A Tool-Based Approach to Managing Crosscutting Feature Implementations

Iris Groher
SEA Institute Johannes Kepler University Linz
iris.groher@students.jku.at

Charles W. Krueger
BigLever Software
ckrueger@biglever.com

Christa Schwanninger
Siemens AG CT SE 2
christa.schwanninger@siemens.com

Abstract

Software product line engineering aims to reduce development time, effort, cost and complexity by taking advantage of the commonality within a portfolio of similar products. The effectiveness of a software product line approach directly depends on how well feature variability within the portfolio is managed throughout the development lifecycle, from early analysis through maintenance and evolution. Variability of features in a product line often has widespread impact, crosscutting not only multiple parts of individual artifacts but also multiple artifacts in multiple stages of the development lifecycle. This paper presents a tool-based approach for managing crosscutting feature variability in software product lines using aspect-oriented principles. The approach makes it possible to handle multi-artifact crosscutting. For code artifacts is is even independent of the languages used for feature implementation. We report on experiences made in industrial settings, including 2006 Software Product Line Hall of Fame inductee, LSI Logic's Engenio Storage Group.

Categories and Subject Descriptors D.2.13 [*Reusable Software*]: Domain engineering, Reusable libraries, Reuse models

General Terms Design, Economics, Management, Measurement

Keywords Software Product Line Development, Crosscutting Feature Variability, Tool-Based Approach

1. Introduction

Most high technology companies are specialized for a specific market and the products in their portfolios have many parts in common. An increasing number of these companies realize that software product line engineering (SPLE) [11] [30] fosters reuse at all stages of the development lifecycle, thus shortens development time and helps staying competitive. Products usually differ by the set of features they include in order to fulfill customer requirements. A feature is defined as an increment in program functionality provided by one or more members of the product line [18].

The effectiveness of a software product line approach directly depends on how well feature variability within the portfolio is managed throughout the development lifecycle. Commonalities and the flexibility to adapt to different stakeholder requirements are captured in a product line's core assets. Those reusable assets are created during domain engineering. During application engineering, products are either automatically or manually assembled using the assets created during domain engineering.

Variability management is the activity concerned with identifying, designing, implementing, and tracing flexibility in SPLE. Variability of features in a product line often has widespread impact, crosscutting not only multiple parts of individual artifacts but also crosscutting multiple artifacts in multiple stages of the development lifecycle. For example, variation in persistence can have a wide ranging impact on different kinds of artifacts such as configuration scripts, SQL, documentation, and tests. Moreover, crosscutting feature variability not only results in complex feature implementations. Instantiating assets by binding the variability to concrete values in application engineering is complex too when variability crosscuts artifacts. Further, tracing of variability, i.e. how a given requirement results in a certain software configuration, is a very difficult task when variability is not sufficiently modularized. To effectively improve the management and tracing of crosscutting feature variability, advanced modularization techniques need to be considered.

Aspect-oriented software development (AOSD) [20][14] improves the way software is modularized by providing means for modularizing crosscutting concerns. Crosscutting concerns are encapsulated as aspects and powerful mechanisms support their composition with other software artifacts. AOSD has the potential to better deal with crosscutting feature implementations as variability can be localized in aspects and later bound using the composition mechanisms AOSD provides. Currently, AOSD subsumes programming languages, design notations and support for analyzing requirements documents for capturing crosscutting requirements. However, there is no integrated approach for all life cycle stages nor dedicated support for capturing variability in the context of product line engineering yet.

This paper presents a tool-based approach for managing and tracing crosscutting feature variability in SPLE using AO principles. The tool provides an infrastructure and a development environment for SPLE and is independent of the kinds of artifacts or the languages used for implementing the software assets. It offers a special abstraction called *mixin* that serves as a means to deal with variability that is spread across multiple modules. At product instantiation time, the tool automatically composes the encapsulated mixin features with the assets they crosscut. Features can thus be shared across multiple modules which allows for effective implementation and binding of variability. This in turn improves traceability and supports evolution.

We report on practical experience made with our approach in industrial settings. The first example is a home automation system

AOSD '08 March 31 – April 4 2008, Brussels, Belgium.

product line, called Smart Home. The second example is a product line that is successfully operated by Engenio and is 2006 Software Product Line Hall of Fame inductee. This product line contains over 100 crosscutting feature implementations that were effectively managed using Gears.

The remainder of the paper is organized as follows: Section 2 motivates our approach. Section 3 describes our tool-based approach to handling crosscutting feature implementations. Section 4 demonstrates the benefits of our approach in industrial settings. We evaluate our approach in Section 5. Related work is discussed in Section 6. Section 7 summarizes the paper and provides an outlook on future work.

2. Motivation

Variability of features often has widespread impact on multiple artifacts in multiple lifecycle stages, making it a predominant engineering challenge in SPLE. To allow for effective management and traceability of features, their modularization is vitally important. Practice has shown that a good modularization of features is a very hard thing to do. Many not only tend to be crosscutting with respect to one specific kind of artifact (e.g. code), they are often spread across different kinds of artifacts (requirements documents, build scripts, tests etc.).

