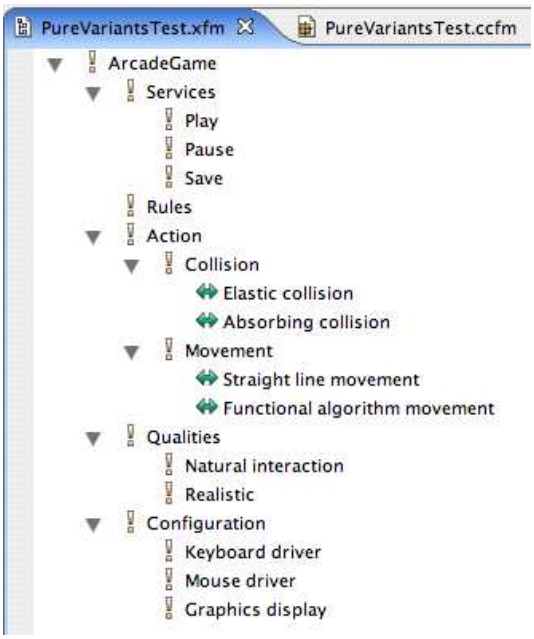

Figure 6.6: Feature model in Pure::Variants
Product line process steps [3] to [5] (see figure 6.4)

6.4.1 Feature, Family, and Variant Models

The Pure::Variants Eclipse plugin offers feature modeling functionality for constructing a hierarchy of mandatory, or-, alternative and optional features. We have used the feature models from the AGM case as an example. See figure 6.6 for a screenshot of this feature model.

The tree shows the hierarchy of features and also the feature types. For example, the mandatory feature *Collision* has two alternative sub-features *Elastic collision* and *Absorbing collision* as we have seen earlier.

With the tool one can also create a family model with the components, parts and source elements which are available in the product line. Based on the earlier discussed basic AGM architecture (see section 3.4.3) we have created a partial family model as illustrated by figure 6.7.

Although the AGM case does not provide components, but only game product instantiated from the product line, we worked out a component structure for illustrative purposes. The solution space for the product family can be concretely modeled e.g. the *EventHandlerDefinitions* component implements three features from the feature model (*play*, *pause*, and *save*) and contains the class `EventHandler` which in turn is implemented in the file `EventHandlerImpl`.

Each product instantiated from the product family has its own feature subset. These feature selections are made in a so-called *variant model* per product. Figure 6.8 shows how a variant model can be populated in the Pure::Variants tool.

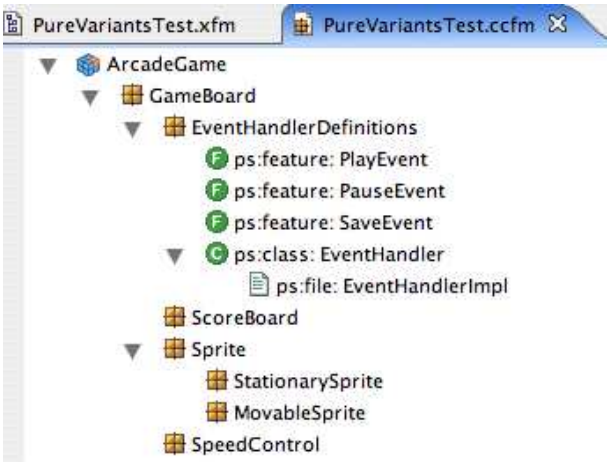

Figure 6.7: Family model in Pure::Variants
Product line process steps [9] to [11] (see figure 6.4)

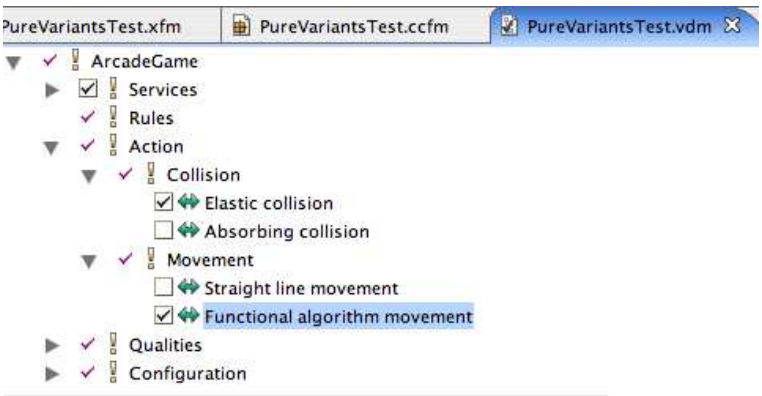

Figure 6.8: Variant model in Pure::Variants
Product line process steps [15] to [27] (see figure 6.4)

Obviously all common features for the product need to be selected. The variation in our AGM feature model is represented by the *Collision* and *Movement* features which both have alternative sub-features. The product developer needs to select from the alternatives before continuing with the product generation. The tool also checks whether the feature selections are valid, by checking for example that only one alternative is selected from a number of features.

6.4.2 XML Representation of Models

All models in Pure::Variants are saved as XML documents. For example, a part of the XML representation of the feature model for the AGM case is depicted in figure 6.9.

```
1 [...]
2 <cm:element cm:class="ps:feature" cm:id="ibICJxoLra_7C0UdB" cm:name="Movement"
3 cm:type="ps:feature">
```

```
4    <cm:relations cm:class="ps:children" cm:id="i9meSQMQDByDZGf3P">
5      <cm:relation cm:id="iBkt-d0j2CpKHjmEL" cm:type="ps:alternative">
6        <cm:target cm:id="ic1I3kg2QMVUGMoPK">./i1-5lexW2y9EocQf0</cm:target>
7        <cm:target cm:id="iGph5VwGYu108nE4M">./iX8Jd3-eUdW1SNv2P</cm:target>
8      </cm:relation>
9    </cm:relations>
10   [...]
11   <cm:relations cm:class="ps:parents" cm:id="iLk1oehR4UxIbFtGV">
12     <cm:relation cm:id="iMfBPD5RPn7L4f_pa" cm:type="ps:parent">
13       <cm:target cm:id="iNh7YsUhrrOXDCjph">./ixQqg30CRz0y1n5Z_</cm:target>
14     </cm:relation>
15   </cm:relations>
16   [...]
17 </cm:element>
18 <cm:element cm:class="ps:feature" cm:id="ijs2MdS-XLAyTYtWC" cm:name="Collision"
19 cm:type="ps:feature">
20   <cm:relations cm:class="ps:children" cm:id="ixHfE8DiUibfv0123">
21     <cm:relation cm:id="ibjaLZr4cJdtv9wuJ" cm:type="ps:alternative">
22       <cm:target cm:id="iCFvTJQyFmRbBtqzN">./iUy1Z_XGtBfBdvNwK</cm:target>
23       <cm:target cm:id="i0eQ5cUpiU1_QrnO3">./irMXpLQ0GapOaLj2J</cm:target>
24     </cm:relation>
25   </cm:relations>
26   [...]
27 </cm:element>
28 <cm:element cm:class="ps:feature" cm:id="i1-5lexW2y9EocQf0"
29 cm:name="StraightLineMovement" cm:type="ps:feature">
30 [...]
31   <cm:relations cm:class="ps:parents" cm:id="iYkrT6InTVYBnn8qV">
32     <cm:relation cm:id="i8vhrICRxPO2u1SVi" cm:type="ps:parent">
33       <cm:target cm:id="iC7vpWS4M0gUQ2sSM">./ibICJxoLra_7C0UdB</cm:target>
34     </cm:relation>
35   </cm:relations>
36   [...]
37 </cm:element>
38 [...]
```

Figure 6.9: Partial Feature Model in XML format

Each feature as an unique id with which it is identified in child- and parent-relations. The XML code is now step-wise explained:

- Lines 1 - 3: the feature `Movement` is defined.
- Lines 4 - 9: the relation of the `Movement` feature with two alternative sub-features (children) of `Movement` is listed and the sub-features are pointed to by the `cm:target` tags.
- Lines 11 - 15: The parent of `Movement` is pointed to (in this case `Action`).
- Lines 18 - 27: The `Collision` feature is defined in the same way as `Movement`.
- Lines 28 - 37: The sub-feature `StraightLineMovement` is defined. Observe that the attribute `cm:id` at line 28 is used previously on line 6 as the content of the `cm:target` element.

The family model XML representation is similar to that of the feature model. A partial family model for the AGM case is listed in figure 6.10.

```
1 [...]
2 <cm:elements>
3 [...]
```

```
<cm:element cm:class="ps:component" cm:id="i1qcBTPpmqBRnN2Z7"
cm:name="EventHandlerDefinitions" cm:type="ps:component">
  <cm:relations cm:class="ps:children" cm:id="i_Cma-wEaukTgMXea">
    <cm:relation cm:id="iJkyNL-IPX45Olt94">
      <cm:target cm:id="iVH3hSoZXUXCQFuyg">./irNZmuR9iSkpXNayr</cm:target>
      <cm:target cm:id="i14I-XAFFTigi7ZNp">./ivmMGl79r9Cfwlabe</cm:target>
      <cm:target cm:id="iv9hwD26QC4hu-XJs">./iWxd1HtL56kSWEJUz</cm:target>
      <cm:target cm:id="iO5TDWyfsvYnh1UoR">./iWLTXpvf1vwMhdm10</cm:target>
    </cm:relation>
  </cm:relations>
  <cm:relations cm:class="ps:parents" cm:id="iYyYE9nqnOmi2ZQGL">
    <cm:relation cm:id="ibQKI8RXduE82eLIM" cm:type="ps:parent">
       <cm:target cm:id="ib5JuUeOZ6AYe_B7t">./i0tv4v-LnoQBeqQu7</cm:target>
     </cm:relation>
   </cm:relations>
   [...]
</cm:element>
<cm:element cm:class="ps:part" cm:id="irNZmuR9iSkpXNayr" cm:name="PlayEvent"
cm:type="ps:feature">
  <cm:relations cm:class="ps:parents" cm:id="iCKOjVRUKLjyrLi0M">
    <cm:relation cm:id="iLPvjcdeIzDKr43Gw" cm:type="ps:parent">
      <cm:target cm:id="iHvi3Sp2eZz7fILEG">./i1qcBTPpmqBRnN2Z7</cm:target>
    </cm:relation>
  </cm:relations>
  <cm:properties cm:id="iC_L8bRrW9sYlXHhz">
    <cm:property cm:fixed="true" cm:id="ihkn1IEDWh8swNXCw" cm:name="fid"
    cm:type="ps:feature">
      <cm:constant cm:default="true" cm:id="iDWEGOs9bfsaOGiRB"
      cm:type="ps:feature">iU8E5RhoUqWaFinnd/ip0O-wpFvPqKRtkMF</cm:constant>
    </cm:property>
    [...]
    </cm:properties>
</cm:element>
[...]
<cm:element cm:class="ps:part" cm:id="iWLTXpvf1vwMhdm10"
cm:name="EventHandler" cm:type="ps:class">
  <cm:relations cm:class="ps:children" cm:id="ijHPChfYRqM3G2RXv">
    <cm:relation cm:id="iITM9c72OLmsyod3e">
      <cm:target cm:id="ik1XRNbLOVQmcxo8r">./ibs5HLvML2_NJ1gDu</cm:target>
    </cm:relation>
  </cm:relations>
  <cm:relations cm:class="ps:parents" cm:id="i1HU_0XvcUzoijTVn">
    <cm:relation cm:id="i8mmYggyBgBJJ8LNe" cm:type="ps:parent">
      <cm:target cm:id="iRl4yQJ1ykplt_gwL">./i1qcBTPpmqBRnN2Z7</cm:target>
    </cm:relation>
  </cm:relations>
  [...]
</cm:element>
<cm:element cm:class="ps:source" cm:id="ibs5HLvML2_NJ1gDu"
cm:name="EventHandlerImpl" cm:type="ps:file">
  <cm:relations cm:class="ps:parents" cm:id="iGVsi8GREM1iJ2yDy">
    <cm:relation cm:id="iJnvMZxylmfP8k1gI" cm:type="ps:parent">
      <cm:target cm:id="iX7mIHm3bVnVywXok">./iWLTXpvf1vwMhdm10</cm:target>
    </cm:relation>
  </cm:relations>
  [...]
</cm:element>
</cm:elements>
[...]
```

Figure 6.10: Partial Family Model in XML format

As one can see the family model addresses relations between elements in the same way as the feature model by referring to unique id's. We again shortly walk through the code:

- Lines 4 - 20: the component `EventHandlerDefinitions` is specified with its four child relations: 3 features and 1 class as we have seen in figure 6.7 and 1 parent relation (the `GameBoard` component).
- Lines 21 - 36: the part `PlayEvent` is listed which is a feature mapping as defined by `cm:type="ps:feature"` on line 22. The feature to which is mapped is specified on lines 28 - 33. The other 2 features mapping are specified in the same way in the complete XML file.
- Lines 38 - 51: the part `EventHandler` is listed which is a class as contained in the component.
- Lines 52 - 60: finally, the source element `EventHandlerImpl` is specified which is a concrete file with the implementation of the `EventHandler` class.

