# Binding-Time Flexibility for Managing Variability (Extended Abstract)

Venkat Chakravarthy and Eric Eide
University of Utah, School of Computing
{vchakra,eeide}@cs.utah.edu

## 1. INTRODUCTION

Product-line software architecture highlights the need to manage software variants in an efficient yet systematic manner. Techniques such as feature-oriented programming and component-based software engineering exist to help manage variability at various stages in the software development cycle.

This paper summarizes a technique that we are currently developing, based on the notion of *binding-time flexibility*, for modularizing and implementing variations within a software product line. The ability to change the binding times of features within a product line can be important for achieving both technical and business goals. For example, it can enable a single product line to address a wide range of platform domains (e.g., from embedded devices to desktops) or market points (from "bare bones" to "professional"). Our technique is designed to enable this level of variation while maintaining important software engineering properties such as traceability between design and implementation, modularity, obviousness, and stability under product evolution.

The parts of a system that facilitate changes to the system's characteristics are called *variation points*. *Binding time* refers to the time at which the decisions for a variation point are set [2]. There are different types of binding times available to the designer — e.g., compile time, link time, load time, and run time — and the choice of a binding time often depends on the application domain being targeted. For instance, embedded systems are often constrained by small memories and real-time deadlines that are not present in the desktop and server system domains. In general, earlier binding times enable better static analysis and system optimization, whereas later binding times enable configuration by users and post-deployment updates. Early and late bindings are therefore appropriate for different domains. In order for a single product line to span multiple domains, the binding times of certain features within the product line must be flexible.

The software engineering technique that we propose allows a programmer to implement variation points that support binding-time flexibility in a modular, traceable, and straightforward fashion. In short, our approach can change the binding time of a feature from being a design-time attribute of a product line to being an assembly-time attribute of a product. The key idea is to describe and encapsulate a variation point within the implementation of a design pattern; the pattern serves to make the point identifiable and stable, and it provides the "hooks" that are necessary for changing the point's binding time. At the time of product assembly (i.e., when the parts of a product are selected, collected, and configured to work together), the assembler decides whether feature bindings are fixed or deferred to a later time. With this ability, software architects can use a single code base to serve different domains, leading to efficient management of variability in a product line.

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

To evaluate our technique, we are applying it to an existing middleware system called RTZen [6]. RTZen has been implemented, to a great extent, in a modular fashion using design patterns. We exploited the structure derived from the patterns and used an aspect-oriented programming language, AspectJ, to identify and isolate strategic join points that correspond to product-line variation points. These join points served to control binding-time decisions and allowed us to create a small product line. In some products, certain variation points are "hard-wired" as appropriate for an embedded system, and in other products, feature selection is left "open" as appropriate for a general-purpose desktop product.

Overall, our experience is that our technique is effective: the implementation of binding-time flexibility is modular and easily controlled, and therefore it is a good way to manage certain types of product variability. However, we have also found that achieving certain product qualities, such as compact and efficient code for embedded systems, can be challenging. The rest of this paper describes our technique and experiment in more detail.

## 2. APPROACH

Design patterns are extremely useful since they introduce modularity, reusability, and flexibility in the structure of software. We have leveraged the inherent stability provided by design patterns and the encapsulation of behavior provided by aspects to provide binding-time variability at assembly time. We have generalized a series of steps to achieve this in a systematic manner.

**Step 1: Identify and characterize the variation points.** In the initial stages of the software development cycle, a feature model helps to define software behavior and guide the discovery of variation points. Once variation points are identified, a designer must analyze the behaviors of those points. If applicable, for each point, the designer chooses one or more design patterns to capture the behavior. The selected patterns clearly characterize the behavior by identifying the participants and their roles and interconnections.

**Step 2: Express domain concerns using binding times.** Variability arises from the need to address different domain concerns. Design patterns allow a product-line architect to address these concerns using binding times. The architect evaluates the binding times of the pattern participants in light of the domain(s) the software is being created for. Certain domains impose restrictions, for example, on the number of participants in the pattern or on the roles the participants can play. These restrictions guide the choice between compile-time and run-time binding. The same restrictions may not apply in other domains, however. The combination of varying domain concerns, participants, and roles highlights the need to encapsulate and manipulate variability in binding times.

As an example, consider an embedded system having a temperature sensor and a clock. The LCD display updates the temperature and time display whenever there is a change in either the sensor reading or clock tick. This is example of the *Observer* pattern.

Domain constraints tell us that there can only be two subjects (the temperature sensor and clock) and one observer (the display). We can therefore use compile-time binding to configure these static decisions about the participants in the *Observer* pattern. If, however, the domain were different and capable of supporting multiple subjects and observers, run-time binding could be used in the configuration. This high-level analysis of the design pattern enables us to first, locate a binding-time variation point and second, manipulate the type of binding to address the domain constraints.

**Step 3: Implement the pattern.** Design patterns can be implemented in many ways. The traditional approach outlined by Gamma et al. [3] implements the pattern within the participants' code. An alternate approach is described by Hannemann et al. [4]: their technique encapsulates the roles of the participants and abstracts the pattern from its application. Our technique for managing variability is compatible with either implementation approach.

