Product Line Feature Typing: Identifying Complexity and Assisting Product Line Development

Sean Walton and Eric Eide
{swalton, eeide}@cs.utah.edu
University of Utah School of Computing

Abstract

Product lines encompass baseline and optional feature extensions that distinguish individual products. Managing these products is difficult both from a development and maintenance perspective, because the features' nature frequently differs. Categorizing the types of features can help product designers plan for product line expansion or re-targeting.

In this paper we present an evolving model called "feature typing." Software products have essentially two types of features: functional or *vertical features* (what the products do) and characteristic or *horizontal features* (what they have). Horizontal and vertical elements of features affect product development, deployment and support. Replacing, adding, or deleting features at various times of the product life translates into incurred company costs.

Purely vertical, purely horizontal, and horizontal/vertical features have unique characteristics that affect development and field support. In all cases, features tend to crosscut the program, so special consideration of feature characteristics helps better product development and deployment.

1. Introduction

Gone are the days when one tool fits all needs. Software development in the commercial field is gravitating toward alignment with the consumer desires for options that better fit their needs through the use of product lines[BAT02, DIK97]. A product line revolves around a baseline product with minimal core functionality, and from there, additional feature options are added to augment the baseline to fit into niches that address certain customer wants[CLE02]. In some cases, the baseline may not even be functional, because one or more options are necessary for interaction. For example, a baseline cell phone may not have a numeric keypad, and two feature options are keypad and voice command. This approach may improve product placement, but tracking each special version creates a need for extra management.

Product life management encompasses project management. Project management identifies and tracks the steps from design to release; product life management, however, tracks the features of a product line from version to version. In many cases, customers do not like to upgrade from a comfortable version, so the overall support timeline looks more like a continuum. Figure 1 illustrates this idea. The bold horizontal line represents the life of a product. Each circle consolidates the design, development, test, and deploy stages and is a baseline product version release. The radiating lines on each version are the product customizations. The line segments between each version represents the support phase, and since each time segment must support the cumulative versions, support clearly becomes a problem as the number of specialized versions increases.

The product line is not limited to the immediate version during development. In most cases, source code is carried from version to version. How and when the optional features are incorporated are important for product continuum support. Also, testing issues enter the picture as the quality assurance team must work with the feature combinatorics.

An essential problem is the identification, description, and management of individual features and their life cycles within the product lines. Although there are several existing models for supporting feature-oriented development, these models lack some analysis that can help illuminate the capabilities of managing a product line from a product planning point of view [ex. LEE02]. For instance, the target market drives the features it has, yet consumers generally want more out of the tools they purchase. Post-release reconfigurability and extensibility are two issues that often plague product designers. Because most features are crosscutting, adding, or *slicing in*[], new features can be difficult – especially later in the product life cycle[]. Clarifying the nature of a feature can help the designer to anticipate and address design and support limitations.

This paper proposes a new perspective on features called *Feature Typing*. Feature Typing is a work in progress to differentiate features of a commercial product and to define how they interface with the baseline product and its development. To illustrate the concepts, the paper describes a prototype program that utilizes each feature type and describes how those features integrate with a whole product. The discussion concentrates on embedded systems and many examples

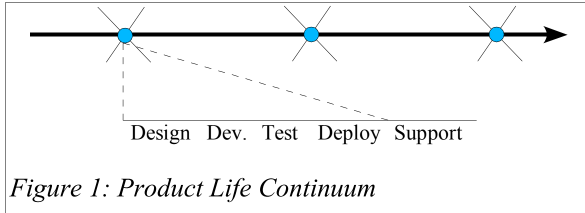


Figure 1: Product Life Continuum

are based on a cell phone. However, the presented concepts are applicable to other product development.