The variant model combines both the feature model and family model in one XML file, where only the selected features and the associated components, parts and source elements are extracted from the original models.

6.4.3 Aspect-Orientation in Pure::Variants

Let us now look at how crosscutting features are modeled in Pure::Variants and how these are represented in the XML specification for the feature model and family model. Then we investigate how the implementation of this feature as an aspect can be modeled in the tool.

We previously introduced an aspect in our case study with the crosscutting feature *Replay Actions* (see chapter4). Figure 6.11 shows the added feature `ReplayActions` in the feature model. As one can see, the feature is part of the services feature.

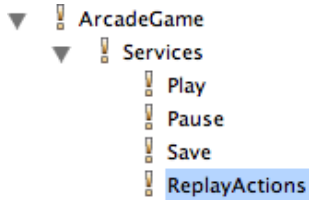

Figure 6.11: Feature Model with Crosscutting Feature *ReplayActions*

We have also added mappings of the `ReplayActions` feature to the family model, as shown in figure 6.12. The feature is mapped on multiple components, namely `EventHandlerDefinitions` for handling the button click by the user through the game interface, `MovableSprite` for saving the game state of all movable elements in the game with each collision of a movable sprite with a stationary sprite, and `SpeedControl` to save the speed setting in the game state.

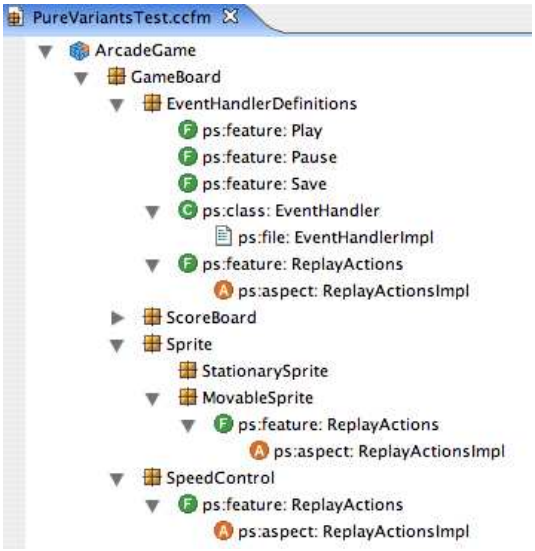

Figure 6.12: Family Model with Crosscutting Feature *ReplayActions*

This means that the `ReplayActions` feature is scattered over multiple components for implementation of the feature and that the feature is tangled with multiple other features in the case of the `EventHandlerDefinitions` component.

As you can see in the model, all mapped `ReplayActions` features are implemented by the `ReplayActionsImpl` aspect part type. The functionality is however limited, because for example pointcut definition and aspect configuration is not possible in the current version (2.0). Pure-Systems, the company behind Pure::Variants, does offer an AspectC++ plugin for Visual Studio and these aspects could be mapped to in the transformation process. Support for other aspect-oriented languages is not available at the time of writing.

XML Representation of Crosscutting Feature
The XML representation of the `ReplayActions` feature in the feature model is depicted in figure 6.13.

```
<cm:element cm:class="ps:feature" cm:id="iSvar_w2sH5nAGFyZ"
cm:name="ReplayActions" cm:type="ps:feature">
  <cm:relations cm:class="ps:parents" cm:id="iug2DMhW-zLKFtLiD">
    <cm:relation cm:id="ioQLtnxl_5aWM1RYQ" cm:type="ps:parent">
      <cm:target cm:id="iGVdrIOU0f5SgBqxB">./iKIBAeesCZsSC94WA</cm:target>
    </cm:relation>
  </cm:relations>
  <cm:properties cm:id="i-3_Bl4yxhVFLQFhx">
  [..]
  </cm:properties>
</cm:element>
```

Figure 6.13: XML Representation of ReplayActions feature in Feature Model

As one can see, the specification of this feature is similar to the other features discussed earlier.

The family model is more interesting. The XML representation for the mapped feature in the family model is shown in figure 6.14.

```
[..]
<cm:element cm:class="ps:part" cm:id="ip490phz4epBV8XpB"
cm:name="ReplayActions" cm:type="ps:feature">
  <cm:relations cm:class="ps:parents" cm:id="itiHuPfBkjBb5KuAj">
    <cm:relation cm:id="iGfOLW57ANa4D1gdb" cm:type="ps:parent">
      <cm:target cm:id="iHieJZnCSk-atHaUM">./i1qcBTPpmqBRnN2Z7</cm:target>
    </cm:relation>
  </cm:relations>
  <cm:properties cm:id="iykmKOaHe79oxpZzy">
    <cm:property cm:fixed="true" cm:id="ickfttGrCML8vk4Lt" cm:name="fid"
    cm:type="ps:feature">
      <cm:constant cm:default="true" cm:id="i4jgOzIA14zen_o1B"
      cm:type="ps:feature">iU8E5RhoUqWaFinnd/iSvar_w2sH5nAGFyZ</cm:constant>
    </cm:property>
    [..]
  </cm:properties>
</cm:element>

<cm:element cm:class="ps:part" cm:id="i21kyXgn4GkZJJwjF"
cm:name="ReplayActions2" cm:type="ps:feature">
  <cm:relations cm:class="ps:parents" cm:id="isKvk15CFvzYXC31L">
    <cm:relation cm:id="iK6mwBoroEHEsQ6yT" cm:type="ps:parent">
      <cm:target cm:id="iUDFBzA2So0OI3TEv">./iSOCma0AF_NWKWZAT</cm:target>
    </cm:relation>
  </cm:relations>
  <cm:vname>
    <cm:mimedesc cm:encoding="utf-8" cm:mimetype="text/plain">ReplayActions</
    cm:mimedesc>
  </cm:vname>
  <cm:properties cm:id="ifbsh22wfRAjKOKS0">
    <cm:property cm:fixed="true" cm:id="ivfLgZrnYDPptZa1m" cm:name="fid"
    cm:type="ps:feature">
      <cm:constant cm:default="true" cm:id="i4qIeJQNSMIgslbT6"
      cm:type="ps:feature">iU8E5RhoUqWaFinnd/iSvar_w2sH5nAGFyZ</cm:constant>
    </cm:property>
    [..]
  </cm:properties>
</cm:element>

<cm:element cm:class="ps:part" cm:id="i7N5DnNjkDiqtR2Fj"
cm:name="ReplayActions3" cm:type="ps:feature">
  <cm:relations cm:class="ps:parents" cm:id="iU4HP08ITcssG3EwI">
    <cm:relation cm:id="iiILZ8eLV8pXpgiBM" cm:type="ps:parent">
      <cm:target cm:id="i-mLLdmJSsTPIm2GP">./iNFlMqbS3RxqHKBBs</cm:target>
    </cm:relation>
  </cm:relations>
  <cm:vname>
    <cm:mimedesc cm:encoding="utf-8" cm:mimetype="text/plain">ReplayActions</
    cm:mimedesc>
  </cm:vname>
  <cm:properties cm:id="iRUBzQJRBvLCAk7Y1">
    <cm:property cm:fixed="true" cm:id="i6IF2UyJ4yk3rOlvf" cm:name="fid"
    cm:type="ps:feature">
      <cm:constant cm:default="true" cm:id="ixhz4UccyZ_QdoAVE"
      cm:type="ps:feature">iU8E5RhoUqWaFinnd/iSvar_w2sH5nAGFyZ</cm:constant>
    </cm:property>
    [..]
  </cm:properties>
</cm:element>
[..]
```

Figure 6.14: XML Representation of ReplayActions feature in Family Model

Each `cm:element` in the XML code is a mapping of the `ReplayActions` feature to a

component, which is again targeted with the id of the component. Each `cm:element` contains a `cm:property` node to make the actual mapping to the feature in the feature model through its id.

The XML representation for one of the `ReplayActionsImpl` aspects is depicted in figure 6.15.

```
[..]
<cm:element cm:class="ps:part" cm:id="iFePtotAE802o-S9r"
cm:name="ReplayActionsImpl" cm:type="ps:aspect">
  <cm:relations cm:class="ps:parents" cm:id="ikVAyXNcqd2IGPoQR">
    <cm:relation cm:id="iMmO5uQPvMJQe3wtH" cm:type="ps:parent">
      <cm:target cm:id="iFurYTokNqu1bAc22">./i7N5DnNjkDiqtR2Fj</cm:target>
    </cm:relation>
  </cm:relations>
  [..]
  <cm:vname>
    <cm:mimedesc cm:encoding="utf-8" cm:mimetype="text/
    plain">ReplayActionsImpl</cm:mimedesc>
  </cm:vname>
</cm:element>
[..]
```

Figure 6.15: XML Representation of *ReplayActionsImpl* Aspect in Family Model

The aspect targets the feature it belongs to in the same way as other relations, as we have seen earlier.

6.5 Summary

After the identification of the problems with crosscutting concerns for production plans and the solutions we propose on both the component level and production plan level, we in this chapter investigated the impact of aspect-orientation on the product line process. We studied this through the concept of generative production plans.

We started this chapter with background information on the different levels of automation of production plans and the main concepts of generative software development.

We then analyzed what generative production plans are, what the requirements are for generation of applications and which technologies are available for automation of production plans.

We have worked out the product line process for the two popular generative technologies *XML-Based Feature Modeling* and *Pure::Variants* and compared both approaches and their tool support against the defined requirements. We prefer the Pure::Variants approach in this report, because it satisfies all requirements, while the XML-Based Feature Modeling approach does not yet provide a transformation process from specification to application (skeleton).

Then the impact of aspect-orientation for the product line process was assessed and we have proposed how the domain and application engineers can work with aspect-orientation in generative production plans, based on our solutions for crosscutting concerns from the previous chapter. Affected are the family model, production plan, component library and the transformation process from production specification to application.

We finished our discussion with an application of the selected Pure::Variants tool to the AGM product line.

Chapter 7

Conclusions

As we have defined in our problem statement, the goal of this thesis was to investigate problems with crosscutting concerns for production plans in software product lines and how these crosscutting concerns can be modularized with aspects.

We have identified the problems with crosscutting concerns on two levels: on the component level, that is in the library of components from which end-products are composed through the production plan, and on the production plan level, that is in the production plan itself with scattering of variable features in the plans (chapter 4).

Aspects have been applied to the identified problems to cope with the crosscutting concerns in the product line context (chapter 5).

We have also studied the impact of aspect-orientation for the product line process with the generative approach to product line engineering which is becoming an important topic for producing software products through an automated process with a generative production plan. We have specifically analyzed the product line process for the two generative technologies *XML-Based Feature Modeling* and *Pure::variants* and the impact of aspect-orientation on the product line process (chapter 6).

In this report a case study of the Arcade Game Maker product line was introduced and used as a running example throughout the report (chapters 3 and 4).

7.1 Research Questions and Answers

We now discuss the results for each research question from the problem statement.

7.1.1 Crosscutting on the Component Level

On the component level, the problems with crosscutting concerns are threefold: the crosscutting concerns need to be modularized as aspects, the aspects should be configured for their specific variations, and the aspects need to be composed with other assets without context-specific information in the aspect implementation.

See section 4.3.4 for the discussion of crosscutting concerns on the component level.

7.1.2 Crosscutting on the Production Plan Level

A product line has a predefined set of features of which each product has a subset of these features. The product developer uses the production plan to make a correct selection of the common and variable features for the to be produced product and needs to configure each selected variable feature by choosing the relevant direct and indirect sub-features for the product. This means that the variability in the product line, that is the variable (sub-)features, is scattered throughout the production plan. The variable features are specified in the context of their parent- and sub-features and the variability in the product line is therefore not modularized.

Especially in large product lines with thousands of features it can be hard for the product developer to keep track of these variations possible for the products. See section 4.3.4 for a more in-depth discussion of crosscutting on the production plan level.

7.1.3 Application of Aspects to Identified Problems

Component Level

We deal with crosscutting concerns on the component level by implementing crosscutting concerns as aspects in the asset library. An aspect then is a separate asset in the product line and is selected for use in an application when the product should contain the crosscutting feature that the aspect implements.

An aspect which contains variability needs to be configured for the specific context the selected variations in which it is used. We have analyzed the framed aspects approach to aspect configuration. When using this approach the product developer creates (or generates) a specification frame that selects the right variations for the aspect in question. The specification frame and the aspectual asset are then input to the frame processor which produces the configured aspect.

A third issue is the composition of aspects with other code assets. Because we work in a product line context the features selected for a product can differ strongly and consequently the selection of assets from the library. Aspects do not implement crosscutting behavior on their own: the aspect needs to be bound to the relevant assets to deploy the crosscutting feature. We haved used the aspect-oriented language JasCo to separate the aspect implementation and binding to a specific context.

The reader is referred to section 5.3.2 for the analysis.