**Step 4: Introduce binding-time variability using aspects.** We suggest that as an initial step, the design pattern implementation support only run-time binding. This helps to clarify and direct the process of introducing binding-time variability. Based on the knowledge gained in step 2, a programmer writes pointcuts to intercept the run-time binding decisions. He or she then writes advice to lift the binding-time decisions from run time to compile time. This can be achieved in a variety of ways. Weaving in relevant aspects as compile-time options provides domain-specific customization. Advice can be written so that, if no aspect is woven in at compile time, the code for run-time binding stays in effect. Alternatively, run-time binding itself can be woven in as advice. In the example from step 2, the code for binding subjects and observers can be aspects that provide either compile-time or run-time binding.

**Step 5: Evaluate the implementation.** Aspects and advice can be implemented in multiple ways, and it is a challenge to find an implementation that satisfies the twin goals of good design and optimal code size and speed. For embedded products, we aspire to achieve the benefits of code inlining and devirtualization through compile-time bindings and optimizing compilation [1]. In more general terms, a system implementer must choose a binding-time implementation strategy in light of a product line's overall goals.

# 3. CASE STUDY

To test our proposed technique we applied it to RTZen, which is an RTSJ implementation of a real-time CORBA Object Request Broker (ORB). We focused on the Portable Object Adapter (POA) subsystem of RTZen. The POA directs requests (method invocations) on CORBA objects to the code that services requests, in a portable manner between different ORB products. The POA is configured using policies that describe aspects of the POA's behavior, and each of these configuration points corresponds to a variation point in our feature model. These are the points where we would like to support variable binding times. In RTZen, each of these points is encapsulated by an instance of the *Strategy* design pattern.

We concentrated on the Thread policy of the POA: this policy determines the POA's support for multiple threads within a multithreaded ORB implementation. The POA supports two threading models, the *Single Thread Model (STM)* and the *ORB Control Model (OCM)*. In RTZen, the Thread policy is implemented as an abstract class (TPStrategy). The STM (STMStrategy) and OCM (OCMStrategy) are implemented as concrete classes extending the abstract class. Run-time binding is implemented by a static method in the abstract class that accepts configuration parameters. This static method originally contained an *if* construct. Constructors of either concrete strategy were called based the run-time values of the parameters to the *if* construct in the static method.

To support variable binding time, in one implementation, we removed the *if* construct from the static method and wrote two aspects called STMAspect and OCMAspect. These aspects, implemented in AspectJ, weave the required constructor calls and initialization parameters into the static method. The aspects allow the threading policy to be set when a product is assembled: this is achieved by compiling the abstract class (TPStrategy), the chosen concrete class (say STMStrategy), and its associated aspect (STMAspect). Weaving in the original *if* construct as an aspect (at compile-time) provides run-time policy binding. In this manner, we can lift the run-time binding decision to an earlier time.

As a final step, we implemented this idea in three different ways and evaluated the implementations. Complete results and our experiences with the different implementations will be provided in a longer version of this paper. In summary, this example demonstrates the utility of design patterns to identify join points and the power of aspects in introducing binding-time variability.

# 4. RELATED WORK AND CONCLUSION

Our approach for managing variability is related to work in a number of areas. For example, Dolstra et al. previously suggested that variation points should support multiple binding times, a notion they call *timeline variability* [2]. They describe the difficulty of implementing timeline variability in a conventional language — their examples are in C — and suggest partial evaluation as a solution. Our technique overcomes difficulties in a more specific and programmer-visible manner through a combination of object-oriented design patterns and aspect-oriented programming. Schultz et al. explored the impact of partial specialization on design patterns and showed that run-time speed can often be improved [5]. Our technique seeks to obtain similar gains in a more visible and predictable fashion: rather than rely on a specializer to prove properties of interest, we rely on product assemblers to assert properties they desire. Our work on middleware product lines is complementary to that of Zhang et al., who propose a *post-postulated architecture* [7] for customizable middleware. They use aspects to modularize features within a middleware system; we focus on controlling the times at which features can be reassembled into products.

In summary, we have described a technique for implementing variation points that support flexible binding times. Our technique helps to manage variability within product lines and preserves traceability between design and code. We are continuing to evolve our approach, and in particular, address the challenges of applying our technique to embedded system product lines.

# 5. REFERENCES


[1] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Optimising AspectJ. In *PLDI '05*, pages 117–128, June 2005.

[2] E. Dolstra, G. Florijn, and E. Visser. Timeline variability: The variability of binding time of variation points. In *Proc. of the Workshop on Software Variability Management (SVM '03)*, pages 119–122, Gronigen, The Netherlands, Feb. 2003.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley.

[4] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *OOPSLA '02*, pages 161–173, Nov. 2002.

[5] U. P. Schultz, J. L. Lawall, and C. Consel. Specialization patterns. In *ASE '00*, pages 197–206, Sept. 2000.

[6] UCI DOC Group. RTZen. http://doc.ece.uci.edu/rtzen/.

[7] C. Zhang, D. Gao, and H.-A. Jacobsen. Towards just-in-time middleware architectures. In *AOSD '05*, pages 63–74, Mar. 2005.