

# Feature Typing: An Early Aspects Technique

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

## Abstract

*An implementation and deployment plan is critically important in the development of a software product or product line. A good plan results from an informed feature analysis that incorporates crosscutting concerns such as variability management and the project time-line. A poor feature analysis, on the other hand, can result in an ineffective project schedule and an implementation that hinders the delivery of new features. An early understanding of product features can help a project planner create an effective plan, and help a designer avoid unsupportable code.*

*Feature typing is our emerging model that describes what features are in terms of two separate natures. The vertical nature of a feature generally follows a UML's use-case path, describing what the feature does. The horizontal nature expresses a program characteristic, describing what the feature has. Because each nature affects feature interaction and product integration and testing, early identification and typing can increase predictability of project time-lines and reduce code complexity.*

*This paper details the feature typing model and its application for two separate case studies. The studies demonstrate the ways that feature typing can help with the design, planning, coding, and testing of feature implementations. The studies also illuminate the limitations of the approach.*

## 1. Introduction

An important step in the development of a software product or product family is the creation of an implementation and deployment plan. After requirements are known and an overall architecture is defined, one must still decide how to map parts of the architecture onto (1) detailed software design elements and (2) a strategy for implementing those elements over time.

Not surprisingly, the translation from an architecture to a design and implementation plan depends greatly on the

properties of the requirements and features of the product or products being developed. For example, if the feature model [4] of a product line contains an optional feature, that variation must be accommodated in the implementations of the affected features. One might choose to implement the optional feature late in the implementation phase—or even as a separate add-on product, to be deployed separately. If the optional feature is implemented late, one can decide to implement the attachment points in the “base product” either early (planned extension) or late (incremental development).

The overall nature of a feature is a combination of many properties, both inherent and imposed. For instance, optionality is a *extrinsic* property: a feature is not optional by nature, but by the role it plays in a product line. Other qualities arise from features themselves, and these *intrinsic* properties must also be considered during the creation of a implementation and deployment plan. Some features encapsulate basic services (e.g., computation) or qualities (e.g., security); others coordinate basic services for useful purposes (corresponding to use cases); and some are combinations of these things. Understanding these different natures is essential in the creation of an implementation and deployment plan. A gap in knowing how deeply features affect the overall product creates uncertainty, which can lead to inefficient or incorrect implementation and maintenance decisions later in the software life cycle.

This paper outlines an emerging model called *feature typing* that describes the natures of software features in a way that helps a project planner create an effective implementation and deployment plan. Feature typing is an early aspects technique because it is applied at the design phase and deals with the impact of a crosscutting concern: namely, the relationships between feature natures and the overall, coordinated strategies for design, implementation, and deployment. We suggest that the intrinsic qualities of a feature can be described as a combination of two natures: functional or *vertical* (what it does) and characteristic or *horizontal* (what it has). This classification supplements other descriptions of features, such as feature diagrams [4] and the distinction between “functional” and “non-functional” requirements [2]. Early identification of

---

This material is based upon work supported by the National Science Foundation under Grant No. 0410285.

vertical and horizontal natures can provide insight for subsequent implementation, such as which features—or parts thereof—should be coded with aspect-oriented programming (AOP) techniques. Feature typing can be applied to the design of a single product, but we believe it is especially suited to the development of product lines and sets of related products, which have a core product and several feature options.

## 2. Model Description

Starting from a standard feature model [4] that describes the abstract parts and variabilities within a software product or product line, *feature typing* classifies features as combinations of functional and characteristic natures. These natures describe how a feature interacts with other features, and therefore impact directly on the design of a feature's implementation and integration with a program in general. Feature typing builds on feature modeling and uses its existing concepts to describe variability and dependencies.

### 2.1. Feature Natures

Features have essentially two elementary parts: the *vertical* part defines what the feature functionally does, and the *horizontal* part describes what characteristics it has. The vertical nature closely follows or is similar to a UML-style use case. For example, a “print” use case in a word processor begins with a user interaction and proceeds to (1) determine the printer type, (2) rasterize the image, and (3) send the image to the printer. The horizontal nature of a feature, on the other hand, consists of the “building blocks” of a program through which vertical features pass, i.e., the steps of a use case. Examples would include the action definitions for intercepting a button click or the interpretation of a temperature sensor value. Horizontal elements can include user-interaction devices (e.g., menus and buttons): input services focus primarily on internal workings and do not necessarily follow a use-case path. The horizontal part of a feature can be viewed as a self-contained object (or objects) offering services to the vertical counterparts, but a horizontal part can require services, too. In many cases, a complete feature will be a combination of vertical and horizontal elements.