Production Plan Level

To modularize the variability in the production plan we work with XML-based feature models and have proposed application of *XQuery* for extraction of features from the model and modularize variable features. This gives handles to the product developer to get better insight in the variability in the product line and to make correct feature selections for a specific product.

We also propose that modularization of variability is supported by tooling, which means that feature selections can be made not only by traversing the hierarchy of features, but the product developer can also select variable features by providing an enumeration of feature names or select features by using wildcards. A graphical user interface should be added to the tooling to offer this functionality.

See section 5.3.2 for the analysis.

7.1.4 Generative Production Plans

To investigate the impact of aspect-orientation on the product line process we have studied what we call generative production plans: production plans that to some degree support automation of the production process. We discussed the requirements for generative production plans and the two popular generative technologies *XML-Based Feature Modeling* and *Pure::Variants* that can provide automation of the production process (see section 6.3.1).

We have worked out the product line process for both approaches and compared these against our requirements (see sections 6.3.2, 6.3.3 and 6.3.4). We prefer the *Pure::Variants* approach, because it satisfies all requirements, while the *XML-Based Feature Modeling* approach lacks support for the transformation process from product specification to application (skeleton).

The impact of incorporating aspect-orientation in the product line process was then explored for the Pure::Variants approach. We propose changes in the family model to specify aspects as separate assets in combination with configuration variables and hooks for aspect configuration and deployment respectively (see section 6.3.5) as we have investigated in our discussion of the application of aspects.

7.2 Recommendations and Future Work

Software product line engineering aims to reduce the costs of manufacturing software products by exploiting the commonalities of a product family and managing the variabilities. However, it appears that product line engineering has not yet focused on crosscutting concerns in production plans.

We think that for coping with these crosscutting concerns aspects can be applied, as aspects are already used throughout the software development cycle to modularize crosscutting concerns and to provide composition mechanisms with other concerns.

Our approach provides solutions for the identified problems with crosscutting concerns on both the component level and production plan level. On the component level, we modularize crosscutting concerns with constructs from aspect-oriented programming. Configuration of aspects for specific variations is achieved with framed aspects and a specification frame to select the right variation for each aspect. Composition of aspects with other selected assets can be handled with the concept of connectors from the JasCo language which completely separate the aspect implementation from its binding to specific context.

On the production plan level, our approach is to modularize the variability in the production plans by using XML-based feature models in the product line and the functional query language XQuery to select features by their type (common or variable) and/or name in stead of by their place in the feature hierarchy.

We have incorporated the proposed solutions in the product line process to illustrate the impact of aspect-orientation on the process. Affected are the family model, production plan, component library and the transformation process from production specification to application.

In this report we have proposed a number of additions for the Pure::Variants tool to incorporate aspects as separate product line assets, to support aspect configuration and composition of these aspects with other code assets, and to make named-based feature selections possible. We recommend to incorporate and graphically support these additions in the Pure::Variants tool.

Future work lies in the transformation process from application specification to a concrete application or application skeleton. The transformation process in the Pure::Variants approach can be modified and extended, but this is only possible in the commercial versions of the tool. Specifically the transformation of aspects would be interesting to investigate.

Bibliography

[1] Akşit, M. et al., *Abstracting object-interactions using composition-filters*, In R. Guerraoui et al., Editors, *Object-based Distributed Processing*, Springer, Verlag, 1993

[2] Aracic, Ivica, Vaidas Gasiunas, Mira Mezini, Klaus Ostermann, *An Overview of CaesarJ*, Darmstadt University of Technology, Germany, 2005

[3] Araújo, J. et al., *Early Aspects: The Current Landscape*, Technical Note, CMU/SEI-2004-TN-**xxx**, CMU-SEI, Pittsburg, 2005

[4] Arrango, G., *Domain Analysis Methods*, In Software Reusability, Schäfer, Prieto-Diaz, D. and Matsumoto, M. (Eds.), Ellis Horwoord, New York, pp. 17-49, 1994

[5] Bakker, J., *Traceability of Concerns*, Msc. Thesis, TRESE research group, University of Twente, 2005

[6] Berg, van den, K.G. et al., *AOSD Ontology 1.0 – Public Ontology of Aspect-Orientation*, Report, IST-2-004349-NOE, AOSD-Europe, 2005

[7] Berg, van den, K.G. & Conejero, J.M., *Disentangling Crosscutting in AOSD: A Conceptual Framework*, European Interactive Workshop on Aspects in Software, Brussel, 2005

[8] Bergmans, L. et al., *Aspect Composition using Composition Filters*, In Akşit, M., Editor, Software Architectures and Component Technology, Kluwer Academic Publishers, Dordrecht, 2002

[9] Brune, K. et al., *C4 Software Techology Reference Guide – A Prototype*, Handbook, CMU/SEI-97-HB-001, CMU-SEI, Pittsburg, 1997

[10] Buschmann, F. et al., *Pattern-Oriented Software Architecture. A System of Patterns*, John Wiley & Sons Ltd., Chichester, UK, 1996

[11] Cechticky, V. et al., *XML-Based Feature Modelling*, Int. Conf. on Software Reuse, Madrid, Spain, pp. 101-114, LNCS 3107

[12] Chastek, G. & McGregor, J., *Guidelines for Developing a Product Line Production Plan*, Technical Report, CMU/SEI-2002-TR-006, CMU-SEI, Pittsburg, 2002

[13] Clemente, Pedro J. & Hernández, Juan, *Aspect Component Based Software Engineering*, 2nd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS), 2003

[14] Clements, P. & Northrop, L., *Software Product Lines: Practices and Patterns*, Boston, Addison-Wesley, 2002

[15] Czarnecki, K. & Eisenecker, U., *Generative Programming Methodes, Tools and Applications*, Addison-Wesley, 2000

[16] K. Czarnecki. *Overview of Generative Software Development*, In J.-P. Bantre et al. (Eds.): Unconventional Programming Paradigms (UPP) 2004, Mont Saint-Michel, France, LNCS 3566, pp. 313328, 2005

[17] Elrad, T. et al., *Aspect-Oriented Programming*, Comm. of the ACM, v. 44, n. 10, pp 29 – 32, 2001

[18] *Feature-Based Framework Modeling*, http://control.ee.ethz.ch/~ceg/fbfm/doc/Overview.html, ETH-Zürich, August 2005

[19] *Open Source Native XML Database*, http://www.exist-db.org, June 2006

[20] Filman, R.E. et al., *Aspect-Oriented Software Development*, Boston, Addison-Wesley, 2005

[21] Gamma, E. et al., *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995

[22] Han, J., *An Approach to Software Component Specification*, Proc. 21st International Conference on Sotware Engineering, Los Angeles, 1999

[23] Hannemann, J. and Kiczales, G., *Design Pattern Implementation in Java and AspectJ*, Proceedings of the 17th Annual ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pages 161-173, November 2002

[24] Harrison, W. et al., *Concern Modeling in the Concern Manipulation Environment*, Workshop on Modeling and Analysis of Concerns in Software, 2005

[25] Hürsch, W. & Lopes C., *Separation of Concerns*, Technical Report, College of Computer Science, Northeastern University, 1995

[26] Institute of Electrical and Electronics Engineers. IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. New York, 1990

[27] Jacobsen A. & Krämer, B., *Modeling Interface Definition Language Extensions*, In Proc. Technology of Object-Oriented Languages and Systems - TOOLS Pacific, Sydney, pp 241 – 252, 2000

[28] Kang, K. et al., *Feature-Oriented Domain Analysis (FODA) Feasibility Study*, Technical Report, CMU/SEI-90-TR-21, CMU-SEI, Pittsburg, 1990

[29] Kiczales, G. et al., *An Overview of AspectJ*, In Proceedings of ECOOP 2001, LNCS 2072, Springer-Verlag, Berlin, 2001

[30] Kiczales, G. et al., *Getting Started with AspectJ*, Comm. of the ACM, v. 44, n. 10, pp 59 – 65, 2001

[31] Krueger, P., *Variation Management for Software Product Lines*, in: Proc. of Second Software Product Line Conference, 2002

[32] Lohmann, D. & Ebert, J., *A Generalization of the Hyperspace Approach Using Meta-Models*, In Proceedings of the 2003 AOSD Early Aspects Workshop, Boston, 2003

[33] Loughran, N. et al., *Supporting Product Line Evolution with Framed Aspects*, 3th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS), 2004

[34] Loughran, N. et al., *Framed Aspects: Supporting Variability and Configurability for AOP*, 8th International Conference on Software Reuse (ICSR), 2004

[35] McGregor, John D., *Arcade Game Maker Product Line*, http://www.cs.clemson.edu/~johnmc/productLines/example/frontPage.htm, Clemson University, October 2005

[36] Mezini, Mira and Ostermann, Klaus. *Conquering Aspects with Caesar*, In (M. Aksit ed.) Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD), March 17-21, 2003, Boston, USA. ACM Press, pp. 90-100.

[37] Mezini, Mira & Ostermann, Klaus, *Variability Management with Feature-Oriented Programming and Aspects*, in Proceedings SIGSOFT 2004, Newport Beach, CA, USA, 2004

[38] *COM: Component Object Model Technologies*, *http* : $//www.microsoft.com/com/default.mspx$, February 2006

[39] Mikhajlov, L. and Sekerinski, E., *A Study of The Fragile Base Class Problem*, in Proceedings ECOOP '98, Lecture Notes in Computer Science, 1445, Springer-Verlag, pp. 355 – 382, 1998

[40] Nagy, István, Bergmans, Lodewijk, Akşit, Mehmet, *Composing Aspects at Shared Join Points*, in Proceedings International Conference NetObjectDays 2005, Erfurt, Germany, 2005

[41] Northrop, L., *SEI's Software Product Line Tenets*, IEEE Software 19(4), pp. 32-40, 2002

[42] Object Management Group (OMG), *CORBA 3.0 specification*, *http* : $//www.omg.org/technology/documents/formal/corba_2.htm$, November 2005

[43] Ossher, H., Tarr, P., *Multi-Dimensional Separation of Concerns in Hyperspace*, Research Report RC21452(96717)16APR99, IBM Research Division, Almaden, 1999

[44] Parnas, D.L., *On the Criteria to be used in Decomposing Systems into Modules*, Commun. ACM, v. 15, n. 12, 1972

[45] Pinto, Mónica, Fuentes, Lidia, Troya, Jose María, *DAOP-ADL: An Architecture Description Language for Dynamic Component and Aspect-Based Development*, in Proc. 2nd International Conference on Generative Programming and Component Engineering (GPCE), Erfurt, Germany, 2003

[46] *pure::variants, variant management*, *http* : $//www.pure-systems.com/Variant_Management.49.0.html$, May 2006

[47] *pure::variants Eclipse Plug-in User's Guide*, Version 1.1 for pure::variants 2.0, $http://www.pure-systems.com/Documentation.93.0.html$, 2005

[48] Salinas, P., *Adding Systematic Crosscutting and Superimposition to Composition Filters*, EMOOSE Msc. thesis, Vrije Universiteit Brussel, Brussel, 2002

[49] *Enterprise JavaBeans Technology*, *http* : $//java.sun.com/products/ejb/$, February 2006

[50] Sutton, S., Rouvellou, I., *Modeling of Software Concerns in Cosmos*, In Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD), Enschede, The Netherlands, 2002

[51] Sutton, S., Rouvellou, I., *Concern Modeling for Aspect-Oriented Software Development*, in Aspect-Oriented Software Development, R. Filman, et al., Editors, Addison-Wesley, 2004

[52] Suvée, Davy, Vanderperren, Wim, Jonckers, Viviane, *JasCo: an Aspect-Oriented approach tailored for Component Based Software Engineering*, 2nd International Conference on Aspect-Oriented Software Development (AOSD), Boston, USA, 2003

[53] Tarr, P. et al. *N Degrees of Separation: Multi-Dimensional Separation of Concerns*, 21st Int. Conf. on Software Engineering, ACM, New York, 1999, pp. 107 – 119

[54] Trigaux, J. & Heymans, P., *Modelling variability requirements in Software Product Lines: a comparative survey*, EPH3310300R0462/215315, FUNDP – Equipe LIEL, Namur, 2003

[55] Wagelaar, D., *A Concept-Based Approach for Early Aspect Modelling*, Vrije Universiteit Brussel, Brussel, 2003

[56] Wang, B. et al., *A Generative and Component based Approach to Reuse in Database applications*, Net.Objectdays, Erfurt, Germany, 2003

[57] Weiss, R.J., *Reuse in the Software Development Process*, Universität Tübingen, Tübingen, 2003

Appendix A

Concern Modeling

A.1 Introduction

Separation of Concerns is an important principle in software engineering for decomposition and structuring of separate concerns in software (see also section 2.3.2). Several modeling approaches for 'regular' concerns as well as crosscutting concerns have been developed, of which a number of approaches are discussed here: *Hyperspaces*, *Cosmos*, *Extended Hyperspace Model*, *CoCompose*, and the *Concern Manipulation Environment*.