When crosscutting features cannot be satisfyingly modularized in the various kinds of artifacts, the following problems arise:

- **Difficult tracing of variability.** In SPLE it is essential to trace variations across the different development phases. Traceability is necessary to e.g. be able to select the necessary variants of code base assets for a new product from the requirements, to select the right test cases or pieces of documentation for this product or to find out why a variation point exists at all. Good traceability of variants is one of the major challenges in current software product line development. Variations that crosscut one or more artifacts make traceability even harder, reducing the cost benefits associated with the mass customization value of product lines.
- **Complex instantiation of products.** During application engineering, products can be derived in a systematic way when the relationships between requirements, design, and implementation artifacts are known. If the locales of variability and their impact on the different product line artifacts are not known, product instantiation gets very complex.
- **Problematic evolution of product lines.** The number of variation points increases during product line evolution. New requirements incorporated into the product line introduce new variation points and existing variation points might become obsolete. If crosscutting variation is not well modularized, a system becomes overly complex and unnecessary variation will never be reduced.

AOSD can be taken as a means to effectively improve the management of crosscutting feature implementations. It provides means to encapsulate crosscutting concerns as aspects and to compose them with other software artifacts.

Single artifacts crosscutting can be handled using one of the many existing AO approaches. Implementation level crosscutting can be handled with many existing AO programming languages, such as AspectJ [19] or CaesarJ [4]. AO techniques are not only applicable at code level. Extensive research is conducted in AO requirements engineering [31] [32] [33] as well as architectural modeling and design [29] [5]. AO requirements engineering techniques help to identify crosscutting concerns already at the very early stages of the development lifecycle and AO architecture and design techniques provide means to separate concerns at the modeling level.

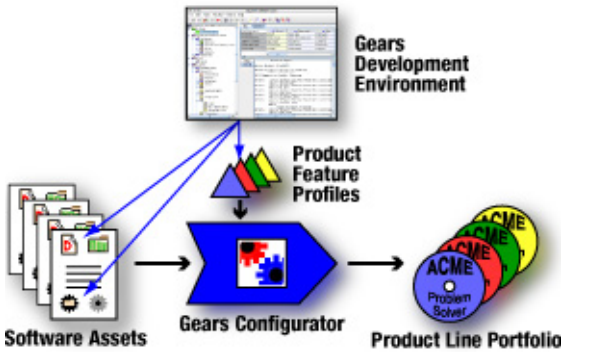

Figure 1. A Gears Software Product Line

There is however a need for uniform handling of variations that crosscut multiple artifacts in multiple lifecycle stages. To the best of our knowledge there is no approach currently available that supports separation and later composition of concerns that crosscut multiple types of artifacts. Especially in SPLE, tools are needed that allow for this advanced separation of multi-artifact crosscutting. This is vitally important as both the cost and time-to-market value propositions of product lines are based on the assumption of complete traceability.

The next section introduces Gears, a tool-based approach to managing crosscutting feature implementations that span over several artifacts in multiple lifecycle stages.

3. A Tool-Based Approach to Managing Crosscutting Feature Implementations

3.1 Variability Management with Gears

Gears is a commercial software product line development tool - an engineering tool designed to facilitate the development of software for a portfolio of similar products with variations in features and functions [9] [22]. In simplest terms, the tool is a software product line *configurator*. As illustrated in Figure 1, Gears instantiates software products that are members of a product line portfolio by automatically composing and configuring shared software assets. This automated assembly is based on formal descriptions called *feature profiles* that model each product in the portfolio in terms of optional and varying feature choices for the product. Gears provides a development environment that contains languages, infrastructure, and editors for creating the variation points in the software assets and the feature profiles. Manufacturers have long used analogous engineering approaches to produce a product line portfolio, using a factory that assembles and configures parts that are specially designed to be reused across the product line.

Gears is intended to work with new or legacy assets, with no or minimal changes to existing assets. Because of this requirement, the variation management mechanisms are independent of any programming language or asset type. The tool works at the granularity of files of any type. Gears works equally well with files that contain for example source code, documentation, requirements, or test cases.

A feature modeling language is used to model the optional and varying features - at the domain level of abstraction - that differentiate the products in a portfolio and to model the unique feature profile for each of the products in the portfolio. The Gears feature modeling language looks very much like parameter declarations in conventional programming languages. Feature declarations have

types, such as Boolean or Integer for modeling simple features, or Set and Enumeration for modeling composite features. In the following example, `Locator` is an optional inventory locator feature of an online shop that is modeled with a Boolean type. `Brand`, a collection of different custom brands that can be individually selected to be included in products, is modeled as an Enumeration type. One and only one of the Atoms can be selected for any particular product instance.

```
Boolean Locator;
Enumeration Brand {
   Atom Acme;
   Atom Uptown;
   Atom OpticsInc;
}
```

To create a feature profile for a product instance, values are assigned to features. Continuing with the above example, a product with a `Locator` and an `OpticsInc` brand would have the following feature profile:

```
Locator = true;
Brand = { OpticsInc };
```

Variation points are used in software assets to manage implementation level differences among the products in the portfolio. Variation points, which are used in software assets such as requirements, source code, test cases, and documentation, enable a single software asset to be shared among all products in a portfolio.

Feature variability in a software product line can have widely different granularities of impact at the implementation level, ranging from a single text token to a complete subdirectory hierarchy. To accommodate this, Gears provides variation points at three different granularities inside a software asset: text patterns, files, and directories. Note that because Gears is language independent, the variation points are independent of any programming language or asset representation.

A Gears variation point encapsulates variations that exist for a file or directory along with the variation point *logic*, written in a special-purpose programming language that maps values from a feature profile to an instance of the file or directory variation point.

For example, consider a product line with a graphic icon in the user interface that is used for customer-specific branding of product instances. This file must contain a different customer-specific icon each time the Gears configurator is run to produce a customer-specific product. To accomplish this, all of the possible icon files are encapsulated in a variation point along with the logic for selecting which icon file to use when a product is actuated.