2. Basic Model Description

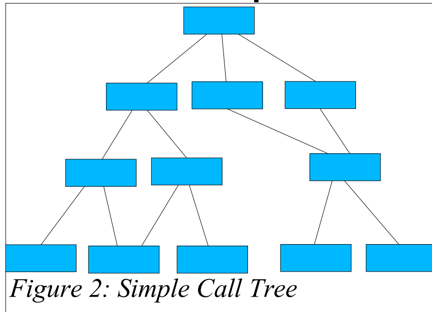


Figure 2: Simple Call Tree

This section describes the proposed new feature typing model that distinguishes vertical (functional) and horizontal (characteristic) elements. Categorizing in this manner helps assign attributes and expected behavior.

Most published descriptions and models of features focus on either *what does it do* (functional) or *what does it have* (characteristic). A "functional feature" represents a sequence of actions based upon some form of event or input and this notion is closely aligned with a FODA (Feature-Oriented Domain Analysis)/UML (Unified Modeling Language)-style use case [GOM00]. For example, a cell phone receives a call, stores the caller ID, and then alerts the owner. This is a functional feature, because it describes what the product does not how the product does it.

A "characteristic feature," on the other hand, is more static and describes those parts of the product that distinguish it. These parts are customarily devices or interfaces. Continuing the cell phone example, the cell phone has a display, transceiver radio, memory storage, and a beeper or a vibrator each having associated support program code. The devices may also be logical, such as a checkbox or an application window.

To better illustrate the concepts of vertical and horizontal features, it is useful to visualize a system as a call graph, such as the one in Figure 2. The program that responds to input begins at a stable state, receives some input, follows a general path through the system, and then returns to a new stable state. The use case is an example of this type of execution sequence. Regardless of how the program is designed (object or functional), the path through the system will always follow a general call tree. The leaves of the call tree are often the device interfaces or core algorithms (like ciphers or spellcheck dictionaries). For this paper, the root of the call tree is the stable state (we omit discussion of initialization for now). The functional features are *vertical features*: they begin at the top of the call sequence and follow a path vertically to the bottom (see Figure 3).

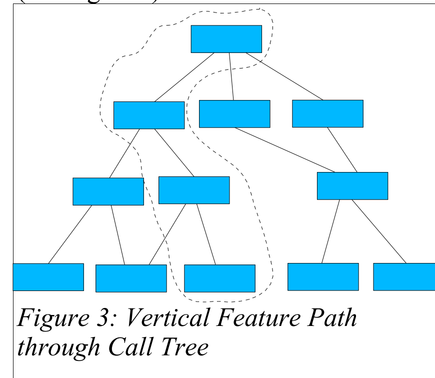


Figure 3: Vertical Feature Path through Call Tree

Continuing to view the call tree from the same top-to-bottom perspective, the leaves often are interfaces with particular devices or are special numerical algorithms. The devices are physical or logical characteristics of the product (e.g., a touch screen or an LCD display) and line up horizontally in the call tree (see Figure 4); therefore, they are called *horizontal features*.

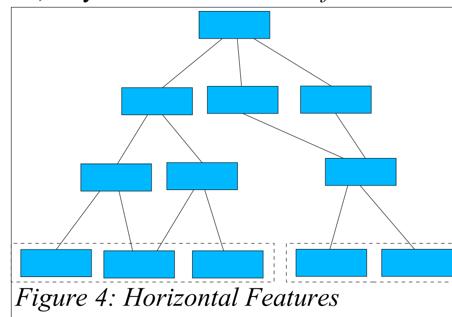


Figure 4: Horizontal Features

Once again, horizontal features represent the characteristics of the product; so, the layer orientation of these features in a call tree is completely arbitrary. In fact, groups of nodes can represent single or multiple horizontal features.

It may appear that vertical and horizontal features are interdependent, one needing the other. However, they are almost orthogonal, because a product can have a vertical feature without a horizontal component (called a *pure vertical feature*) if the augmenting feature effectively adds paths (and sometimes boxes) to the middle of the tree with no new leaves. An example is cell phone text messaging: the message can use the sample protocols and user interfaces. The meaning of the user interfaces simply differ from the original.