A.2 Hyperspaces

Tarr et al. [53] state that the primary goals of software engineering are to improve software quality, to reduce the costs of software production, and to facilitate maintenance and evolution. To reach these goals reduced software complexity, improved comprehensibility, promotion of reusability and facilitation of evolution are needed.

Reduced complexity and improved comprehensibility require *decomposition* mechanisms to divide software into meaningful and manageable pieces. They also require *composition* mechanisms to put pieces together usefully. Reuse requires the development of large-scale reusable components, low coupling, and powerful, *non-invasive* adaptation and customization capabilities. Ease of evolution depends on low coupling and also requires *traceability* across the software lifecycle, mechanisms for minimizing the impact of changes, and substitutability. The IEEE Standard Computer Dictionary defines traceability as "*the degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another*" [26].

Tarr et al. identify a number of problems that complicate software engineering [53]:

- Software comprehensibility tends to degrade over time.
- Many common maintenance and evolution activities result in high-impact, invasive modifications.
- Artifacts are of limited reusability, or are reusable only with difficulty.

- Traceability across the various software artifacts is limited, which further complicates evolution.

According to Tarr et al., these problems are in large part due to limitations and unfulfilled requirements related to separation of concerns due to which not *all* concerns of importance in software systems are separated. Existing formalisms for separation of concerns provide only small, restricted sets of decomposition and composition mechanisms, and these typically support only a single, "dominant" dimension of separation at a time. This is called the "tyranny of the dominant decomposition".

Tarr et al. state that to overcome these problems support is needed for *multi-dimensional separation of concerns*. They propose a model of decomposition and composition and introduce the concept of *hyperslices*. A hyperslice is a "*set of conventional modules, written in any formalism*"[53]. Hyperslices are intended to encapsulate concerns in dimensions other than the dominant one. The modules within it contain all, and only, those units that pertain to, or address, a given concern. Hyperslices can *overlap*, in that a given unit may occur, possibly in different forms, in multiple hyperslices. A system is written as a collection of hyperslices, thereby separating all the concerns of importance in that system, along as many dimensions as are needed.

After the decomposition of a software system in a number of hyperslices, these need to be composed. This is where the concept of a *hypermodule* comes into the picture. A hypermodule is a "*set of hyperslices, together with a composition rule that specifies how the hyperslices must be composed to form a single, new hyperslice that synthesizes and integrates their units*" [53]. Because of this composition property, a hypermodule is appropriate wherever a hyperslice may be used. Hypermodules can thus be nested and thereby promote abstraction and encapsulation. Composition is based on commonality of concepts across units: different units describing the same concept are composed into a single unit describing that concept more fully. This process involves three steps: *matching* units in different hyperslices that describe the same concept, *reconciliation* of differences in these descriptions, and *integration* of the units to produce a unified whole. It is the task of the composition rule in the hypermodule to specify the details of composition.

A composition rule is a combination of a concise, general rule, and detailed, specific rules that specify exceptions to the general rule or handle cases that it cannot handle. Generally, units of different hyperslices are matched by name. Detailed rules take care of more complex matchings. Composition rules can be defined at the hypermodule level, or per hyperslice to allow each hyperslice itself to specify how it is to be composed. However, putting the composition rule at a higher level allows more flexible overlap en enhanced reuse [53].

To use the hyperspaces model, one must *instantiate* it for particular artifact development formalisms. Instantation entails determining which notational constructs map to units and modules, deciding how to represent hyperslices, and providing support for composition of hyperslices. Tackling the issues involved in this process is the job of the software engineers, based on the specific context of the software system to be developed.

A.3 Cosmos

Although progress has been made on "advanced separation of concerns" (ASOC), such as multi-dimensional separation of concerns [53] and aspect-oriented program-

ming [17], Sutton & Rouvellou identify a number of problems in current concern modeling approaches [50][51]:

- Current ASOC tools provide only limited support for explicit concern modeling.
- Representations of concerns tend to be tied to particular tools or artifacts.
- Concern modeling usually occurs just in the context of a particular type of development activity, such as coding and design, and not in the entiere life cycle.

Sutton & Rouvellou state that concern modeling should play a central role throughout the development life cycle and propose *Cosmos*, a general-purpose concern-space modeling schema. The goals of Cosmos are [50][51]:

- Support the representation of arbitrary concerns.
- Support the representation of *composite* concerns.
- Support the representation of arbitrary *relationships* among concerns.
- Support the *association* of concerns to arbitrary software units, work products, or system elements.
- Be language independent; that is,
  - Not depend on any particular programming language or other development formalism.
  - Accommodate different development formalisms appropiate to different stages of the life cycle.
  - Be able to capture information that is not necessarily reflected in particular development formalisms.
- Be methodology independent.
- Be applicable across the software life cyle.
- Support a variety of types of software engineering tasks (as appropriate to the development methods in which it is used).

Cosmos models software concern spaces in terms of *concerns*, *relationships* and *predicates*. See figure A.1 for an outline of the elements in the Cosmos concern model. Note that items in normal font are part of the core schema and items in italic font are representative schema extensions used in particular concern models.

Cosmos divides concerns into two main categories, logical and physical. Logical concerns represent conceptual concerns; physical concerns deal with "real world concerns, such as work products, software units, hardware units, and services.

Logical concerns are further categorized as classifications, classes, instances, properties, and topics. Classifications are for modeling systems of classes and allow for multiple classifications of concerns. Classes are for categorization of concerns. Instances represent particular concerns. Properties are characteristics, such as performance, configurability, etc. Topics are groups of concerns of generally different types, typically related to a theme of user interest.

- Concerns
  - Logical
    - Classifications
    - Classes
    - Instances
    - Properties
    - Types
  - Physical
    - Collections
    - Instances
    - Attributes
  - Predicates
  - Relationships
  - Groups
- Predicates
  - // subtypes not elaborated
- Relationships
  - Categorical
  - Classification
  - Generalization
  - Instantiation
  - Characterization
  - Topicality
  - Attribution
  - Membership
- Interpretive
  - *Contribution*
  - *Motivation*
  - *Admission*
  - *Logical implementation*
  - *Logical composition*
  - *Logical requisition*
- Physical
  - Physical association
  - *Physical requisition*
- Mapping
  - Mapping association
  - *Physical implementation*

Figure A.1: Outline of Cosmos Concern Model Elements [51]

Physical concerns comprise instances, collections, and attributes. Physical instances represent particular system elements. Collections represent groups of physical instances. Attributes are the specific properties of instances or collections.

Relationships are divided into four categories: *categorical*, *interpretive*, *physical*, and *mapping*. Categorical relationships reflect fundamental semantics of the concern categories, for example generalization. Interpretive relationships relate logical concerns according to user-assigned semantics, for example contribution, which indicates that one concern contributes in some way to another concern. Physical relationships associate physical concerns. Mapping relationships represent (non-categorical) associations between logical and physical concerns. These are important – along with interpretive relationships – for purposes such as dependency analysis, impact assessment, and change propagation.

Predicates represent integrity conditions over various relationships and can be classified accordingly.

When modeling a software system with Cosmos crosscutting concerns (see section 2.3.3) can be identified by looking at recurring themes under various kinds of concern. Such crosscutting concerns represent other dimensions by which the concern space can be organized and topics are one way to represent them [50][51].

A.4 Extended Hyperspace Model

During software development, different concerns have to be taken into account in the different life cycle stages of the development process. These different concerns can be understood as different *views* to the system, or as Tarr et al. call them, *dimensions of a concern space* [53]. Concerns are usually described is some formalism.

Lohmann & Ebert call these formalisms *artifact languages* [32]. They state that there is a strong relationship between views/dimensions and artifact languages, and that for succesful multi-dimensional seperation of concerns (MDSoC) over the whole software development cycle, one therefore needs a model that supports on-demand integration of artifacts and artifact languages. They find that current attempts for MDSoC and AOSD mainly focus on single languages and mainly on the implementation phase.

Lohmann & Ebert propose the *Extended Hyperspace Model* as an approach to MDSoC by language integration using *meta-models* and the original Hyperspaces idea [53]. An ideal artifact language describes just one kind (dimension) of concerns. Each artifact language, like the UML family of languages and its sublanguages, consists of a set of syntactical constructs that can be used to build concrete artifacts. Lohmann & Ebert call these syntactical constructs *unit types* and state that syntax and semantics of an artifact language itself have to be defined in some formalism: *meta-modeling*. Each syntactical concept (unit type) of the language is represented by a meta-class in the meta-model, and each relationship between them is represented by a meta-association.

In the (Extended) Hyperspace model, units and concerns are organized in a concern space. A concern space "*consists of a set of dimensions, a set of concerns and a set of units. Each concern is placed in exactly one dimension. Each unit is mapped to zero or more concerns, namely the concerns it addresses*" [32]. The aim of a concern space is to integrate units and concerns of a software system in such a way, that concerns can be easily identified and separated, that relationships between different concerns become clear and that a software system can be built out of selected concerns [43]. Integration of artifact languages corresponds to integration of the underlying meta-models. The key benefit of integration is the extra knowledge it provides about inter-dimensional connections and dependencies that is not available from the original artifacts.

To get additional views on a the concern space, Lohmann & Ebert introduce the notion of primary and secondary concerns. *Primary concerns* are artifact-based concerns, ie. a `class` in a class-diagram, and their dimensions are called *primary dimensions*, ie. a `<Classes>` dimension. Concerns and dimensions which are not (yet) represented by their own artifacts and artifact languages, ie. special concerns stemming from the application domain or "on-demand" dimensions for additional user- or task-oriented views, are called *secondary concerns* on their *secondary dimensions*. Being nog based on artifact languages, secondary dimensions are not meta-modeled. However, the units from the primary dimensions can be mapped to the secondary concerns to which they relate. In this manner, secondary dimensions provide *on-demand* and *alternative separation of concerns* along arbitrary dimensions [32].

A.5 CoCompose

Aspects or crosscutting concerns (see section 2.3.3) are expressed in many different ways by current aspect-oriented languages. A few examples are the join point mechanism used by AspectJ [29], the hyperspaces mechanism [53] (see section A.2), and the composition filters mechanism [8] (see section 4.2.3). One specific aspect-oriented mechanism may be more efficient to use than another depending on the application context.

According to Wagelaar this diversity in representing aspects contributes to several problems when trying to model aspects in an early stage of the software lifecycle,

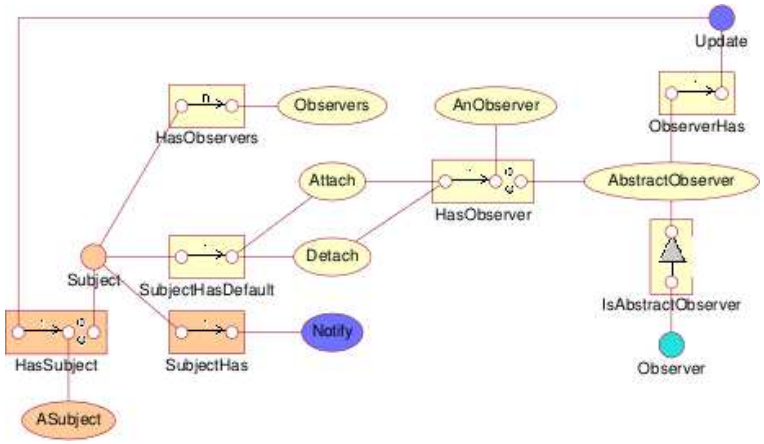

Figure A.2: An Observer Composite Solution Pattern [55]

for example [55]:

- In current object-oriented software modeling approaches, the implementation semantics (e.g. method, class, aspect) of each software element must be determined when introducing the element, even though it may be hard to choose the exact semantics at that moment.
- If there are gaps in the process of automated code generation from sofware models, these may require manual transformations between software lifecycle stages. If, in addition, the software structures in different lifecycle stages differ, one local update in one stage may require numerous updates in the other. The original update is not well *traceable* in the other stage either. For example, Cosmos (see section A.3) does not commit to implementation constructs initially, but still, the transformation of software models described using such early stage approaches must be done manually.

Wagelaar proposes *CoCompose* as a software modeling approach that supports aspects or crosscutting concerns [55]. The CoCompose modeling language introduces *concepts* as a central modeling element to overcome the problem of early commitment to implementation constructs. *Composites* are introduced as a reusable abstraction of design concepts, such as inheritance, superimposition, or a specific design pattern. Instances can be used to connect the concepts into a coherent model.

Concepts can be mapped to implementation constructs such as aspects, classes, instances, parameters, attributes, methods or pointcut designators. Concepts can contain embedded implementations in order to avoid making each detail of the software system explicit as a concept. These implementations are expressed in an implementation language to which the developer wants to map. Concept implementations can have constraints to describe the extent of their applicability.

Composites can contain *composite roles* and *published concepts* in their interface. Composite roles describe the role a concept plays in the relationship defined by the composite. Published concepts represent concepts exported by the composite. The definition of a composite can contain embedded descriptions of the structure of the design concept represented by the composite. This structure, called *solution pattern*, again consists of concepts and composites. See figure A.2 for an example solution pattern in the CoCompose language.