Figure 2 illustrates the key concepts of a variation point using the above mentioned example. The file `Brand_Icon.gif` is a *projection file*, which is the file that is instantiated by the variation point and used in a customer-specific product. The three files, `Acme.gif`, `Up.gif`, and `Optics.gif`, are *variant files*. These are the different icon files encapsulated in the variation point. One of these variant files is selected to provide the content of the projection file `Brand_Icon.gif` when the variation point is instantiated. The logic description, which is also encapsulated in the variation point, provides the mapping from the feature declaration parameters to the different realizations of `Brand_Icon.gif`. The first clause in the logic description says to select the variant file `Acme.gif` when the feature declaration parameter `Brand` has been assigned the value of `Acme` in a product definition. Recall from the example introduced above that Brand is declared as an Enumeration type with Acme, Uptown, and OpticsInc as possible values.

The Gears configurator (also called the *actuator*) takes a feature profile for a product and instantiates that product by composing software assets and by instantiating variation points within those

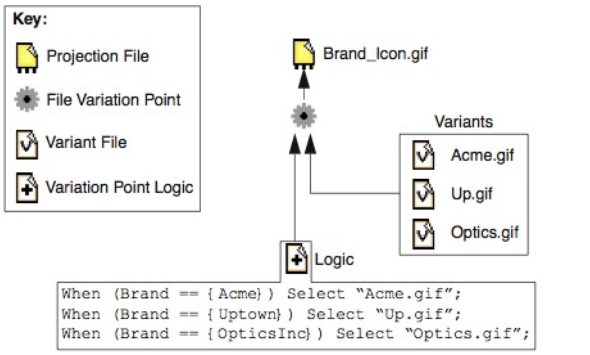

Figure 2. A Gears Variation Point

assets. Developers can automatically compose and configure all of the assets (e.g. requirements, source code, test cases, and documentation) for any product at any time, based on the current state of the assets under development. For example, a developer's change to source code and test cases can immediately and automatically be actuated, built, and tested in all products in the portfolio.

Although many feature declarations will fall cleanly into one module or another, there may be cases where a feature declaration is needed in two or more modules. Rather than duplicating the same feature declaration in multiple modules, Gears provides the mixin abstraction which allows to create a feature declaration in one place and then "mix it in" to the feature declarations of multiple modules. The next section explains the mixin concept in detail and outlines the similarities and differences to AO constructs.

3.2 Mixins

The crosscutting nature of feature variability in software product lines has led to Gears constructs that are analogous to AO constructs. Mixins are modular encapsulations for feature declarations and product definitions that can be shared in multiple modules. For example, assume that the company brand of the example introduced in the previous section is a common source of variation in multiple modules. The brand could be added as a feature declaration in each of these modules, but this duplication leads to possible inconsistencies. One could accidently actuate one module with one company brand and another module with a different company brand, resulting in inconsistent branding. The Gears mixin allows to declare a common feature declaration in one place and then import this declaration for use in multiple modules.

The Gears logic clause operation consists of a Select statement (see Figure 2) followed by an optional list of *pattern statements*. Pattern statements are only valid for file variation points and specifically for file variation points that contain text files. They provide zero or more transformation steps that apply pattern-based text substitutions to the projection file being realized from a Select statement. Transformations are applied to the projection file, where a match pattern is replaced by one or more substitution patterns. Patterns in Gears are based on and serve the same purpose as Java Patterns, which themselves are based on the regular expressions and substitutions of Perl. Thus, the match pattern is a regular expression that can match text in the projection file and the substitution patterns are the replacement for the matched text.

Match patterns and substitution patterns can either be defined inline as literal strings in the pattern statements, or they can be defined in a separate Patterns definition document and referred to by name in the pattern statements. Inline definitions are appropriate for match and substitution patterns that are used once. Definitions in separate Patterns definition documents are appropriate for match

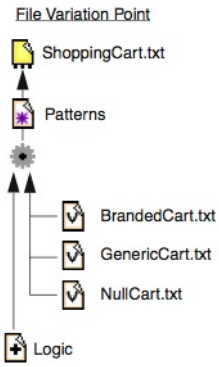

Figure 3. A Gears Variation Point with Patterns

and substitution patterns that are defined in one place (i.e., in the Patterns document) and then used in multiple pattern statements in one or more variation points.

A file variation point with an Apply statement is illustrated in Figure 3. It is identical to the file variation point with a single Select statement, except that patterns are applied during the realization of the projection file, `ShoppingCart.txt`.

There are different pattern statements. `Apply` replaces the matched string with the substitution pattern. `Append` concatenates the substitution pattern onto the end of the matched string. `AppendLine` concatenates the substitution pattern plus a newline.

In the following example, literal match and substitution patterns would replace all occurrences of the string "PageTitle" with the string "AcmeInc Online Store".

```
Apply "PageTitle" -> "AcmeInc Online Store";
```

Here, all occurrences of strings ending with "Title" would be replaced with the string "OpticsInc Online Store".

```
Apply ".*Title" -> "OpticsInc Online Store";
```

It is also possible to pass parameters into substitution expressions. Substitution expressions which accept parameters are called *parameterized substitution patterns*. The parameters will be expanded into the substitutions text using parameter value substitutions – the predefined variables #0#..#N-1#, where N is the number of parameters passed into the substitution. The parameter values are literal strings that appear between parenthesis following the substitution reference. In the following example, the `SetMaxItems` substitution takes a single parameter that sets the maximum number of items in a shopping cart:

```
MaxItems="_MaxItems_"
-> SetMaxItems = "SetMaxItems(@#0#@);"
```

An example pattern statement in a Logic file which passes a parameter value into the substitution is:

```
Apply MaxItems -> SetMaxItems("20");
```

During actuation, the text "`_MaxItems_`" would be replaced with "`SetMaxItems(20);`". It is an error if the number of parameter values passed into a substitution does not match the number of parameters expected by the defined substitution pattern. Any variables within a parameter value string will be expanded before they are passed into the substitution.