Feature typing should not be confused with models that distinguish *functional* and *nonfunctional* requirements [2]. Functional requirements describe behaviors that accomplish desired tasks, whereas nonfunctional requirements describe constraints or attributes of the implementation (e.g., performance). Feature typing focuses on the analysis of features that are derived from functional requirements. It is future work to extend the model to characterize discrete features that encapsulate nonfunctional concerns.

### 2.2. Feature Types

Feature natures can be combined to form different *feature types*. Each type expresses the overall characteristic or constraint of a feature, arising from its natures, on program development. Each feature type represents a level of complexity and level of impact on other product code. This complexity and impact are often of great concern to developers and testers. Classifying a feature's type during the design phase, *before* it is implemented and integrated, narrows the uncertainty gap that is often encountered in project management. Feature typing can also improve feature management in general.

The most common type of feature, a **horizontal/vertical feature (H/VF)**, combines vertical and horizontal elements to perform its expected functionality. For instance, the print (send to printer) function in a word processor allows a user to convert a document into some printer language and then send it to the desired printer. The steps from selecting “print” up to transmission to a device is the vertical part of the printing feature. The module that manages the printer functionality and directs the data stream to the appropriate operating system device is the horizontal part of the feature. As in this example, an H/VF provides the tools along with the use case. H/VFs touch several parts of the system; thus, they should be considered early in the design phase and are likely to be part of the “core” product implementation. A purely vertical or purely horizontal feature, as described below, can augment a software design without as much impact as an H/VF.

A **purely vertical feature (PVF)** relies on horizontal features and adds new use cases, thus expanding the usefulness of the product. A PVF is a feature that omits a horizontal element. As stated earlier, a vertical element is a path of steps through the program, so if a feature omits a horizontal element, it must hook into established blocks at one or more points. Referring back to the word processor print function, a user typically accesses printing through a menu, but he or she may use a hot key command instead. Both provide the same functionality but by different paths; so, assuming the menu is the H/VF, the hot key is a PVF to the same functionality. A more interesting PVF is found in document automation, such as mail merge. When a user prints, the meaning of the base print command is augmented by the mail-merge feature. It still uses the foundational print component, but between that and the print command, the new PVF performs additional work of pulling data, merging it into the document, and reformatting.

Augmenting a product with a PVF requires up-front planning during the design phase to ensure that necessary interfaces are well established in the implementation. As many product managers can attest, however, customers often change product requirements during development. Be-

cause PVFs have small impacts on other parts of a product (compared to the impacts of other feature types) and their use of existing interfaces, PVFs appear to be the cheapest to add to an in-flight project.

A **purely horizontal feature (PHF)** adds components to a product without adding use cases. Again, horizontal features represent the characteristics of the program (what the program has). Example PHFs might include alternate printers (local or remote) and printer drivers; a module that pulls merge data from an SQL database; or a new sprinkler zone in a sprinkler management system. Adding or extending PHFs to a product requires up-front design to handle the changing characteristics, especially if the PHF is developed as a separate product—an add-on, deployed after the base product is sold.

Feature types allow a product designer to better classify each feature that is integrated into a product. Moreover, foreknowledge of the typing process can “feed back” into a project time line and strengthen the process that an architect uses to plan a product line feature set. PHFs, as modules in the system, may be expected to have the least interaction with other features. On the other hand, the vertical features—ones that really do coincide with use cases—can be flagged as potential causes of feature interaction. Early identification of PVFs and H/VFs could help identify places where the work of Lee et al. might assist in detangling interdependencies [8].

### 3. Applying the Model

During the design and planning phases, a designer considers the user requirements and software architecture and attempts to plan the remainder of the project, from implementation to deployment. At this time, feature typing can assist in feature impact analysis, cost analysis, and project scheduling. This section summarizes how feature typing can be applied to identify the types of product features and thereby improve the design, implementation, and deployment plans.

Vertical features generally follow use-case paths through a system, as stated earlier. There appear to be two separate vertical feature types. The first corresponds to the user-required scenarios defined in the use cases (user-initiated cases); the second involves background tasks to keep the system running correctly (background cases). Background cases include such things as maintaining connections, updating visual data, and taking snapshots of user work. The background cases can be integrated within the base system, or they can simply be tasks that fire without user intervention. In either case, both must be identified in the design.