Each composite's definition can also contain *implementation generators*, which can implement the composite in a specific implementation language. By being able to define several solution patterns and implementation generators for one composite, the composite isn't tied down to a single implementation. It represents a high-level element that can be implemented in several ways [55].

In order to close gaps in automated code generation, an automated process has been developed for translating CoCompose models into specific implementation languages. This process is based on Design Algebra [48] and consists of three steps [55]:

- *Flattening the model*: eliminating composites through the application of their solution patterns. Since each composite can have several solution patterns, this will result in a set of models.
- *Determining the implementation form* (e.g. class, method) of each concept. The possible forms for implementing a concept are determined by (1) the form contraints of implementation generators and (2) the available embedded implementations of that concept and (3) the form constraints on the chosen embedded implementations for other concepts.
- *Generating the code* for the transformed model. First, a skeleton implementation is generated, based on the chosen implementation forms of all concepts. Any embedded implementations of the concepts are inserted. Then, the implementation generators for each composite are applied.

Wagelaar has created a prototype CoCompose tool for visual modeling, automated translation into an implementation language (currently Java or ConcernJ), and managing a repository of composites for reuse [55].

A.6 Concern Manipulation Environment

The Concern Manipulation Environment (CME) is an AOSD environment in which software is organized and manipulated in terms of concerns. CME is a tool-based approach to multi-dimensional concern modeling across the software life cycle and supports the identification, definition, encapsulation, extraction, and composition of concerns [24].

A.7 Summary

In this chapter we have studied a number of popular concern modeling approaches. The Hyperspaces approach states that support is needed for *multi-dimensional separation of concerns* to overcome the problem of a single, 'dominant' dimension of separation at a time. The approach proposes a model of decomposition and composition and introduces the concept of *hyperslices* as a set of conventional modules, written in any formalism. Hyperslices are intended to encapsulate concerns in dimensions other that the dominant one.

A second approach is Cosmos which is a general-purpose concern modeling schema and models concerns, different types of relationships between concerns, and predicates which represent integrity conditions over various relationships.

We have also discussed the Extended Hyperspace Model that introduces the concept of a *concern space* that consists of a set of dimensions, a set of concerns and a set

of units. Each concern is placed in exactly one dimension. Each unit is mapped to zero or more concerns, namely the concerns it addresses. In this way concerns and units are integrated and software systems can be built out of selected concerns through this mapping.

The CoCompose approach focuses on modeling crosscutting concerns next to the 'regular' concerns. Also mappings can be made from concerns (called *concepts*) to their implementation (called *solution patterns*).

We finished this chapter with the Concern Manipulation Environment which is an Eclipse plugin for modeling concerns.

Appendix B

Class Diagrams for AGM Case

For better understanding of the structure of the code assets in the AGM Product Line Case, this chapter shows the class diagrams of all the relevant packages and the classes within. These class diagrams have been generated from the actual C# code for the games using *Borland Together Architect*. Thanks go out to John D. McGregor – the creator of the AGM Case – for providing the code assets.