The big difference between classic AO and Gears is, that Gears targets all artifact types with the same approach, not only code or design or requirements. In the following paragraphs, further similarities and differences between Gears mixins and AO concepts are outlined. Practical examples are contained in the section on industrial product line applications.

Aspects and Mixins. Crosscutting feature implementations are modularized in Gears, analogous to the concept of aspects in traditional AO. They are encapsulated into Mixins and woven into the assets they crosscut.

Join Points and Variation Points. Gears variation points are locations of variation in software assets where crosscutting feature implementations can influence asset implementations. This is analogous to the concept of join points in AO, where crosscutting concerns can be composed with asset implementations. In Gears, variation points are explicitly identified as file variation points or directory variation points. This is needed to accommodate the diversity of variations that a single feature can cause in different locations and in different lifecycle stages. Within an artifact, any point can be a join point, while in AO join points are typically a deliberately chosen small set of points in the dynamic control flow of an application.

Pointcuts and Match Patterns. For cases where crosscutting software product line features do not have uniform variations across assets, Gears provides match patterns to specify the locales of text-level variations within a variation point. As described in the next paragraph, substitution patterns are used to weave in feature-specific behaviors at or around the locales of match patterns. Match patterns in Gears are based on the regular expressions and substitutions of Perl. A match pattern is a regular expression that can match text in a projection file and the substitution patterns are the replacements for the matched text. Match pattern can be compared to pointcut definitions in traditional AO. Different to mixins, AO pointcuts are predicates over dynamic join points, not pure text matching, which makes AO very powerful. Another important difference is that Gears does not perform syntactic or semantic checks. Text pattern matching only is very powerful but also error prone. AO's restriction on meaningful join points helps to prevent errors and understand the impact of aspects on other software artifacts.

Advice, Logic, and Substitution Patterns. The logic inside a Gears variation point provides the mapping from crosscutting features to implementation-level advice by first selecting a file variant and then applying one or more patterns to produce the final projection file. This is analogous to weaving of advices at join points in traditional AO. The selection of the file variant and the application of each pattern can be controlled by arbitrarily complex logic expressions involving one or more features and feature interactions. In AO, pointcuts may expose runtime information to advices, whereas mixins only provide text substitution at the matched points.

Scoping of Crosscutting Concerns. This is managed by Gears modularity constructs (modules, mixins, imports). With this mechanism, features can be scoped to target a subpart of the potential join points only. Features encapsulated into a Mixin have to be imported into the modules they crosscut. With this approach it gets less likely that unintended join points match. In traditional AO it would be necessary to explicitly change the pointcut of an aspect to achieve the same or to have several aspects with different pointcuts and switch them on or off as needed.

4. Industrial Product Line Applications

4.1 Smart Home

The first industrial example we use to illustrate our approach is a home automation system product line (see also [30]), called Smart Home. It is a demonstrator based on real system requirements from Siemens AG. In homes you typically find a wide range of electrical

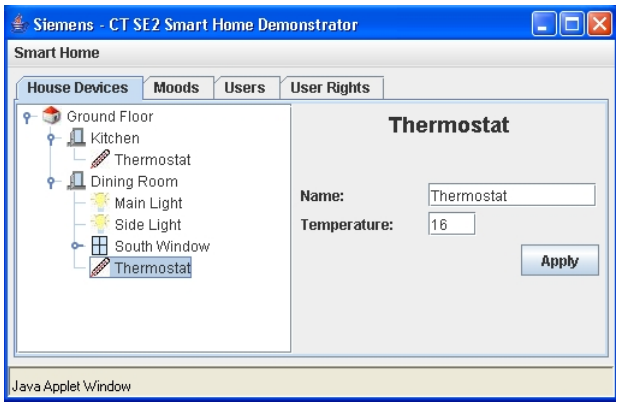

Figure 4. Smart Home User Interface

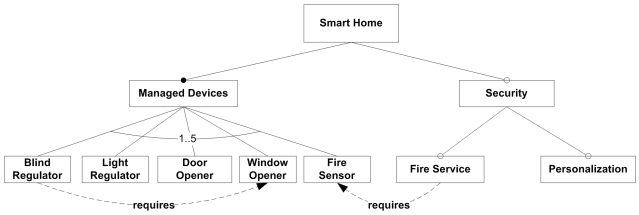

Figure 5. Smart Home Feature Model

and electronic devices such as lights, thermostats, electric blinds, fire and smoke detection sensors, entertainment equipment, and communication devices. Smart Home networks those devices and enables inhabitants of a home to monitor and control the status of devices from a common user interface. The home network also allows the devices to coordinate their behavior in order to fulfill complex tasks without human intervention.

Sensors are components that measure physical values of their environment and make them available to Smart Home. Controllers activate devices whose state can be monitored and changed and actuators change physical properties of the home. All installed devices are part of the home network. The status of devices can either be changed by inhabitants operating on the user interface or by Smart Home using predefined policies. Policies let the system act autonomously in case of certain events. For example, in case of fire and smoke detection, windows get shut automatically and the fire brigade is called.

