Chapter 5
Variability Implementation

Jan Bosch and Rafael Capilla

What you will learn in this chapter
- *Mechanisms to implement software variability*

1 Introduction

Software variability is modeled, reasoned about, and discussed in many organizations, but at some point, it needs to be realized in the software of a system or product line. The subject of this chapter is to discuss the realization of variability in a software system or software product line.

The realization of a variation point can be achieved by a variety of technologies and approaches. Selecting the optimal approach is driven by two factors. The first is the abstraction level at which the variation point is explored, ranging from the architecture to the code level. The second is the stage in the life cycle at which the variation point is bound, whether the binding is permanent as well as the stages during which variants can be added to the variation point.

Choosing the right realization mechanism is of significant importance for two reasons [1]. The first is that it often is difficult to change the selected mechanism once it has been chosen. The reason for this is that variants are written to operate with a specific mechanism. In addition, frequently, variants are written by other organizational units or even other organizations altogether, as in the case of software ecosystems [2], which complicates changing the selected mechanism.

J. Bosch (✉)
Chalmers University of Technology, Gothenburg, Sweden
e-mail: jan@janbosch.com

R. Capilla
Rey Juan Carlos University, Móstoles, Madrid, Spain
e-mail: rafael.capilla@urjc.es

The second reason is that over time, many variation points tend to be bound at later and later times in the software development life cycle. A rigid realization mechanism that complicates this process will cause tension in the organization and inefficiencies in development.

Consequently, it is important to focus attention on variability realization. The remainder of this chapter is organized as follows. The next section provides a conceptual context of software variability management using the software life cycle by discussing the software variability realization implications in the different stages. The subsequent section discusses the abstraction levels at which variability can be captured. This is followed by the main part of the chapter where we present the different variability realization mechanisms. The chapter is closed by a discussion of relative advantages and disadvantages of different mechanisms and a conclusion.

2 Introducing, Selecting, and Binding Variants

Software variability can be discussed at several levels of abstraction, but at some point it needs to be implemented in the software system. For this, we need to have a good understanding of the software variability life cycle. This life cycle is obviously related to the overall software development life cycle. Although one can have different perspectives on the software development life cycle, in this chapter we consider the following stages:

- *Requirement specification*. During this stage, the team aims to maximize the clarity of what is to be built. There may be explicit requirements for variability, but equally often decisions are taken as part of the requirement specification process that reduces the required variability.
- *Architecture design*. The top-level breakdown of the system into its main components is the stage where the first variation points can be, and often are, introduced.
- *Detailed design*. Once the overall breakdown of the system is agreed and in place, the focus can shift to the design of the individual components. At this level, additional variation points can be introduced.
- *Software development*. Especially more narrowly defined variation points in the system are implemented using code-level variation points.
- *Compilation*. The compilation stage is often where the first variation points are bound to variants.
- *Linking*. During linking, especially higher-level variation points are often bound to specific variants. Most bindings during compilation and linking are permanent and cannot be changed in later stages.
- *Installation/configuration*. Assuming the software system is installed and configured at the customer, binding of variation points takes place during installation in response to settings selected by the customer installing the product.

- *Start-up*. During system start-up, several variation points can be bound to variants. Often, configuration files are used that are read during system start-up to bind certain variants to the remaining variation points.
- *Run-time*. Finally, the variation points that are not permanently bound in earlier stages can be bound and rebound during run-time. During installation and especially start-up and run-time, the binding of variants to variation points is often not permanent and can be rebound during at run-time.

During the software life cycle, a variation point evolves through a number of phases. The first is the *introduction* of the variation point at a specific stage in the life cycle. Frequently, this is in the earlier stages, but there are techniques that allow for the late introduction of variation points in the system. The second stage is the *addition of one or more variants* to the variation point. These variants capture the differences in behavior that are required from the system. The third stage is the *binding of a variant* to the variation point. At this point in the life cycle, the variant bound to the variation point can still be rebound. The final stage, though not reached by all variation points, is the *permanent binding* of the variant to the variation point. A variation point is bound permanently in a life cycle phase if in all subsequent phases it cannot be rebound to a different variant. At this point, the variation point, for all purposes, has been removed from the system at that phase in the life cycle.