The designer and planner analyze the use cases to identify individual H/VFs, PVFs, and PHFs. This stage helps

the designer, manager, and customer know the difficulty or ease of implementing each individual feature. The feature types affect the delivery schedule and post-delivery support. Because H/VFs often touch much of a system, they become the focus of phased delivery in incremental deployment strategies. PVFs can be used to augment the system without as much effort as the H/VFs and may provide time-line flexibility. Finally, PHFs require additional design work up front if they are to be added later. The feature types may be identified using the descriptions below.

An **H/VF** is recognized as integrated function and characteristic; removing either one changes the feature’s behavior. The steps in each scenario involve working with different components, represented as nouns in the description, in the system which are important for the operation of the system function. H/VFs must be included in an initial design; integration postponement increases costs and time.

A **PVF** is identified as a scenario whose steps either eventually coincide with another scenario, or whose last scenario step refers to a different, specific scenario step.<sup>1</sup> PVFs may be added later to the design, after implementation has begun, assuming that the infrastructure exists to support them.

A **PHF** appears as a selection list in the scenarios. The list must be dynamic in nature: it must be flexible, allowing a product manufacturer or user to add and remove items. The list may not be something programmed; instead, it may represent configuration options, such as a list of target platforms. The items must have similar interface characteristics with an existing horizontal nature of an H/VF. Dynamic PHFs must be provisioned for at design time. An implementation may not be able to integrate unplanned PHFs.

The classifications H/VF, PVF, and PHF imply traits that affect implementation and delivery times. This helps in estimating the time-line and scheduling the best times to begin implementing particular features.

### 4. Case Studies

We applied our feature typing model to two case studies, the Java Pet Store and a wireless network simulator.

---

<sup>1</sup>One might confuse the device that initiates the PVF use case (e.g., a hot key or touch screen) as a horizontal feature, and thus incorrectly label a PVF as an H/VF. The input device is not necessarily a horizontal feature, since it only starts the use-case sequence. If, however, the device is also involved with the output of the use case, then it truly would be an H/VF because it has both functional and characteristic elements.

In the case of the Java Pet Store, we followed a reverse-engineering approach and attempted to add new functionality to the existing system. The wireless simulator, in contrast, involved the development of a system from scratch. Our results were mixed. We found that our model was applicable in the design stages in both cases: we were able to define and categorize features for the two programs, and we succeeded in incorporating new features in the wireless network simulator. However, we faced roadblocks when trying to implement new features in the multi-tiered J2EE design of the pet store. This section details the two case studies and critiques the results.

#### 4.1. Java Pet Store

The Java Pet Store [12] is an example Web application based on the J2EE architecture. To determine how we could apply the feature typing model to this existing system, we began reverse engineering the work by:

- ✓ Identifying the features of the program,
- ✓ Applying the model to those features, and
- ✓ Exploring extending program with new features.

These steps test the applicability of feature typing for a scenario in which new and *unplanned* features are added late in the development cycle. This is an interesting test, although we intended feature typing to be applied early in the development of software products, for *planned* extensions.

Our first task was to characterize the existing features, including the discoverable vertical and horizontal elements. During our analysis, we performed a white-box and a black-box review to get a thorough understanding of the program. First, we identified the following use cases:

- |                              |                             |
|------------------------------|-----------------------------|
| • Change localization (H/VF) | • Browse inventory (PVF)    |
| • Create user (H/VF)         | • Put item in basket (H/VF) |
| • Login user (H/VF)          | • Check out (H/VF)          |
| • Logout user (PVF)          |                             |

We organized an initial feature model around these use cases. The types of the features in that model are as shown.

The initial use cases and features rely on a three-tier model that consists of a Web interface (JSP), intermediate glue (Java), and a database (PointBase). These tiers comprise horizontal elements of the overall system. In the list shown above, for cases marked as H/VF features, the feature adds functionality at each tier. For example, *Login User* is an H/VF because the state for “logged in” must be added at every level. *Logout User* is a PVF: although it changes a user’s state, it does not add new state.

We successfully identified and categorized each viewable feature. In some cases, however, the categorization was estimated from what was viewable and our basic knowledge of J2EE. We did not include the middleware itself in our white-box review—we looked only at the Java and JSP code of the Pet Store. We were therefore unable to make a complete appraisal of the existing system, since much work is done inside the J2EE framework.