Figure 4 shows the user interface of Smart Home. On the left side, the house including its floors and rooms is displayed as a tree. Rooms contain different kinds of devices such as lights, windows, an thermostats. When a device is selected, its properties and current status shows up on the right side. Users can then monitor and control the status of devices according to their needs.

Varying types of houses, different customer demands, the need for short time to market, and saving of costs drive the need for a Smart Home product line and are the main causes of variability. Figure 5 shows a simplified version of the feature model of Smart Home. Filled circles indicate mandatory features, empty circles indicate optional features. For feature groups (indicated by an arc) a range is given that defines how many features of the group can be part of a valid configuration. Additional constraints, such as requires relationships between features, are represented by labeled arrows. This notation is similar to the one presented in [12].

Devices that can be managed by Smart Home include blind and light regulators, door and window openers, and fire sensors. Blind regulators require window openers. The security feature is optional. Personalization, an optional subfeature of security, enables inhabitants to define device settings that are automatically applied when the respective person is present in a room. Fire service automatically performs certain actions in case of fire detection and requires fire sensors to be included in the configuration.

The Smart Home product line is implemented in Java using OSGi [27] as service platform. A Smart Home installation consist of a specific arrangement of pre-built sensors and actuators (although a specific system can have custom devices). We therefore keep a library of software components that control certain types of hardware. Depending on the chosen hardware, the software components that are needed are included into the desired product.

The product line can handle arbitrary devices, as long as they provide the interfaces that are defined for each kind of actuator or sensor within Smart Home. The main part of the Smart Home software consists of controllers that implement the device orchestration to provide useful behavior.

To show the usefulness of capturing crosscutting features over multiple types of artifacts, we give an example that is simpler than in reality, but sufficient for showing the capabilities of Gears mixins. Imagine that the windows and doors are automatically shut and locked whenever a presence aware sensor detects that the last person left the house. This controller would be part of a security feature. Another controller automatically opens the windows in a room when somebody is at home and the outside temperature reaches a certain threshold. The windows and the blinds are shut when another higher threshold is met. This behavior is part of a room climate control feature.

In the Smart Home product line, the kind and number of devices can be customized, as well as the kind of controllers including some variation in their behavior, e.g. the concrete temperature that triggers the window opening and shutting. The inhabitant of the house does of course not care about the devices and the controllers, but only about their orchestration into features.

The optional fire service feature crosscuts other features. If the fire service feature is installed, the behavior of the security and the room climate feature is changed in case the fire sensors detect fire. The doors are either unlocked or their locking is prevented when the last person leaves the house, the doors nevertheless are shut automatically. The windows are all shut and kept shut, no matter what the climate control feature demands.

A feature profile with the fire service feature, both climate control and security included and a special kind of fire sensor selected could look like the following:

```
FireService = true;
KindofFireSensor = { LindyFireAndSmokeSensor };
ClimateControl = true;
Security = true;
```

What kinds of artifacts are in this case influenced (crosscut) by the optional fire service feature?

Source Code. If the fire service feature is installed, also fire or smoke detectors have to be installed. That means, that the software controlling those devices has to be included in the product. The logic for the security and room climate features is influenced by the fire service feature, i.e. the behavior of the controllers has to be changed. The state of the fire detectors has to be taken into consideration in addition to door, presence awareness, and temperature sensors. This additional code is woven into the code of the security and climate control controllers.

Build Files. The code for fire and smoke detectors and for the controllers that are involved in fire handling needs to be compiled and linked with the code of the other features selected for the Smart Home product. Figure 6 shows how patterns are applied to build

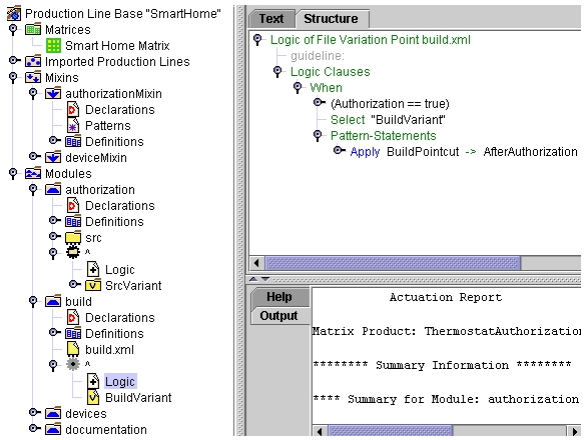

Figure 6. Variability in Build Files

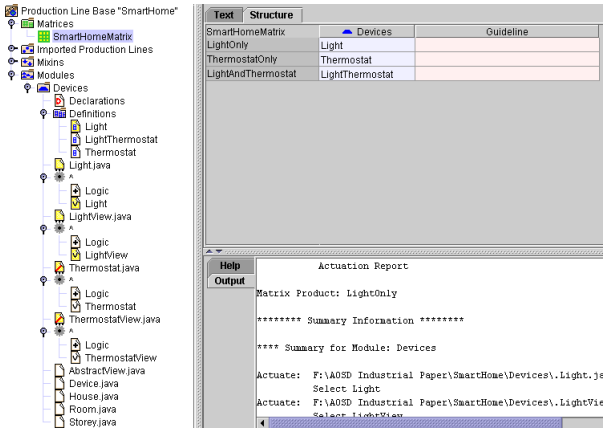

Figure 7. Gears Product Instantiation

files according to some feature selection. In this case, the build file is modified when the Authorization feature is present in the system.

Test Cases. The software for every house instance needs to be tested before installation. For every included feature new or changed test cases have to be added. For the fire service feature, the security and the climate controllers' test cases need to be enhanced with tests that, for example, send simulated fire sensor signals.