Similarly, but with some constraints, a product can

have a horizontal feature alone without adding the vertical feature support (called a *pure horizontal feature*). The program call tree replaces or gets new boxes. No new use cases path are introduced. For instance, some cell phones can switch networks just by swapping a SODIMM, and an Internet browser can display PDF documents or 3D imaging just by loading a plugin.

Orthogonality is important for managing the product life continuum, because upgrades, updates, and patches are intrinsic to product support. If the product designer can plan for future product versions, the needed hooks can be incorporated. These can be invisible to the consumer, giving longer lifespan to the product.

3. Feature Characteristics

A baseline product often requires additional functionality to make it marketable in specific niches. “Slicing in” vertical and/or horizontal features imply product/schedule impacts at different times in the product life cycle. Anticipating features can save investment. One cellphone manufacturer appears to have understood this when they included an additional – and unused – custom microcontroller in the cellphone design so that they could do an in-field firmware upgrade when such support was needed []. The result could be lower costs and higher customer retention.

The rest of this section defines the individual feature types in our model and outlines the impacts they have on product development.

3.1 Feature Type Details

Despite the marketing attractiveness of product lines, managing such a project can be difficult. Design and development are only parts of the product release schedule, yet they receive the most attention despite the fact that well tested and deployed units and good support keep the customers coming back. By understanding the nature of horizontal and vertical features, product management as a whole can better incorporate the capabilities and plan for the complexity.

Horizontal/Vertical Feature (H/VF) As in the cellphone manufacturer's design example above, the most common feature slice is the horizontal/vertical feature (H/VF) that combines program function with product characteristics. These are customarily designed into the product before release, and comparably, have the largest code footprint. See Figure 6 for a summary.

Purely Vertical Feature (PVF) The purely vertical feature (PVF) is less common than an H/VF, but is not rare. In-field updates on mobile devices are fairly common to fix defects in firmware code. Still, feature upgrades can be included as well. In fact, the cellphone example, above, could be considered a PVF because the hardware were already in place; i.e., the upgrade needs no new characteristic (chip), assuming the firmware support is already installed.

- Adding NAT capabilities to a personal router.
- Changing functions on a calculator or microwave.
- Adding instant messaging to a cell phone.

Figure 5: Purely Vertical Feature Examples

PVFs are additional functionality or new interpretations to similar input. For example, adding a square root operation to a simple calculator is an H/VF, because a user interface needs to change for the function (vertical feature) plus the algorithm for square root (a horizontal feature, because the algorithm itself does not go up to the interface level). But, then, adding a distance function is purely vertical because the basic square root function already exists. See Figure 5 for examples.

- Adding GPS to a PDA.
- Adding graphics or ADC chip to a system.
- Adding a new plug-in to a browser.

Figure 6: Horizontal/Vertical Feature Examples

Purely Horizontal Feature (PHF) A purely horizontal feature (PHF) is likely to exist somewhat frequently in the pre-release code, since programming is simple and reuses a lot of code. However, PHF updates are the least common in the embedded world because of the hardware and firmware required to support it. PHFs do not add any functional features. Instead, PHFs rely on existing program code to support PHF enhancements.

Post-release horizontal features require a standardized interface through which the device identifies itself to an in-field installed device driver. Examples of post-release these interfaces are becoming more commonplace: Bluetooth and USB are two standardized interfaces that enable in-field H/VF installations.

A PHF enhancement is unique in that it extends the product without user intervention. One example is the hardware/firmware update that changes a wireless device from one network to another. Lastly, PHF introduces the least code footprint and has few crosscutting concerns, because it changes only the device code associated to it and is often completely hidden from the user. See Figure 7 for examples.

- Another watering zone in a sprinkler system.
- Another light, radio, or sensor to a sensory network mote.
- New cypher to an SSL package.
- Browser plugins.