One aspect of variation points is them being open or closed. At a certain phase in the software development life cycle, if variants can be added to a variation point, it is considered to be *open*. Many variation points will, in a later phase, become closed, meaning that the set of available variants can no longer be extended. This is largely orthogonal to the binding of a variant to a variation point. For instance, in an internet browser, a codec variation point can be bound to a particular variant, but the user can still add new codecs (variants) to the browser.

The coding effort for implementing binding times of features to support dynamic changes (e.g., system features that can be activated dynamically) can be reduced if we adopt flexible approaches like the one described in [3], where code-level idioms based on aspect-oriented languages can be used to avoid duplicate code for static and dynamic binding and enhance maintainability as well.

There are more complicated cases that we will not discuss in this chapter, including the reduction of the set of variants during progressive stages in the life cycle due to constraining dependencies as well as cases where variation points are permanently bound because of dependencies on other variation points and variants where their selection limits the set of alternatives to one. As discussed in earlier chapters, variation points and variants have dependencies on other variation points and variants. As the designer or customer configures the system, choosing a variant for one variation point will limit the set of possible variants for other variation points. Occasionally, this can lead to situations where a variation point has no remaining variants (e.g., the variability included in dead code will have no effect on the selection and realization of those variants). This, however, does not necessarily lead to an illegal configuration as the system configuration may not need the functionality provided at the variation point.

3 Variability Abstraction Levels

Depending on the size of the functionality that is to be variable, different variability abstraction levels can be identified at which reasoning about and realization of software variability can take place. We identify the following three levels:

- *Architecture*. At the architecture level, the primary mechanism for variability is the replacement of top-level components with other implementations of these components or the binding of optional components depending on the context in which the system is deployed.
- *Component*. At the component level, variability is often more pervasive and complex and often this is the main level at which variability is modeled. This is more concerned with extension points, superimposition[1] of code, wrapping, and other mechanisms that adjust the behavior of components.
- *Code*. At the code level, there is a large set of variability mechanisms available. The main concern, however, is that the code-level mechanisms can be applied for normal algorithmic implementation as well as for managing variation points.

Appreciating the differences between variation points at different levels of abstraction is quite important as each level brings its own advantages and disadvantages. Selecting the right level should be driven by the variability that is specified in the requirement specification, the expected evolution of the variation point, and the binding time of the variant to the variation point. In addition, specific trace mechanisms should be defined to track the changes from one abstraction level to another and vice versa, as managing the variations in one level (e.g., the variability defined in the architecture does not mandate how this will be implemented) is radically different from another level (e.g., different implementation mechanism can be used for coding variability at the code level) and the modification of the structural variability (i.e., the variability defined in a feature model representing the variants and variation points to describe system features) impacts the lower levels or configurations files supporting allowed options.

4 Variability Realization Mechanisms

There are several techniques to implement the variability which is described in feature models and each of these techniques is used in different stages of the life cycle and is driven by the time when variants will be bounded (i.e., variability realization). Basic variability enabling mechanisms are described in 1, 4, 5, such as inheritance, parameterization, conditional compilation directives, dynamic

[1] Superimposition of code is a black box component adaption technique that allows one to impose predefined but configurable types of functionality on a reusable component. Using superimposition, additional behavior is wrapped around existing behavior.

libraries, etc., but all these ways to implement variability in code are sometimes driven or limited by the language, framework, or technology used.

To provide a perspective driven by software variability management needs, we focus the discussion of variability realization mechanisms based on an earlier work by one of the authors [6]. In Table 5.1 below, we present an overview of techniques at different levels of abstraction and with binding times in different stages of the software development life cycle.

4.1 Binary Component Replacement

Intent. The intent of binary component replacement during linking is to permanently bind a specific component implementation. This allows the system to be bound to specific components needed for a particular configuration of the overall system. “Replacement” refers to a binary component that is specifically added for a concrete product or configuration instance.

Solution. The binding to binary libraries can be done at compilation and linking times prior to deployment. If linking is realized at run-time, the variability must manage this binding internally to the system assuming all libraries are available.

Example. Dynamic libraries such as Apache modules can be uploaded and bound at run-time when needed, whereas Linux kernel modules are linked before deployment when the kernel is recompiled.