Design Models. Since we want to be open enough to extend the functionality of Smart Home with custom features any time, we also want to have a complete design document for every instance of a home automation installation. This design document helps the developers of the custom extensions to understand the software that has to be extended.

User Documentation. When the owner of a house orders the fire service feature, the documentation of the house automation should also contain the description of this feature in a way that is easy to understand. Therefore, the change of behavior of the climate control and the security feature in case the fire service is included should be integrated into the respective chapters on security and climate control.

In Gears, all parts of the optional fire service feature are encapsulated in a mixin. According to the feature profile, the parts are woven into the respective artifacts. The behavior of the fire service feature and its impact on other artifacts can therefore be completely modularized. This allows for tracing of the crosscutting fire service feature from requirements, to design, code, and tests. The problem of complex product instantiation is also solved, as Gears automatically composes the encapsulated feature with the rest of the system at the desired points at product instantiation time. Whenever the fire service feature needs to be changed according to new requirements, the changes are local to the mixin. Figure 7 shows how products are actuated in Gears. A matrix includes all feature profiles that have been defined for the portfolio. In order to create a product, one of the feature profiles (one row) has to be actuated.

4.2 LSI Logic's Engenio Storage Group

We report on experiences made with our approach in another industrial setting. It is a product line that is successfully operated by Engenio and is 2006 Software Product Line Hall of Fame inductee.

Engenio is in the business of providing feature-rich, high-performance storage servers to major OEM vendors such as IBM, SGI, Cray, StorageTek, and Teradata. Each OEM customer wants to take advantage of Engenio's core competence in storage technology but each one also wants unique and differentiated solutions. Thus, the need for efficient product line engineering is central to its business model. There are over 200,000 installations of Engenio's storage servers worldwide, worth over $9 billion US dollars. Engenio's Controller Firmware Development group is the focal point for their software product line. In recent years, this group has grown to 225 developers, working at four distinct geographic sites. The group currently provides firmware for approximately 110 products, with about one million lines of code of embedded software going into each product. Approximately 80% of the code is common among all products [17].

The Gears feature model for Engenio's product line contains over 100 crosscutting features - features that crosscut multiple of the more than 500 variation points in the source code assets. Crosscutting features are partitioned into two different Gears mixins, one for hardware related features, such as the variety of hardware platforms supported, and one for higher level software features. The feature model is proprietary since it represents a valuable part of the intellectual property for the product line. However, we can here illustrate the publicly visible crosscutting features corresponding to the hardware platform.

There are currently 6 major hardware platforms in the product line with minor variations among them. These are modeled with 7 different feature declarations in the Gears feature model. Because the platform choice has wide ranging implications across many parts of the firmware, the platform features are treated as crosscutting features and modeled in a Gears mixin. Four of the feature declarations crosscut assets within only a single subsystem, while the others crosscut - and are therefore imported in- 2 to 13 different subsystems.

There are approximately 228 different variations points that are crosscut by the platform features.

5. Evaluation

Gears is specifically designed to support variability management for product line engineering. The intent is not to support AO in general. For capturing aspects only in design or only in code, there are numerous languages and tools available in good quality. Variations that crosscut multiple artifacts are very hard to deal with as there is no approach currently available that supports separation and composition of concerns that crosscut multiple artifacts.

The Gears tool set has a clear advantage whenever a crosscutting concern spans several types of artifacts, e.g. code, documentation, and design documents. There is tooling available to connect crosscutting concerns over artifacts boundaries but these tools only make concerns visible [35]; concerns cannot be modularized and woven separately as done in Gears.

This ability to handle artifacts of different nature together with the light weight implementation of Gears results in a primitive join point model, namely text pattern matching. This can be tedious for bigger groups of join points that should be matched.

For the same reason, no syntactic or semantic checks are done during weaving. Errors will surface after weaving probably at compile time in code or even only at runtime. Post-processing design documents or build files may reveal errors for woven artifacts, documentation, however, cannot be checked automatically at all. While the latter cannot be improved with any AO language, the former two are handled well in existing AO programming languages and increasingly also in AO design tools.

For relatively stable product lines with not too many variations introduced constantly, the missing checks and the possibly complex pointcut specifications are no problem. In such cases, the products can be built and tested regularly. For constant massive addition of new crosscutting variations, tedious definition of pointcuts and missing syntactic and semantic checks can be an obstacle for introducing the tooling as only means to manage crosscutting variations in code and design. In such cases combining Gears with traditional AO tooling would be beneficial. This approach is currently implemented in a project conducted by the company HomeAway and will be published in a technical report [21].

The challenges identified in Section 2 are resolved in Gears as follows:

- **Tracing of variability.** As variability can be satisfyingly modularized, fine grained tracing is possible. Traces from requirements documents to design documents, implementation, and tests can be established as variability that crosscuts artifact boundaries is localized in one place. Gears also includes a *membership analysis tool* that provides an impact analysis report by deriving the conditions under which a file, directory, variation point, or variant is a member of an actuated product, the variation point hierarchy that determines its membership, and the set of product definitions of which it is a member. This further improves traceability.
- **Instantiation of products.** The Gears actuator produces custom product instances from the product line by activating the variation points. For each module actuated, Gears uses the values from a given product definition to evaluate the logic and instantiate the projection for every variation point in the module. Crosscutting feature implementations are localized in one place, in mixins, and automatically composed with the software assets at the appropriate locations at product instantiation time.
- **Evolution of product lines.** When existing variation points become obsolete during product line evolution, they can be removed easily as they are localized. The membership analysis tool of Gears can be used to analyze the impact that a variation point has on software assets and products which improves understanding and helps to reduce unnecessary variation. The introduction of new variation points is also improved when variability and its impact is well understood.