Figure 7: Purely Horizontal Feature Examples

3.2 Feature Comparisons

In order of magnitude, H/VF, PVF, and PHF have lessening complexity in terms of feature slicing, but they have increasing complexity because of base-product support (see Table 1). Comparably, adding a horizontal and vertical feature crosscuts more than either feature by themselves. Similarly, a vertical feature crosscuts more of the product and digs deeper from top to bottom of the call tree than the horizontal feature revises up the tree.

Needs – What the feature needs to work correctly.

Impact – When can the aspect be woven in.

Invasiveness – How much of the code has to be touched by the aspect.

Inflation – The increase in program size after weaving.

Complexity – The feature's complexity based on the baseline (core) program or an added feature.

	PHF	PVF	H/VF
Needs:			
Vertical:	Yes	Perhaps	Perhaps
Horizontal:	Unlikely	Yes	Perhaps
Predefined Interface:	Yes	No	Perhaps
Commonality:			
Design:	Many	Moderate	High
Development:	Moderate	Many	High
Testing:	Few	Few	None
Deployment:	Few	None	None
Support:	Some	(Replace)	(Replace)
Invasiveness:			
Crosscuts:	Few	Some	Most
Inflation:			
Code:	Least	Moderate	Highest
Data:	Moderate	Low	Highest
Complexity:			
Feature:	Low	Moderate	High
Baseline:	High	Low/None	Low/None

Table 1: Feature Type Summary

On the other hand, PHFs require more baseline product support (this support could be included in a vertical feature), meaning that the program code may have to include “glue code” that could be used as plug-ins later. The standalone glue is unlikely to be tested, and for embedded devices, in-field code corrections is complicated and costly. This complexity is often unwelcome to the QA staff.

4. Impacts on Product Life Continuum

A designer, once understanding the different types of features, needs to know how each might affect the project time-line and product quality. This section poses some ideas that might help point to potential problems.

4.1 H/VF Impact

A H/VF has many crosscuts throughout the project code, since its effects begin at the user interface and reach into the leaves of the program. The leaves (particularly devices) are frequently the most difficult to debug; this is further complicated if the new device support is located in an aspect advice.

Delays from before the QA phase significantly hamper validation, and many test cases have to be revised to account for the modification. Even if the use cases for the feature are well defined and the scenarios are easily enumerated, testing is significantly impacted if incorporated after design is complete. Therefore, H/VFs are best included in the design of the baseline product.

During the product life cycle, it is possible to retrofit an H/VF during the product support phase, but the designers need to include the needed device hooks is the base product. The hooks could be in the form of a function table that extends the features to the rest of the program. However, because selective deletion from a function table can cause corruption, feature removal (or replacement) is very difficult without wholesale firmware replacement or other extraordinary efforts.

4.2 PVF Impact

A PVF does not touch as much of the program code as an H/VF, because at some point, it uses program code some already in place. Complexity depends on how much and how deeply the new feature digs into the calling tree. As the feature digs deeper, the PVF begins to be limited like an H/VF. Therefore, a “shallow” PVF could be added as late as pre-deployment.

Post-deployment and into the support phase, the program code is less accessible, and users are often

reluctant to change something that works for them. It may also be that the code is inaccessible, as in an embedded device. PVFs are, therefore, very difficult to install in the field. Plug-in methods can alleviate post-release installation. Like the H/VF, removing a PVF faces the same table corruption dilemma.

4.3 PHF Impacts

Unlike vertical features, if the program is carefully written, adding or removing PVFs can be very simple. The program must dynamically recognize the new horizontal feature and incorporate its function in the runtime functionality.

H/VFs, PVFs, and PHFs each behave differently in integration, deployment, and support. It is estimated that as the feature becomes more vertical, the ease to integrate later in the life span diminishes rapidly. So, planning ahead can help lengthen the lifespan of a particular product, if so desired.