We then explored extending the program with new feature candidates. We wanted our candidates to (1) be useful, meaningful, and interesting; (2) demonstrate the use and significance of the feature typing model; and (3) minimize revisions to the existing code. The extensions assumed a black-box view of the system and did not take into account any knowledge gained from the white-box analysis. Our proposed new features included:

- Create a new localization for Spanish (PHF)
- Save/restore the basket between sessions (PHF)
- Add a new payment method (PHF)
- Add searching capabilities (H/VF)
- Add emailing to current customers (H/VF)

Unfortunately, the implementation of these features ran into roadblocks. With the exception of localization (which uses externalized strings), the new features were unprogrammable without changing the Web and middleware tiers. The intra-tier calls in the existing code were inflexible, and changes meant rewriting sections of code to accommodate our new (mostly horizontal) components. Significant rewriting would have been required even if the new features were “grafted on” via AOP techniques. We even attempted to work between the Web and middleware tiers, as would be required for some of the extensions above, but those layers suffered the same limitations.

We draw three main conclusions from our experience with the Pet Store case study. First, feature typing was applicable to the features we found: we were able to identify and classify seven existing features and propose five new ones. Second, feature typing appears to be best used in the initial design and planning stages: i.e., it provides limited benefits in cases of reverse engineering and unplanned extension. Third, multi-tier environments present a challenge to developers that want to distinguish the different feature types. This case study is a work in progress, and we plan to pursue alternative approaches in future work.

#### 4.2. Wireless Network Simulator

We worked on a second case study that applies the concepts of feature typing during the initial design of a software product. This case study implemented a simulation of

a wireless network. Using the case study, we were able to demonstrate the different feature types and benefit from the analysis.

The purpose of our test application was to simulate a planar field of 200 wireless, communicating, sensory devices (“motes”). Each mote simulates a reliable peer-to-peer datagram protocol, implemented simple routing, and is represented as an independent Java thread. One mote (the “sink”) is designated as the destination for all status messages. During network boot-up, the motes attempt to find and route to the sink.

We first defined a feature model of the complete program. After identifying a set of “core” features, we came up with a set of additional features that would add value to the basic simulation. (An additional goal was to find features that exemplified the different feature types.) Some of the peripheral features that we added to our model are the following:

**Minimap (H/VF)** — a component showing a point-to-point trace of where the mouse moved over the field of motes.

**Site Temperature (H/VF)** — a component using a real-time Internet weather image of the United States. Each mote reads its point over the map and transmits it with the status message. The sink then translates the data into a blip on a small component.

**Change Color on Click (PVF)** — a feature relying on the peer-to-peer messaging to send a message to change its AWT color.

**Stop/Resume Simulation (PVF)** — a purely vertical feature using the window messaging to stop and start the simulation based on window focus. When the window loses focus or is iconified the threads stop; likewise, when the window regains focus, the threads restart.

**Site Precipitation (PHF)** — a feature relying on the *Site Temperature* feature. It adds a new component that represents the real time national precipitation.

These example features created an interesting product. By designing the features to be optional, we created an interesting product line.

We then applied feature typing to organize the subsequent design and implementation steps. With the exception of the *Stop/Resume Simulation* feature described above, each product feature was planned and designed during this initial system development. We were able to type all of the features in our model, and using that information, produce a well-organized implementation plan. First, we implemented and tested the features in the core of the simulator. Due to the complexity of the system, we chose to use

an iterative implementation strategy, knowing that module rewrites would be inevitable. In other words, we chose to code basic functionality before we coded variability mechanisms for other feature attachments. Second, we added extension points and implemented the peripheral features using aspects in AspectJ. This process was straightforward, and because of the way that we planned the product, features could be hand-selected at deployment.

An interesting trade-off that we anticipated was the implementation of *Site Precipitation* PHF, which depends on *Site Temperature*. The development plan postponed support for multiple data sources until other parts of the system functioned correctly. When implemented, the core program was modified to recognize a dynamically managed list of image resources. The plan that arose from feature typing held: the implementation of the changes went smoothly via planned aspects.

### 4.3. Discussion

The case studies showed that feature typing can help with design and planning. Feature typing can support both reverse engineering and new development tasks, although our experience is that the benefits are greatest when it is used during initial development. Feature typing aided the planning of new features for both the Java Pet Store and the network simulator. The classifications also demonstrated the level of integration that features have in their systems. For example, the implementation of “localized” messages in the Java Pet Store was like that of an H/VF, which made it difficult for us to add Spanish localization as an unplanned feature.