6. Related Work

Researchers started to reason about the appropriateness of AO technologies for software product lines already before 2000 and a considerable body of work emerged since then [1] [2] [16] [24] [25] [26] [3] [28]. When it comes to domain and application implementation, most of the existing work evaluates the appropriateness of aspect-oriented programming for PLE using programming languages such as AspectJ [19], CaesarJ [26], and ArchJava [28]. The typical approach is to implement crosscutting features in the AO extension to an object-oriented language during domain engineering. During application engineering the aspects or aspect variants which are needed in the respective products are integrated into the final product using an aspect weaver. These approaches are tightly coupled with specific AO languages.

A certain kind of language independence concerning the tooling is however reached using program transformation systems. In [15] a system for language independent weaving is described where generic transformation rules are specified using an abstract grammar and a grammar adapter transforms the generic transformation to a language-specific form. This work however targets the evolution of legacy systems, not product line engineering.

In [34] xApproach is described, which is a framework and tools for manipulating software engineering artifacts by transforming them into XML models and subsequently manipulating these representations. It takes advantage of the fact that there are standardized XML-based source code representations of programming languages like Java and C++. The tooling can be used for various use cases, e.g. separate implementation and weaving of aspects like security to achieve separation of concerns. A use case that is important in this context is the support for explicit variation points that can be foreseen in XML documents and bound by an operator at various points in time. Viewers convert the XML documents back to e.g. a specific programming language. This also allows for language independent AO and variability management.

Aspect weavers that manipulate the Microsoft .NET intermediate language (MSIL) are also to some extend language independent AO mechanisms. Both, aspects and base code can be implemented in any language that can be compiled to MSIL. Examples of such languages are Weave.NET [23] and DotSpect [13]. All of these languages can be used to implement crosscutting variations, but give not explicit support for variability management.

Sextant [35] is a tool that supports finding crosscutting concerns that span different kinds of artifacts, e.g. XML deployment descriptors, code, and documentation. It is well suited for tracing variations in the context of software product lines. The tool however does not allow manipulating artifacts.

pure::variants [36] [8] is a feature management and configuration tool that supports product derivation similar to Gears. It uses feature models as a language for describing the features and variants of a product line and their dependencies in the problem domain. The description of the solution domain is done in family models that hold components and their relationships, whereby component is a term for any kind of artifact in the solution space. Both models can be related to each other in the sense that features of the feature model are realized by components in the family model. For product derivation, the application architect can select features in the feature model and the corresponding elements in the family model are combined to a concrete solution. Additional transformation modules and a language for combining transformations support the production of concrete artifacts. The backend interface is open and allows plugging in custom generators.

In [10] the concept of mixin-based inheritance is introduced. Mixins become the basic definitional construct and inheritance is interpreted as mixin composition. Mixins are a concept of a programming language and cannot be used for any kind of software artifact as possible with Gears mixins. Even though the work does not target product line implementations, languages that support mixin-based inheritance can be used for feature implementation during domain engineering. During application engineering features can be composed using mixin composition.

The AHEAD tool suite [6] is based on the idea of step-wise refinement [7] whereby complex programs can be created from simple programs by incrementally adding features. Similar to our approach, AHEAD can work with any kind of textual artifact.

The main difference is that Gears provides explicit support for encapsulating crosscutting feature implementations.

7. Summary and Future Work

Software product lines aim at taking advantage of the commonality within a portfolio of similar products. For their success, effective management of feature variability is vitally important. Features in a product line often have widespread impact. They tend to crosscut not only multiple parts of individual artifacts but also multiple artifacts such as requirements, code, test cases, and documentation. AOSD can help in modularizing crosscutting variability, however there is no approach currently available to apply AO across different kinds of artifacts.

The variant management tool Gears includes a mechanism, called mixin, to capture and weave crosscutting variation into different kinds of artifacts when necessary. Different to AO, Gears targets all artifact types with the same approach. Mixins encapsulate crosscutting feature behavior and match patterns are used to specify the locales of text-level variations within a variation point. Substitution patterns are used to weave in feature-specific behaviors at or around the locales of match patterns.

In this paper, we explained the principles of Gears mixins and compared them to AOSD. We identified the advantages and disadvantages of both approaches depending on the characteristics of the variability in a product line. This serves as the starting point for further collaboration to explore the synergies between both technologies and together solve open issues such as integrating advanced AO concepts into variation management tools.

In the future we will also investigate in what cases traditional AO concepts are needed and when it is sufficient to capture crosscutting feature variability in mixins using simple text-based substitution mechanisms.

Acknowledgments

This work is supported by AMPLE Grant IST-033710. The authors would like to thank Markus Völter and Michael Kircher for their valuable comments on earlier drafts of this paper.

References

[1] ALVES, V., JR., P. M., COLE, L., BORBA, P., AND RAMALHO, G. Extracting and evolving mobile games product lines. In *SPLC* (2005), pp. 70–81.

[2] ANASTASOPOULOS, M., AND MUTHIG, D. An evaluation of aspect-oriented programming as a product line implementation technology. In *ICSR* (2004), pp. 141–156.

[3] APEL, S., LEICH, T., ROSENMÜLLER, M., AND SAAKE, G. FeatureC++: On the symbiosis of feature-oriented and aspect-oriented programming. In *GPCE* (2005), pp. 125–140.