5. Case Study: Wireless Net Simulation

While postulating the nature of features, some of the ideas had to be tested. We created a prototype product line to demonstrate the different types of crosscutting feature slices. This prototype needed to be "interesting" to enable feature extensibility; also, the sliced-in features needed to be natural (not contrived) to a user of the prototype.

This section describes the prototype and the added features. This discussion is organized by feature type, not project time-line. A project post-mortem follows the product description, and summarizes the lessons learned from our implementation of the prototype product line.

5.1 Baseline Program

We first created the baseline program using AspectJ, and then afterwards selected and augmented functionality with natural features that demonstrate each feature type. Over time, we converted the features into aspect advice to allow feature selection.

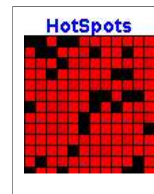
The baseline program simulates a field of wireless motes in a window GUI; and each mote is modeled by an independent thread. The baseline works independently from the optional features to ease testing. The program simulates the communications between each mote in a "perfect" network where no communication loss is experienced. Each mote receives, transmits, and routes messages to neighboring motes. At the beginning of the simulation, all the motes determine which are the nearest neighbors using a broadcast ping. One mote is designated as the "sink" (the black square in the lower middle of Figure 8), i.e., the destination for status messages.

Each mote has peer-to-peer messaging with message loop recognition. The networking subsystem also has a very limited routing cache of nearest neighbors and known route to the sink. The motes wait (sleeping) for an event to either route a message, accept and process a message, or send status.

During the simulation, each mote periodically transmits its status to the sink through its neighbors. Lastly, each mote sends a broadcast ping when clicked, for program verification.

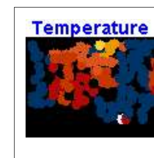
On top of this base program, we added various optional features. The following subsections describe each of the horizontal/vertical, pure vertical, and pure horizontal features.

5.2 Horizontal/Vertical Features



H/VFs are the easiest to design in a product, and three augment the functionality of the simulation. The first is a graphic component that shows which motes are successfully getting their status messages to the sink. This is an example of a H/VF, because the vertical element permits the "user" (sink) to translate the status message into a dot on the display, and the device is the visible component. When a mote transmits its status through the network, the sink marks that mote as still "alive." The mote liveness appears as a NxM matrix of squares with a fading color. Over time, it becomes obvious which motes are unable to get their messages routed through because of routing islands.

The second H/VF tracks the path that the mouse takes as it moves over the field of motes. This is an example of an H/VF, because the vertical element allows the user (person) to move the mouse over the field, and the horizontal element is the visible component. This feature enables each mote to alert the sink when the mouse passes over it. The sink, in turn, tracks these messages in a graphical component that shows the path (see the inset box labeled "Mini-Map").



The last H/VF grabs a weather (national temperature) image off the Internet and uses the pixel data as sensory input for each mote. This is an example of an H/VF, because the vertical feature permits the users

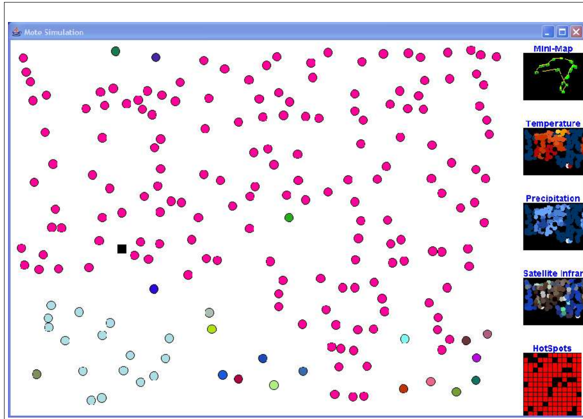


Figure 8: Wireless Net Simulation

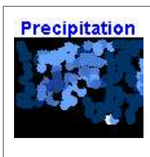
(motes) to transmit and interpret, and the horizontal feature is the visible component. each point to the The mote transmits this information as part of its status, and the sink directs the data to a component which displays the color-data on a miniaturized image (see the inset box labeled “Temperature”).