B.1 GameDefinitions package

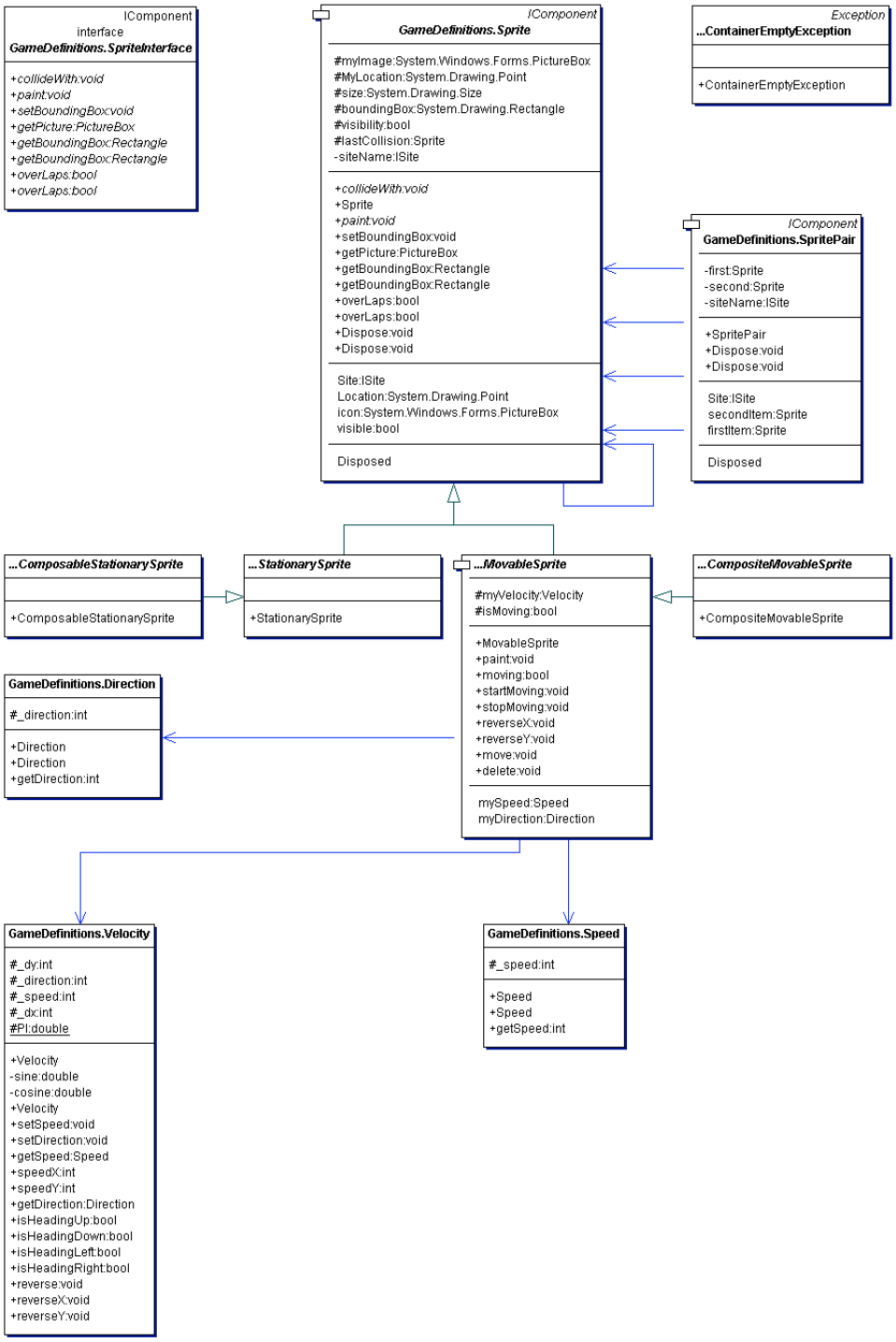

Figure B.1: Class Diagram for the GameDefinitions package

B.2 GameBoard package

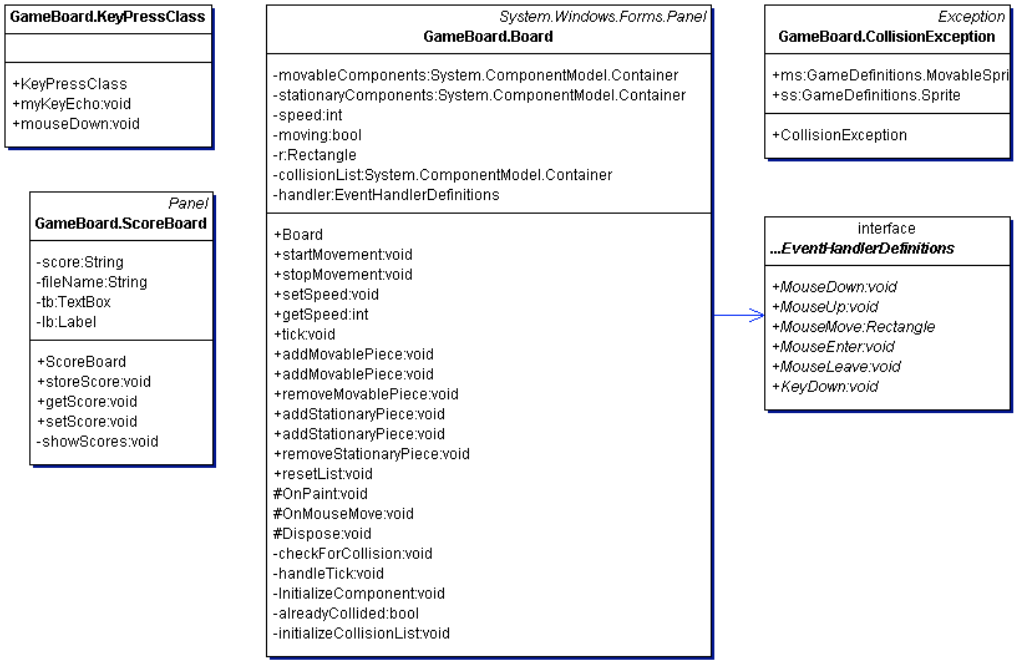

Figure B.2: Class Diagram for the GameBoard package

B.3 BricklesDefinitions package

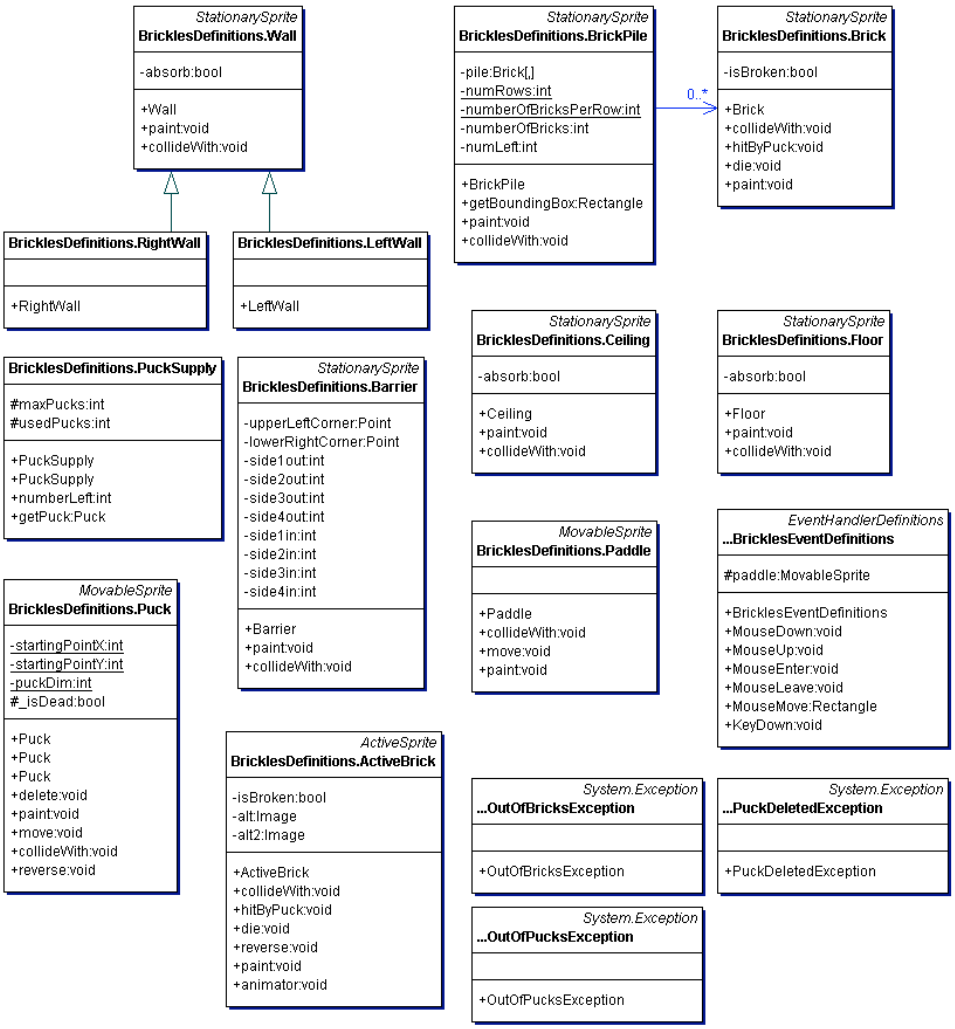

Figure B.3: Class Diagram for the BricklesDefinitions package

B.4 Brickles package

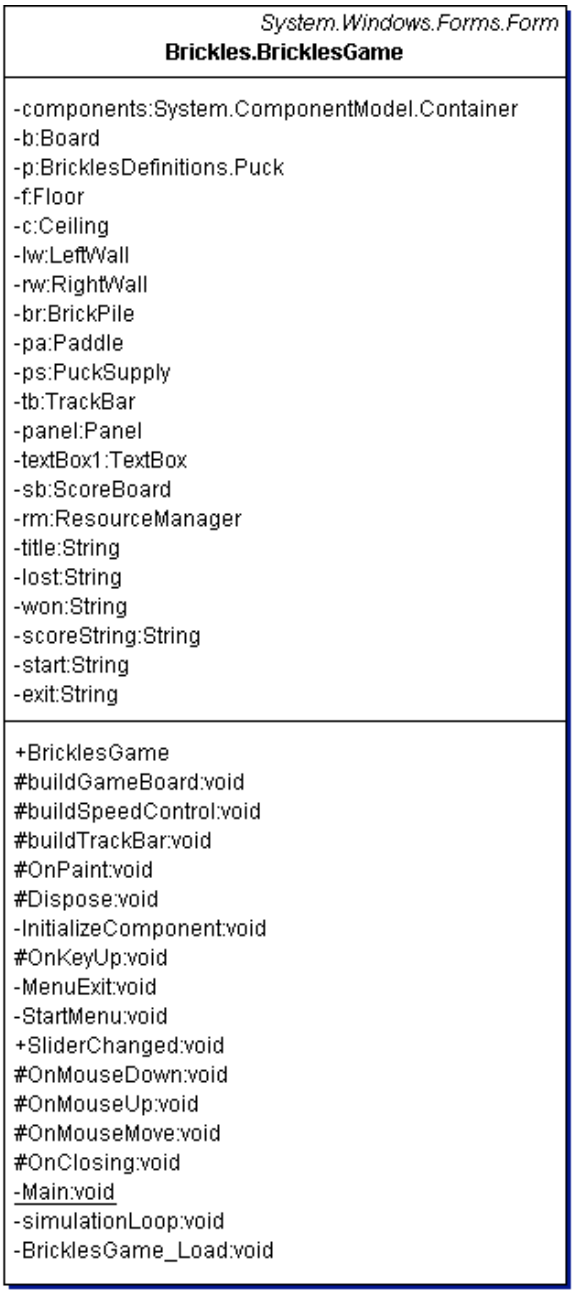

Figure B.4: Class Diagram for the Brickles package

B.5 PongDefinitions package

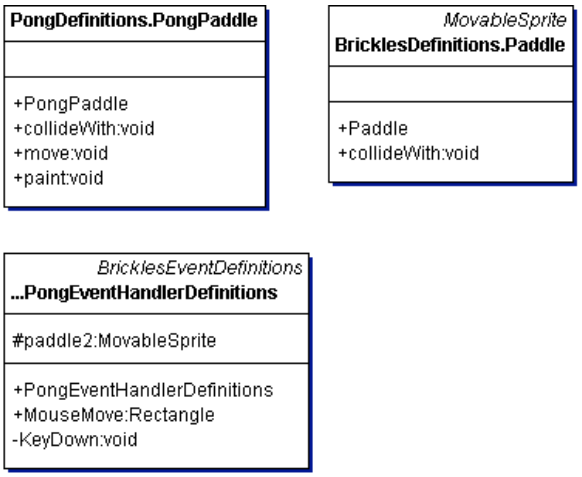

Figure B.5: Class Diagram for the PongDefinitions package

B.6 Pong package

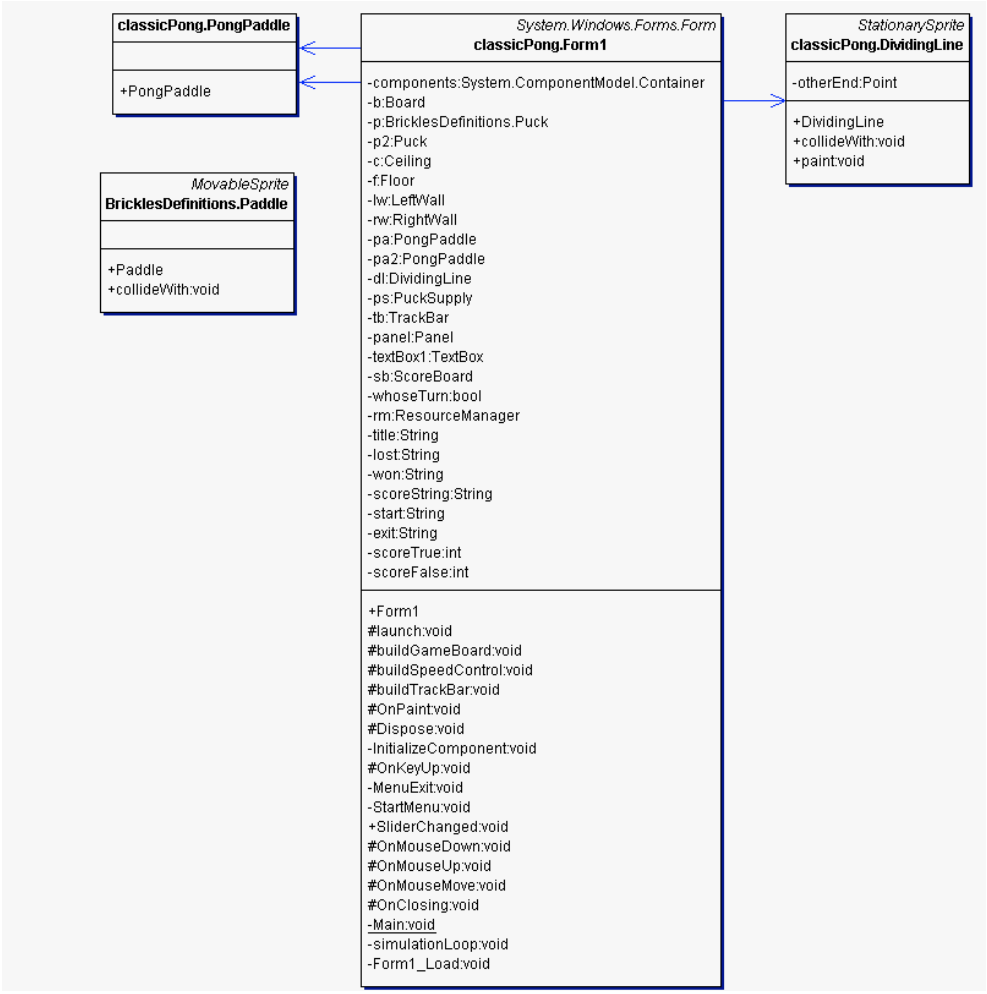

Figure B.6: Class Diagram for the Pong package

B.7 BowlingDefinitions package

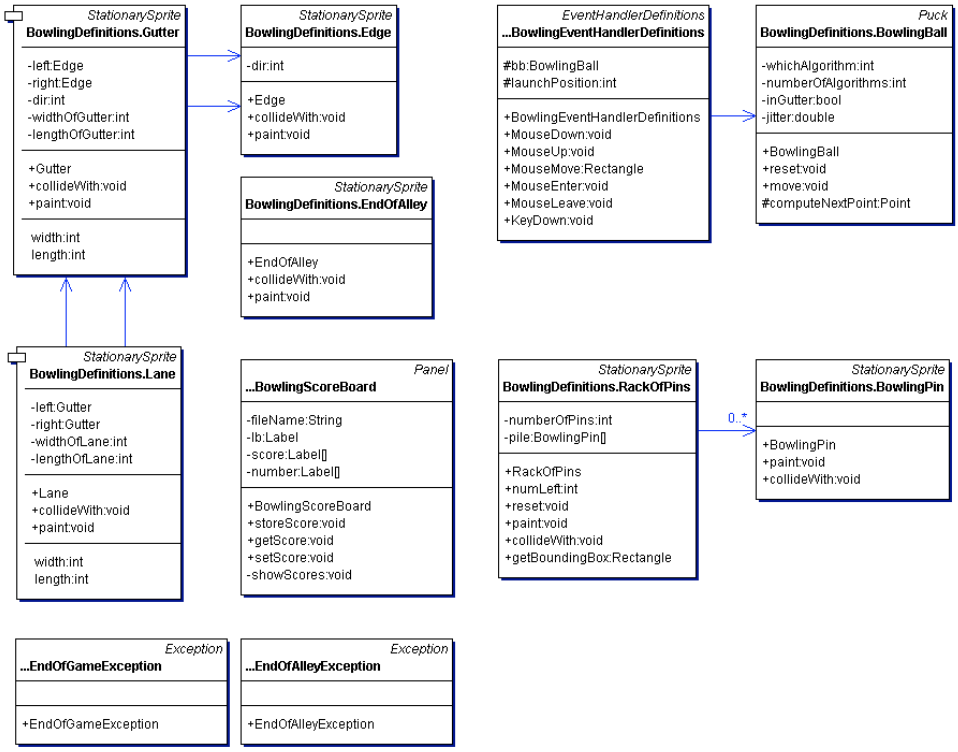

Figure B.7: Class Diagram for the BowlingDefinitions package

B.8 Bowling package

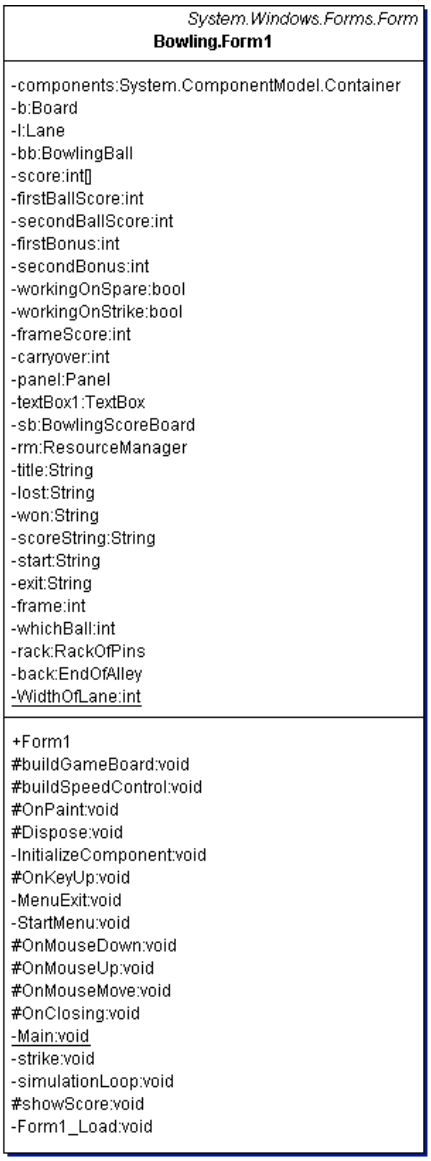

Figure B.8: Class Diagram for the Bowling package

Appendix C

Production Plan for AGM Case

This appendix includes a copy of the relevant information in the generic production plan from the AGM Product Line [35]. The document is discussed in the Case Study in chapter 3.

Carnegie Mellon
Software Engineering Institute

Arcade Game Maker Production Plan

John D. McGregor

August 2003

Table of Contents

1 Overview

1.1 Identification

The Arcade Game Maker Product Line organization will produce a series of arcade games ranging from low obstacle count to high with a range of interaction effects. More detailed information can be found in the Arcade Game Maker scope document. This document describes how a product is produced in the AGM product line.

1.2 Document Map

The Arcade Game Maker Product Line is described in a series of documents. These documents are related to each other as shown in Figure 1. This map shows the order in which the documents should be read for the first time. Once the reader is familiar with the documents, the reader can go directly to the information needed.

This is the production plan. Product Line organizations use this document to capture how the product teams will build a new product. This document follows the outline provided in [Chastek 02].

1.3 Using this document

This document is a template for product-specific production plans. Chapter 6 is the site of the major items that must be modified to make this the production plan for a specific product.

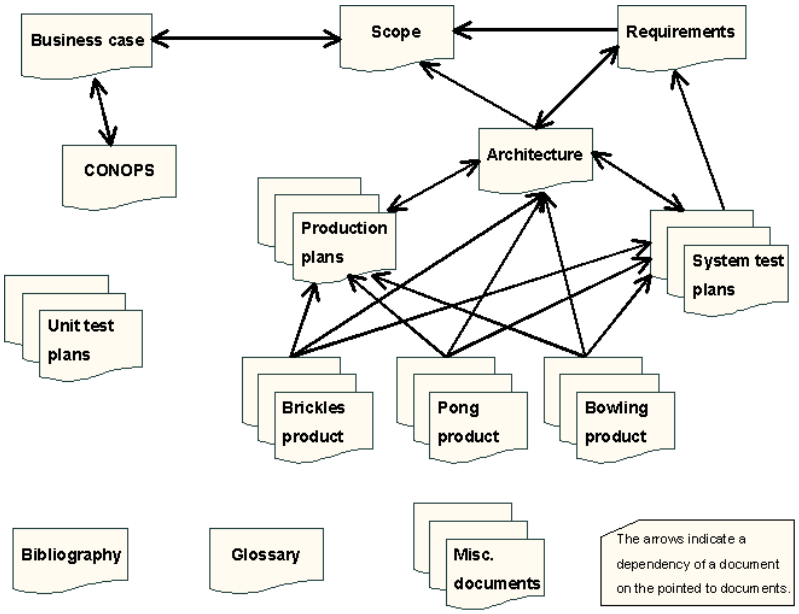

Figure 1- Document Map

1.4 Concepts

See the Glossary document for definitions of basic concepts.