[4] ARACIC, I., GASIUNAS, V., MEZINI, M., AND OSTERMANN, K. An overview of Caesarj. *Transactions on AOSD I, LNCS 3880* (2006), 135 – 173.

[5] BANIASSAD, E. L. A., AND CLARKE, S. Theme: An approach for aspect-oriented analysis and design. In *ICSE* (2004), pp. 158–167.

[6] BATORY, D. Feature-oriented programming and the ahead tool suite. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 702–703.

[7] BATORY, D., SARVELA, J. N., AND RAUSCHMAYER, A. Scaling step-wise refinement. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering* (Washington, DC, USA, 2003), IEEE Computer Society, pp. 187–197.

[8] BEUCHE, D., PAPAJEWSKI, H., AND SCHRÖDER-PREIKSCHAT, W. Variability management with feature models. *Sci. Comput. Program. 53*, 3 (2004), 333–352.

[9] BigLever Gears. http://www.biglever.com/, 2007.

[10] BRACHA, G., AND COOK, W. R. Mixin-based inheritance. In *OOPSLA/ECOOP* (1990), pp. 303–311.

[11] CLEMENTS, P. C., AND NORTHROP, L. M. *Software Product Lines: Practices and Patterns.* Addison Wesley, 2001.

[12] CZARNECKI, K., HELSEN, S., AND EISENECKER, U. W. Staged configuration using feature models. In *SPLC* (2004), pp. 266–283.

[13] DotSpect project. http://dotspect.tigris.org/, 2007.

[14] FILMAN, R. E., ELRAD, T., CLARKE, S., AND AKSIT, M. *Aspect-Oriented Software Development.* Addison-Wesley Longman, Amsterdam, 2004.

[15] GRAY, J. G., ZHANG, J., ROYCHOUDHURY, S., AND BAXTER, I. D. C-SAW and genAWeave: a two-level aspect weaving toolsuite. In *OOPSLA Companion* (2004), pp. 27–28.

[16] GRISS, M. L. Implementing product-line features by composing component aspects. In *SPLC* (2000), pp. 271–289.

[17] HETRICK, W. A., KRUEGER, C. W., AND MOORE, J. G. Incremental return on incremental investment: Engenio's transition to software product line practice. In *OOPSLA Companion* (2006), pp. 798–804.

[18] KANG, K., COHEN, S., HESS, J., NOVAK, W., AND PETERSON, S. Feature-oriented domain analysis (FODA) feasibility study. Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[19] KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., AND GRISWOLD, W. G. An overview of AspectJ. In *ECOOP* (2001), pp. 327–353.

[20] KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C. V., LOINGTIER, J.-M., AND IRWIN, J. Aspect-oriented programming. In *ECOOP* (1997), pp. 220–242.

[21] KRUEGER, C. HomeAway, Inc.: A case study in applying BigLevers three-tiered methodology. Tech. Rep. 20704301, BigLever Software, 2007.

[22] KRUEGER, C. W. Easing the transition to software mass customization. In *Software Product-Family Engineering, 4th International Workshop, PFE 2001, Bilbao, Spain, October 3-5, 2001, Revised Papers* (2001), pp. 282–293.

[23] LAFFERTY, D., AND CAHILL, V. Language-independent aspect-oriented programming. In *OOPSLA* (2003), pp. 1–12.

[24] LOUGHRAN, N., AND RASHID, A. Framed aspects: Supporting variability and configurability for aop. In *ICSR* (2004), pp. 127–140.

[25] LOUGHRAN, N., SAMPAIO, A., AND RASHID, A. From requirements documents to feature models for aspect oriented product line implementation. In *MoDELS Satellite Events* (2005), pp. 262–271.

[26] MEZINI, M., AND OSTERMANN, K. Variability management with feature-oriented programming and aspects. In *SIGSOFT FSE* (2004), pp. 127–136.

[27] OSGi Alliance. http://osgi.org, 2007.

[28] PAVEL, S., NOYÉ, J., AND ROYER, J.-C. Dynamic configuration of software product lines in ArchJava. In *SPLC* (2004), pp. 90–109.

[29] PINTO, M., FUENTES, L., AND TROYA, J. M. DAOP-ADL: An architecture description language for dynamic component and aspect-based development. In *GPCE* (2003), pp. 118–137.

[30] POHL, K., BÖCKLE, G., AND VAN DER LINDEN, F. *Software Product Line Engineering: Foundations, Principles, and Techniques.* Springer, Berlin, 2005.

[31] RASHID, A., MOREIRA, A. M. D., AND ARAÚJO, J. Modularisation and composition of aspectual requirements. In *AOSD* (2003), pp. 11–20.

[32] RASHID, A., MOREIRA, A. M. D., AND TEKINERDOGAN,

B. Special issue on early aspects: aspect-oriented requirements engineering and architecture design. *IEE Proceedings - Software 151*, 4 (2004), 153–156.

[33] RASHID, A., SAWYER, P., MOREIRA, A. M. D., AND ARAÚJO, J. Early aspects: A model for aspect-oriented requirements engineering. In *RE* (2002), pp. 199–202.

[34] REICHEL, C., AND OBERHAUSER, R. Xml-based programming language modeling: An approach to software engineering. In *SEA* (2004).

[35] SCHÄFER, T., EICHBERG, M., HAUPT, M., AND MEZINI, M. The sextant software exploration tool. *IEEE Transactions on Software Engineering 32*, 9 (2006), 753–768.

[36] SPINCZYK, O., AND BEUCHE, D. Modeling and building software product lines with eclipse. In *OOPSLA Companion* (2004), pp. 18–19.