5.3 Vertical Features

PVFs are more difficult to implement without several horizontal features to support them; nevertheless the prototype successfully includes two PVFs. The first PVF is a simple test of the mote's network. When a mote is clicked, it sends a broadcast message to its neighbors to change color. This message is propagated until the route path visibly drops off. This vertical feature relies on the established program code to route messages. Note in that several motes appear to have the same shade. The sink was clicked to started the simulation, and at the same time, a “change color” was sent.

The second PVF stops each thread when the GUI loses focus or is iconified, and when the focus is restored, the threads resume. The PVF, in this case, depends on the existing program code that manages the window events.

5.4 Horizontal Features



PHFs are the most difficult without initial support code: the program has to include logic that detects and uses new horizontal features without additional use cases. The last two examples utilize the image logic in the H/VF, above, and adds national precipitation and satellite infrared images to the set of components on the prototype (see the two inset boxes labeled “Precipitation” and “Satellite Infra”). The devices are the components themselves.

In this example, the initialization program code for the component includes logic to get and convert the images. On the top level, the program queries to determine the number, names, and IDs of each resource available. Then, it creates the visible components and links them to the incoming messages. When the component gets a message, it pulls out the data that it needs. Since the Temperature H/VF covered most of the implementation, these PHFs only needed to have the resource defined in the initialization code.

5.5 Prototype Conclusions

The prototype successfully and effectively demonstrates the different types of features and allows for selective slicing, creating a compile-time customization that is consistent with product line expectations. Even though the program may be somewhat artificial, because it attempted to demonstrate concepts that this paper presents, it provided important insights to our model of vertical and horizontal features:

1. The baseline had to be very solid before proceeding with slicing in the new features. Testing new features with a buggy baseline was intractable.
2. We found it easier to add and test a vertical or horizontal feature in the baseline. Then, after verification, we pulled out the new feature and converted it into aspect advice.
3. Getting the AspectJ advice to work correctly required unusual programming when manipulating the thread loops.
4. Incremental augmentation usually did not interfere with earlier advice statements (perhaps due to the small program size).
5. De-aspectizing was fairly simple to do – as long as the source was fully understood (large source trees may not benefit here).
6. Plug-in support for PHFs was fairly straightforward but required some planning and adjustments to the status messaging.
7. Once the support for PHFs was in place, adding new graphics components to the program was very easy.

The prototype program illustrates the changes that a designer and developer need to make in order to integrate the described feature types.

6. Model Limitations

Feature typing is our attempt to define the nature of software feature. Our current work and experience with feature typing has some limitations. The analysis and prototype proved the existence and usefulness of the three different types of features, and distinguishing the "does" feature from the "has" feature is helpful in design, development, and support. However, the model's H/VF is still too broad and remains a large domain – most of the implemented features in product lines fit primarily in this category. Nevertheless, some H/VFs may tend to be more like either the PVF or the PHF, thus opening the way to derive more specific conclusions. Also, the prototype, and, in part, the model itself, relies heavily on aspect oriented programming (AOP) which is not universally accepted in the commercial domain. Programmers who wish to take advantage of feature slicing in their product lines may have to use alternate methods.

7. Related Work

Our feature typing model parallels the concepts found in *Generative Programming* (Czarnecki/Eisenecker). They use, in fact, the terms "vertical" and "horizontal" to describe scopes and domains [CZA00]. This paper revises this usage to discuss the terms in the form of products in a product line. Because feature typing defines the feature and spans the product life, the methods learned in generative programming are complementary and can crossover easily.

Our work is also related to aspect oriented programming (AOP). This research relies heavily on AOP to slice in new feature options and does not conflict with AOP's model. However, understanding and use of this paper's model requires access to AOP concepts and tools [KIC97].