The wireless network simulator was more successful in demonstrating the different feature types and how to integrate them. We were able to show each feature type and how each represents a different level of integration with the rest of the program. By employing feature typing during design, we were able to create an effective project implementation plan. Outside the core product, each new feature was coded as an AspectJ aspect in a straightforward way. This simplifies the inclusion and exclusion of features, thus making an effective product line. As new features were added, the core program had to be changed (as planned) to accommodate an AspectJ implementation. (This practice was observed by Murphy et al. in their analysis of program development with AOP [10].) For instance, we had to change the basic thread loop to codify the *Stop/Resume* feature. Our plan also accounted for a redesign in the mote’s sampling methodology to detect and sample new resources automatically while implementing *Site Precipitation*. Finally, the simulator study showed that the benefits of feature typing are maximized when one can pair this early aspects technique—which helps to modu-

larize crosscutting design and deployment concerns—with aspect-oriented implementation technology.

## 5. Related Work

Feature typing builds on the concepts found in *Generative Programming* [4]. In describing domain engineering, the authors of that book use “vertical” and “horizontal” to describe scopes and domains. We overload these terms to describe the forms of features in a product line. Generative programming techniques are complementary to feature typing and can be informed by a feature type analysis.

Many researchers have investigated techniques for software product lines [1, 3, 5] and features. Our feature typing model provides additional insight for (required and optional) feature integration within product lines. Feature typing does not conflict with feature-oriented programming (FOP) [9]; instead, it helps by providing information to guide the implementation and deployment of features over time. In general, feature modeling categorizes each feature based on domain, and then associates each feature based on dependence [8]. The feature typing model is not domain-specific, but uses Lee’s work to define the interdependence within a product’s feature set.

Jacobson et al. have researched aspect-oriented software development use cases (AODUC) [6, 11]. Our feature typing model is complementary to AODUC: AODUC expresses how to represent the aspects from a development perspective, whereas feature typing provides more detail on the features themselves. This empowers the developer to apply features in different phases of the development, consistent with project design.

Finally, our work is related to aspect-oriented programming [7]. Feature typing greatly benefits when AOP is used to augment core functionality with new feature options.

## 6. Conclusion

Feature typing is an emerging method that identifies the general natures of software features. In the model, features have two effective natures: functional or *vertical* (what they do) and characteristic or *horizontal* (what they have). During the project analysis and design phases, a designer can use feature typing to better organize the required and optional features and thus better divide program development into workable phases. Feature typing illuminates potential feature interaction, alleviates issues such as time-line uncertainty in program development, and gains benefits from aspect-oriented programming.

In this paper, we presented the feature typing model and how it can help in the design and planning steps of the product life cycle. We applied the model to two programs, the

Java Pet Store (J2EE) and a simulator (AspectJ). We identified and classified features for each. We enhanced the simulator with examples of each feature type, but we encountered obstacles with the J2EE program. From this we learned that the model works well and independently from programming models, but in practice, it is best suited for aspect-oriented implementations. We learned that certain feature types, such as purely horizontal features, require core product adaptation. Finally, we learned that retroactively applying the model may not be as fruitful as starting with a clean design.

## Acknowledgments

We thank the anonymous reviewers of this paper for their many comments. Their reviews helped us to greatly improve our presentation and discussion of feature typing.

## References

- [1] D. Batory, R. E. Lopez-Herrejon, and J.-P. Martin. Generating product-lines of product-families. In *ASE*, 2002.
- [2] L. Chung, B. A. Nixon, and E. Yu. *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers, 2000.
- [3] P. Clements and L. Northrop. *Software Product Lines*. Addison-Wesley, 2002.
- [4] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
- [5] D. Dikel, D. Kane, S. Ornburn, W. Loftus, and J. Wilson. Applying software product line architecture. *IEEE Computer*, 1997.
- [6] I. Jacobson and P.-W. Ng. *Aspect-Oriented Software Development with Use Cases*. Addison-Wesley, 2005.
- [7] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP ’97: Object-Oriented Programming*, volume 1241 of *LNCS*, 1997.
- [8] K. Lee, K. C. Kang, and J. Lee. Concepts and guidelines of feature modeling for product line software engineering. In *Software Reuse: Methods, Techniques, and Tools: Proceedings of the Seventh Reuse Conference (ICSR 7)*, volume 2319 of *LNCS*, 2002.
- [9] M. Mezini and K. Ostermann. Variability management with feature-oriented programming and aspects. In *ACM SIGSOFT*, 2004.
- [10] G. C. Murphy, R. J. Walker, E. L. A. Baniassad, M. P. Robillard, A. Lai, and M. A. Kersten. Does aspect-oriented programming work? *Communications of the ACM*, 2001.
- [11] D. Stein, S. Hanenburg, and R. Unland. A UML-based aspect-oriented design notation for AspectJ. In *AOSD*, 2002.
- [12] Sun Microsystems, Inc. The Java Pet Store, 2005. <http://java.sun.com/developer/releases/petstore/>.