1.5 Readership

This document is intended to provide some level of information to all of the stakeholders in the Arcade Game Maker framework but is primarily intended for product development teams. The production plan describes for a manager the resources that are needed to produce a product. Technical members of the organization can use the production plan as the detailed process for producing a product.

2 Strategic view of product development

2.1 Assumptions

- The company has an existing pool of software developers who are highly technical. They have fielded products on a variety of hardware platforms and are accustomed to being involved down to the driver level of the software.

- The product line contains games that are similar in content but that differ in platform. Differences in platform translate into differences in graphics implementation, which is a major feature of these products.

2.2 Qualities

We will briefly discuss two types of properties: product qualities and production process qualities.

2.2.1 Product qualities

The products must be enjoyable to play in order to be a success. This requires both a colorful display and realistic action.

- Any new game elements should add to the quality of the display. It should be colorful and representative of the item it represents.

- The action of the game must proceed sufficiently fast to demand the player's attention. When constructing a game, if the number of elements slows the game, alternatives must be investigated.

- The action of the game must look like what the player expects. The motion and reactions of movable elements must be realistic. As elements are added to the game, their actions and their boundaries must be correctly set through parameters so that collisions appear real.

2.2.2 Production process qualities

The production process in the AGM product line is largely manual. The personnel are very technical so the process allows for hands-on manipulation of the product. To provide these technical personnel with a tool that automates the production process to a high degree would have been a waste of their talent, frustrating for them, and a poor use of the resources needed to produce the automation.

2.3 Products possible from available assets

The products that are possible from the available assets only include games like those constructed so far. Using the incremental approach the asset base is only developed to the extent needed to construct the current set of products. As additional increments are completed, the variety of games that are possible will greatly increase.

2.4 Production strategy

The production strategy for the AGM product line includes a domain-based design approach and a manual construction approach. The analysis of requirements and development of the architecture will be based on domain information. Each new product will be built by having software developers manually specialize assets where necessary, gather other domain-based, core assets that are needed and then build an executable by running a compiler and linker.

The core assets are being built incrementally. The product teams for the earlier products will have fewer assets to choose from than will the teams on later products. For earlier product teams, one responsibility will be to identify candidates for core assets in the later increments. The freeware development will provide many opportunities for this.

The formal statement of the strategy[1] is:

We will position ourselves as the leading provider of rapidly customized, high-performance, low cost games by producing products that are easily modified, have better performance than our competitors, are sufficiently low cost to deter potential entrants from entering the market, and require sufficiently few resources to allow their use on any embedded computer. We will produce the initial products using a traditional iterative, incremental development process using a standard programming language, IDE, and available libraries. We will create domain-based assets, including a product line architecture and software components, for the initial products in a manner that will support a migration to automatic generation of the second and third increment products.

[1] See memo 04-01 for the complete rationale for this production strategy

3 Overview of available core assets

The available core assets include analysis, design, and implementation assets.

3.1 Naming conventions

Names that end in “Interface” are interface definitions.

Names that start with “test” are DotUnit test case suites.

3.2 Analysis-level assets

The requirements document includes several useful assets.

- The domain analysis model organizes the concepts in the main domain, games, and provides the attributes of each concept as well as the relationships among them. Developers who are new to the domain should study this asset to understand the background for the product on which they are working.
- The feature model shows the features of products in the product line.
- The use case model provides a superset of requirements for products. Select the appropriate use cases for your product.

3.3 High-level design assets

The main high-level design asset is the software architecture. The architecture is described in detail in the two-volume architecture description.

3.4 Source code

Each of the interfaces in the existing architecture has been implemented. The C# components are named according to the names given in the architecture documentation.

3.5 Test cases

Test cases may be reused in building a product when the tests cover functionality or features that are included in the new product. This section describes existing test cases.

Test cases are not available for all assets yet. Here we describe the current status of test assets.

3.5.1 Unit tests

Individual unit tests are constructed using the DotNet testing framework. The source code is in the form of classes. These tests are available from the Unit Test icon on the Document Roadmap.

Unit test classes are currently available for:

- Velocity/Speed/Direction cluster
-

3.5.2 Integration tests

Integration of units that result in a component is tested at the API level. Integration of units that result in GUI level is tested using the same procedure as the system test.

The current integration tests that are available are:

3.5.3 System tests

System tests are currently administered by hand. Each system test is a scenario that is derived from a specific use case. When a use case is used on more than one product, the related test cases can also be used on that new product. These test cases are documented in the test plan for a product.

Test plans are currently available for:

- Brickles
-

3.6 Basic inputs and dependencies

3.6.1 Inputs

- The main inputs to each game are mouse and keyboard events.
- When the change case for saving a game is implemented, an additional set of inputs will be the data saved to the file.

3.6.2 Dependencies

- The game products are heavily dependent upon the graphics library of the programming language.
- When the change case for saving a game is implemented, the game products will be dependent on the operating system.

3.7 Variations

3.7.1 Absorbing vs reflecting

The stationary game elements participate in the game by providing behavior during a collision. There are two major behaviors available in the core assets at the moment. A stationary element may reflect the movable element according to the laws of physics or the element may absorb the movable element and cause it to be deleted from the game. A parameter on each element determines which of these behaviors is performed.

3.7.2 Event Handling

The event handling routines vary from one game to another. An implementation of the EventHandlerDefinitions interface is provided as a parameter to the GameBoard component. Each of the mouse and keyboard events are handled in a way defined in this implementation.

4 Detailed production process

This step-by-step process was developed as the AGM developers built the Brickles and Pong freeware games. It will be used by future product teams.

4.1 Outline

The process for producing a new product has five major steps:

- Product definition and identification
- Incremental analysis
- Design product
- Build product
- Test product

4.2 The process

4.2.1 Product identification, definition and analysis

1. The products were identified as part of the product line planning process. Since each product is a single game, identifying the game to be implemented identifies the product.
2. The definition of the game consists of defining the rules of the game. There are several versions of most of these games so the product team must first decide on a set of rules to implement.
3. Analyze features for the new game that are variations from previous games; Identify existing features that must change for this game

4.2.2 Design new game

1. Plan how to provide those features from existing components
2. Plan how to provide the remaining features from new assets

3. Design the new implementation of the EventHandlerDefinitions interface.

4.2.3 Build Code

1. Start new ClassLibrary in a Visual Studio Project using {game name}Definitons as its name – Use this for new classes other than the game definition itself
2. Start new Windows Application in a Visual Studio Project using the name of the game.
3. Copy the Form1.cs file from a previous product line project
4. Change the namespace name to the new game
5. Configure the new gameboard
6. Copy data.txt from a previous game's working directory. This is the resource file for the game.
7. Edit data.txt to reflect the new game
8. Compile the resource file
9. Copy the compiled resource file to the Debug directory

4.2.4 Test the New Game

1. Each core asset is tested, either by inspection or execution, as it is created or revised. When the asset is revised, the previous testing materials are revised and reapplied to the new version of the asset.

   For a code asset, the unit test asset is coded as a DotUnit test class.
2. The initial generic game test set is revised for each new game. In addition a game specific test set is created.

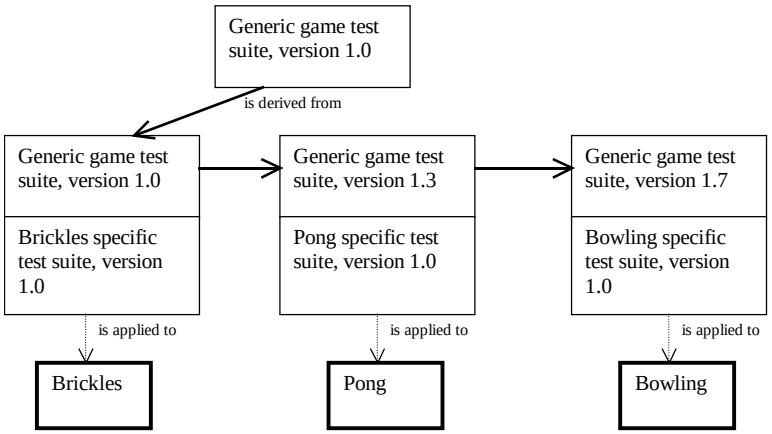

Figure 2 - Test cases related to products

3. The system test cases are maintained as text documents and are applied by hand.

5 Tailoring production plan to product-specific production plan

The production plan is, in general, very generic. The information in the plan applies to all products built using the current asset base. There are, however, some sections of the plan that must be modified for a specific product. Here we are giving a guide to product teams using this plan on how to modify it for their specific product.

The two most important and obvious sections that must be modified are the schedule and the bill of materials. Since we are using a manual product production approach the schedule defines which personnel are needed and when. The bill of materials will provide a useful means of tracking the use of core assets.

6 Management information

6.1 Schedule

In the product specific production plan this section contains the detailed schedule based on the process defined in section 4. Here we provide a template schedule. In the product-specific production plan this template is completed. The personnel assignments are made to specific people and the time estimates are changed based on any special circumstances for the product.

Table 1 – Schedule template

Process step	Product specific	Who	When
Product Identification		Product planning	Done
Product Definition		analyst	≅ .5 day
Product Analysis	Analyze the rules of the new game	analyst	≅ .5 day
Product Design	Identify new elements needed by game	designer	≅ 2 hours
	Identify changes to existing elements	designer	≅ .5 day
	Create product-specific implementation of the Game interface	designer	≅ .25 day
	Create product-specific implementation of EventHandlerDefinitions	designer	≅ .25 day
Product Build	Create new .Net projects for libraries and application	developer	≅ .5 hour
	Create new make file	Automated/developer	≅ .5 hour
Product Test	Create unit tests for new elements	developer	≅ .5 day
	Modify existing unit tests for existing elements	developer	≅ .5 day
	Execute unit tests	developer	≅ .5 day
	Modify/extend product test suite	tester	≅ 1 day
	Execute product tests	tester	≅ .5 day

6.2 Production Resources

The primary resources needed for product production are the Visual Studio .Net environment and the UML modeling tool being used by the organization.

During the analysis and design steps in the product production process the UML model is extended to include any new elements that must be defined.

The .Net environment is used to create any new components that are required. The environment is then used to build an executable.

6.3 Bill of materials (BOM)

Here we provide a template BOM. For each product-specific production plan fill in the template with exactly what will be used.

This game requires the following list of code assets:

Component	Source	Cost
product-specific Game component	In-house	
the generic GameBoard	In-house	
required Sprites (add specifics here)	In-house	
Implementation of EventHandlerDefinitions interface	In-house	

These are all in-house developed components so the BOM has a total cost of zero.

6.4 Product-specific details

The rules of the game, which are the most unique parts of the product, are distributed across the Sprites, the EventHandlerDefinitions, and the Game component. Most of the rules enforced by Sprites are common across most, if not all, of the games. The EventHandlerDefinitions component is developed specifically for a game as is the Game component.

The single most important product-specific detail is the animation loop in the Game class. This defines the sequence of events that occur in the Game.

6.5 Metrics

Two metrics will be important to the product production process: number of new lines of code and the number of unique lines of code.

6.5.1 New lines of code

This metric will describe the number of lines of new code that has to be written for this product. This code may or may not be used in another product later. A higher value for this metric indicates more effort required to produce the product and impacts cost and schedule.

6.5.2 Unique lines of code

This metric will describe the percentage of a product that is code not used in any other product. This metric will change over time as code assumed to be unique gets reused or as code assumed to be reusable is not reused after some time or product horizon. A higher value of this metric indicates less similarity between products and indicates a longer pay back time.

7 Attached Processes[2]

7.1 Constructing the Production Plan

[Chastek 02] describes how to build a production plan. We will not repeat that information here. However, in Figure 3 we show a very high-level version of the process in [Chastek 02].

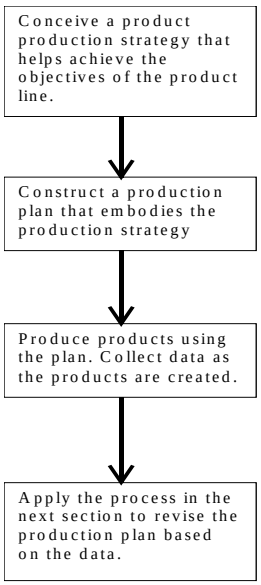

Figure 3 - High-level planning process

[2] This section is the "attached process" described in [Clements 02]. For the product line production plan, this process defines how the production plan is built initially. The process focuses mainly on modifying the production plan of the product line.

7.2 Changing the Production Plan

The production plan is reviewed as experience grows and as the asset base is revised. Data is collected at the delivery of each product. The AGM product line organization will review the generic production plan at the end of each increment using the data collected. The review is scheduled after the revision of the core asset base is completed for the next increment. The member of the core asset team that owns the plan initiates the review.