Batory, et.al., have investigated software product lines [BAT02, CLE02, DIK97]. While this paper focuses on embedded systems, which more closely ties software to the hardware, the model presented here attempts to refine the product line concepts in an effort to assist design, development, deployment, and support. Alone, the new model can assist other efforts, but product lines are an immediate suitable consumer.

The fourth related model is called feature oriented programming (FOP). Again, this model does not conflict with FOP; instead, it enhances the layering model with feature categorization [].

The another model is named "Feature Modeling" which categorizes each feature based on domain, and then, associates each feature based on dependence [LEE02]. The feature typing model is non-domain specific, but uses Lee's work to define the interdependence of a product's feature set.

Jacobson, et.al., have researched aspect-oriented software development use cases (AODUC) [JAC05, STE02]. Again, like other related work, the feature typing model is complimentary with AODUC, because AODUC expresses how to represent the aspects from a development perspective, whereas the feature model provides more detail on the features themselves. This empowers the developer to apply features in different phases of the development, consistent with project design.

8. Conclusion

Product line development depends strongly on the capability to create products with a similar baseline functionality but differing options or add-ons. Features play an important role in augmenting the baseline into desirable products. Understanding of the general nature of features can assist designers identify when and how certain customer options can be integrated.

Surveying the market, features are identified by *what the product does* and *what it has*. This paper introduced *feature typing* and explained the concepts behind three types of features using a combination of functional and characteristic features, labeled "vertical" and "horizontal" features. To explore the nature of these types of features, a program prototype was developed. This paper discussed the results of this exploration and provided some conclusions.

Feature typing has the capability of redefining how and when a product is developed and supported. In a time when security is at odds with users wanting to "stick with what works," having the capability to dynamically certain types of features easily is very powerful for both the user and the developer.

References

[BAT02] BATORY, D.S., LOPEZ-HERREJON, R.E., AND MARTIN, J.P. "Generating Product Lines of Product-Families." ASE, 2002

[CLE02] CLEMENTS, P., NORTHROP, L. *Software Product Lines.* Addison-Wesley, 2002.

[CZA00] CZARNECKI, K., EISENECKER, U.W. *Generative Programming: methods, tools, and applications.* Addison-Wesley, 2000.

[DIK97] DIKEL, D., KANE, D., ORNBURN, S., LOFTUS, W., AND WILSON, J. "Applying Software Product line Architecture." *IEEE Computer*, 1997.

[GOM00] GOMAA, H.. "Object Oriented Analysis and Modeling for Families of Systems with UML." In

W. B. Frakes, editor, *Proceedings of the Sixth International Conference on Software Reuse*, June 2000.

[JAC05] JACOBSON, I., and NG, P. “Aspect-Oriented Software Development with Use Cases.” Addison-Wesley, 2005.

[KIC97] KICZALES, G., LAMPING, J., MENDHEKAR, A., MAADA, CH., LOPES, CH., LOINGTIER, J.-M., IRWIN, J. "Aspect-Oriented Programming." Proc. of ECOOP W7 (lyvaskyla, Finland, Jun. 1997), LNCS 1241, 220-242

[LEE02] LEE, K., KANG, C.K., AND LEE, J. “Concepts and Guidelines of Feature Modeling for Product Line Software Engineering.” In C. Gacek, editor, Software Reuse: Methods, Techniques, and Tools: Proceedings of the Seventh Reuse Conference (ICSR7), Austin, USA, Apr.15-19, 2002, LNCS 2319, pages 62–77. Springer-Verlag, 2002.

[MEZ04] MEZINI, M., OSTERMAN, K. “Variability Management with Feature-Oriented Programming and Aspects.” *ACM SIGSOFT,* 2004.

[STE02] STEIN, D., HANENBURG, S., AND UNLAND, R. “A UML-based Aspect-Oriented Design Notation for AspectJ.” *ACM AOSD Proceedings*, 2002.