The review is intended to identify any inconsistencies between the newly revised set of assets and the production plan. Prior to the review, the architecture documentation and any information in the plan about the assets should be updated to reflect the new asset base.

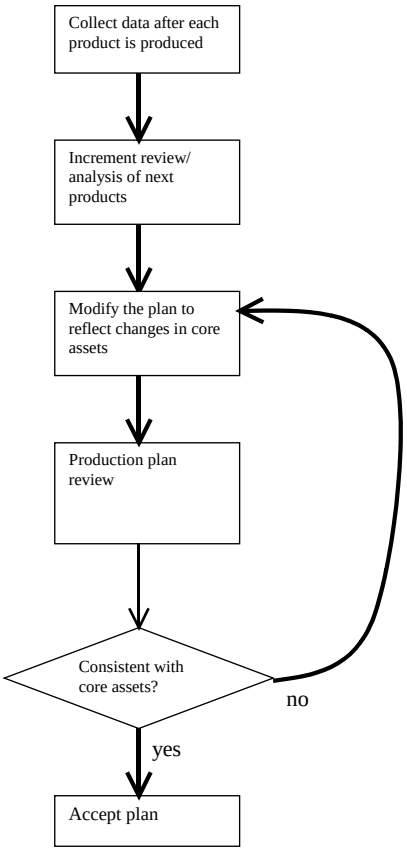

Figure 4 – Process to change the production plan

Appendix D

Management Information from Brickles Production Plan

This appendix includes a copy of the management information provided in the Brickles production plan from the AGM Product Line [35]. The document is discussed in the Case Study in chapter 3.

6 Management information

6.1 Schedule

In the product specific production plan this section contains the detailed schedule based on the process defined in section 4. Here we provide time estimates for most tasks. In the product-specific production plan this template is completed. The personnel assignments are made to specific people and the time estimates are changed based on any special circumstances for the product.

Table 1 – Time Estimates for producing the Brickles product

Process step	Product specific	Who	When
Product Identification		Product planning	Done
Product Definition		analyst	? .5 day
Product Analysis	Analyze the rules of the new game	analyst	? .5 day
Product Design	Identify new elements needed by game	designer	? 2 hours
	Identify changes to existing elements	designer	? .5 day
	Create product-specific implementation of the Game interface	designer	? .25 day
	Create product-specific implementation of EventHandlerDefinitions	designer	? .25 day
Product Build	Create new .Net projects for libraries and application	developer	? .5 hour
	Create new make file	Automated/developer	? .5 hour
Product Test	Create unit tests for new elements	developer	? .5 day
	Modify existing unit tests for existing elements	developer	? .5 day
	Execute unit tests	developer	? .5 day
	Modify/extend product test suite	tester	? 1 day
	Execute product tests	tester	? .5 day

6.2 Production Resources

The primary resources needed for product production are the Visual Studio .Net environment and the UML modeling tool being used by the organization.

During the analysis and design steps in the product production process the UML model is extended to include any new elements that must be defined.

The Brickles requirements document includes only those use cases that relate to the Brickles game.

The .Net environment is used to create any new components that are required. The environment is then used to build an executable.

A .Net solution has been created for the Brickles game.

6.3 Bill of materials (BOM)

Here we provide a template BOM. For each product-specific production plan fill in the template with exactly what will be used.

This game requires the following list of code assets:

Component	Source	Cost
product-specific Game component	In-house	30 lines of new code[1]
the generic GameBoard	In-house	40 lines of new code[1]
required Sprites	In-house	20 lines of new code[1]
MovableSprite		15 lines of new code[1]
StationarySprite		0 lines of new code[1]
Puck		5 lines of new code[1]
Paddle		5 lines of new code[1]
Brick		5 lines of new code[1]
Left Wall		5 lines of new code[1]
Right Wall		5 lines of new code[1]
Floor		5 lines of new code[1]
Ceiling		5 lines of new code[1]
BrickPile		25 lines of new code[1]
Implementation of EventHandlerDefinitions interface	In-house	5 lines of new code[1]
BricklesEventHandlerDefinitions		15 lines of new code[1]

[1] Since this is an example, only relative costs are given instead of actual costs.

The costs of assets are documented at the product for which they are first used. The profitability of the product is computed differently and amortizes the costs of assets across all of the products that use the asset.

6.4 Product-specific details

The rules of the game, which are the most unique parts of the product, are distributed across the Sprites, the EventHandlerDefinitions, and the Game component. Most of the rules enforced by Sprites are common across most, if not all, of the games. The EventHandlerDefinitions component is developed specifically for a game as is the Game component.

The single most important product-specific detail is the animation loop in the Game class. This defines the sequence of events that occur in the Game.

For Brickles the Game class includes catchers for the exceptions that signal the two different endings to the game.

6.5 Metrics

Two metrics will be important to the product production process: number of new lines of code and the number of unique lines of code.

6.5.1 New lines of code

This metric will describe the number of lines of new code that has to be written for this product. This code may or may not be used in another product later. A higher value for this metric indicates more effort required to produce the product and impacts cost and schedule.

Since Brickles is the first product in the product line technically all the lines of code are new.

6.5.2 Unique lines of code

This metric will describe the percentage of a product that is code not used in any other product. This metric will change over time as code assumed to be unique gets reused or as code assumed to be reusable is not reused after some time or product horizon. A higher value of this metric indicates less similarity between products and indicates a longer pay back time.

The Game module and the event handling definitions, 45 lines of code, are unique to this product.

Appendix E

Example Use Case from AGM Requirements

This appendix includes a copy of an example use case provided in the the Requirements document from the AGM Product Line [35]. The document is discussed in the Case Study in chapter 3.

Appendix A – Use Cases

Play the Game

Use Case ID: AGM001

Use Case Level: **abstract**

Scenario

Actor: GamePlayer or GameInstaller

Pre-Conditions: AGM011 has completed successfully

Detailed Description

Trigger: Actor selects game executable and initiates execution

Actor	System responds by
Selects **PLAY** from the menu	Initializes the game and displays the gameboard
Left mouse click (or equivalent) to begin play	Starting game action
Uses left mouse button (or equivalent) or keyboard to enter commands	Responds to the command in the expected manner
Responds to Won/Lost/Tied dialog with left mouse click (or equivalent)	Returns the gameboard to its initialized, ready to play state

Post-conditions: Actor has Won/Lost/Tied and the game is ready to play again

Alternative Courses of Action:

Actor	System responds by
At any time the actor may select **`EXIT`** from the menu	See use case AGM002

Extensions:

Actor	System responds by
See use case AGM006	
See use case AGM007	
See use case AGM008	

Exceptions:

Actor	System responds by

Concurrent Uses:

Related Use Cases: Exit the game

External Supporting Information

Requirement Originator: domain analyst

Rationale For Requirement: This is the main purpose of the product

Additional Relevant Requirements:

Decision Support

Frequency: on demand

Criticality: high

Risk: low

Modification History

Use Case Recorder: John D. McGregor

Initiation Date: Friday, June 13, 2003, at 8:11 AM

Last Modified: Wednesday, June 2, 2004, at 9:25 AM by Systems Staff

Exit the Game

Use Case ID: AGM002

Use Case Level: **System End-to-End**

Scenario

Actor: GamePlayer or GameInstaller

Appendix F

Glossary

The research areas of software product line engineering and aspect-oriented software development have their own lingo and terminology. This glossary provides definitions of the main concepts and terms used in this document, for which there is general agreement.

F.1 AOSD

Within the European network of excellence AOSD-Europe an ontology has been created for AOSD terminology to enable effective communication and integration of activities within the network. The definitions in this glossary are taken from [6].

Advice An *advice* is an aspect element, which augments or constrains other concerns at join points matched by an pointcut expression [6].

AOSD *Aspect-Oriented Software Development (AOSD)* is a set of emerging technologies that seeks new modularisations of software systems. AOSD allows multiple *concerns* to be separately expressed but nevertheless be automatically unified into working systems. [20].

Aspect An *aspect* is a unit for modularizing an otherwise crosscutting concern [6].

Composition *Composition* is the integration of multiple modular artefacts into a coherent whole [6].

Concerns A *concern* is an interest, which pertains to the systems's development, its operation or any other matters that are critical or otherwise important to one or more stakeholders [6].

Crosscutting *Crosscutting* is the scattering and/or tangling of concerns arising from the inability of the selected decomposition to modularize them effectively [6].

Crosscutting Concern A *crosscutting concern* is a concern, which cannot be modularly represented within the selected decomposition. Consequently, the elements

of crosscutting concerns are scattered and tangled within elements of other concerns [6].

Decomposition *Decomposition* is the breaking down of a larger problem into a set of smaller problems which may be tackled individually [6].

Early Aspects *Early Aspects* are crosscutting concerns that are identified in the early life cycle phases of a software system's development. These phases include the requirements analysis, domain analysis and architecture design phases. Early aspects cannot be localised and tend to be scattered over multiple early life cycle artifacts or artifact elements (such as sections in a requirements document, or modules in an architectural design) [3].

Join Point A *join point* is a point of interest in some artefact in the software lifecycle through which two or more concerns may be composed [6].

Join Point Model A *join point model* defines the kinds of join points available and how they are accessed and used [6].

Pointcut A *pointcut* is a predicate that matches join points. More precisely, a pointcut is a relationship from JoinPoint -¿ boolean, where the domain of the relationship is all possible join points [6].

Scattering *Scattering* is the occurrence of elements that belong to one concern in modules encapsulating other concerns [6].

Separation of Concerns *Separation of Concerns* is an in depth study and realization of concerns in isolation for the sake of their own consistency (adapted from "On the Role of Scientific Thought" by Dijkstra, EWD 447) [6].

Tangling *Tangling* is the occurrence of multiple concerns mixed together in one module [6].

Tyranny of Dominant Decomposition The *Tyranny of Dominant Decomposition* refers to restrictions (or tyranny) imposed by the selected decomposition technique (i.e. the dominant decomposition) on software engineer's ability to modularly represent particular concerns [6].

Weaving *Weaving*: Historically this term is used to refer to composition of aspects with other concerns in the system. See composition. [6].

F.2 SPLE

Concept Model A *concept model* describes the concepts in a domain in some appropriate modeling formalism and informal text [15].

Core Assets *Core assets* often include, but are not limited to, the architecture, reusable software components, domain models, requirements statements, documentation and specifications, performance models, schedules, budgets, test plans, test cases, work plans, and process descriptions. The architecture is key among the collection of core assets [14][41].

Development *Development* is a generic term used to describe how core assets (and products) come to fruition. Software enters an organization in one of three ways: the organization builds it (from scratch or by mining legacy software), purchases it (largely unchanged, off the shelf), or commissions it (contracts with someone else to develop it especially for them). So the term *development* might actually involve building, acquiring, purchasing, retrofitting earlier work, or any combination of these options [41].

Domain *Domain*: An area of knowledge scoped to maximize the satisfaction of the requirements of its stakeholders, includes a set of concepts and terminology understood by practitioners in that area, and includes the knowledge of how to build software systems (or parts of software systems) in that area [15].

Domain Analysis *Domain analysis* can be defined as the process of identifying, capturing and organising domain knowledge about the problem domain with the purpose of making it reusable when creating new systems. The result of domain analysis is a *domain model* which can be reused to implement various applications [4].

Domain Definition The *domain definition* defines the scope of a domain and characterizes its contents by giving examples of existing systems in the domain, counterexamples, and generic rules of inclusion and exclusion [15].

Domain Engineering *Domain engineering* is the activity of collecting, organizing, and storing past experience in building systems or parts of systems in a particular domain in the form of reusable assets (i.e., reusable work products), as well as providing an adequate means for reusing these assets (i.e., retrieval, qualification, dissemination, adaptation, and so on) when building new systems [15].

Domain Lexicon A *domain lexicon* defines the domain vocabulary [15].

Domain Model A *domain model* is an explicit representation of the *common* and *variable* properties of the systems in a domain, the semantics of the properties and domain concepts, and the *dependencies* between the variable properties. In general, a domain model consists of a *domain definition*, *domain lexicon*, *concepts models* and *feature models* [15].

Domain Scoping *Domain scoping* identifies the domains of interest, the stakeholders, and their goals, and defines the scope of the domain [15].

Feature A *feature* is a property of a domain concept, which is relevant to some domain stakeholder and is used to discriminate between concept instance. In the context of domain engineering, features represent reusable, configurable requirements [15].

Feature Model A *feature model* defines a set of reusable and configurable requirements for specifying the systems in a domain. A feature model prescribes which feature combinations are meaningful, which of them are preferred under which conditions and why [15].

Software Architecture A *software architecture* is a description of the subsystems and components of a software system and the relationships between them. Subsystems and components are typically specified in different views to show the relevant functional and nonfunctional properties of a software system. The software architecture of a system is an artifact. It is a result of the software development activity [10].

Software Product Line A *software product line* is a set of software-intensive systems sharing 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 [15].