Implications. This variability realization technique is easy to manage and to implement with few consequences to the system, as security is an aspect well covered in this case. By contrary, the unavailability of run-time libraries or incompatibility problems with existing version may cause severe problems.

4.2 Binary Component Selection

Intent. The intent of binary component selection is similar to selecting one component among a set of existing alternatives, and the binding time for selecting a component goes from installation to post-deployment time.

Solution. Dynamic components, libraries, and files are selected and bound among several. The alternatives can be bound more statically at installation time while they become more dynamic from start-up to post-deployment time, and variability is often realized externally to the binaries.

Example. Like in the previous case, any dynamic library or configuration file aimed to update the current system configuration or functionality fits under this category. In this case, the variability is managed externally to the component but some variants or system features can be defined in specific configuration files that can be uploaded dynamically.

Table 5.1 Variability realization mechanisms

Abstraction level	Binding time				
	Compilation	Linking	Installation/configuration	Start-up	Run-time
Architecture	N/A	Binary component replacement	Binary component selection	Binary component selection	Binary component selection
Component	Variant component specialization Optional component selection Code fragment superimposition	Optional component selection Code fragment superimposition	Optional component selection Variant component implement.	Optional component selection Variant component implement.	Optional component selection Code fragment superimposition Run-time variant component specialization Variant component implementation
Code	Condition on constant Code fragment superimposition	N/A	N/A	Condition on variable	Condition on variable

Implications. The implications are similar like in the previous case, but incompatibility problems of system features in system configuration or binary files may arise if these have not been pre-checked before. For instance, an older version of binary file is selected and such instance is incompatible with the existing version of the system or application running.

4.3 Variant Component Specialization

Intent. The intent of variant component specialization is to adjust a component implementation to the product architecture when the provided interfaces of a component implementation representing a variant feature vary. Specialization assumes a context-specific extension that is then developed for an individual product/configuration instance.

Solution. Separating the interfacing parts into different classes facilitates the interaction between components as we can decide what variant of the interfaced component to include in the product architecture. The variability in this case is bound externally but variants are realized at system design.

Example. A software using an enhanced security detection mechanism is only used in certain cases under a set of predefined conditions.

Implications. Several implementations must coexist that can be selected dynamically, sometimes at start-up time or at run-time.

4.4 Optional Component Selection

Intent. The intent of optional component selection is to include or exclude a particular component implementation, often selected from a set of existing alternatives.

Solution. System features are included or excluded as we separate the optional behavior in a different class or component. The binding time for an optional functionality goes from compilation to post-deployment time, as system features can be added or modified at any time. Binding is done externally by configuration management tools or by the compiler.

Example. A smart home system that adds or removes optional functionality for different customers and at a different cost (e.g., the system can use different security access methods). A basic package configured at compilation/linking time can be modified later by, for instance, adding a new module at configuration time.

Implications. Decoupling optional behavior is not always easy and depends on how the structural variability is defined and implemented in the system and the dependencies among the variants.

4.5 Code Fragment Superimposition

Intent. The intent of code fragment superimposition is to impose predefined types of functionality on a reusable component without directly affecting the source code.

Solution. With this solution, we superimpose product-specific behavior and concern's additional behavior is wrapped around existing behavior. In this case, the binding is realized externally and variability is bound at compilation or linking time, but run-time superimposition is also possible.

Example. Any crosscutting functionality (e.g., aspects) introduced in the system functionality constitutes an example of superimposition (e.g., different authentication methods based on internal or external authentication systems and the user or the system itself can select among one of these). At run-time, the Eclipse platform offers a way to dynamically add or remove plug-ins that include new functionality to the main platform.

Implications. Positively, superimposition enables that different concerns are separated from the main functionality. However, understandability on how the final code works becomes harder.

4.6 Run-Time Variant Component Specialization

Intent. It supports the selection between different specializations inside a component implementation during run-time, as different requirements may demand such capability.

Solution. The component implementation must provide a number of alternative executions that can be switched at run-time. Different design patterns (e.g., strategy, template method, or abstract factory) can be used to separate behavior into several classes and use inheritance or polymorphism to implement the required variability. In this case, the functionality for binding is internal.

Example. The case of a smart home system which provides sensors to detect several data, such as temperature, humidity, smoke, or people. The fire detection system can be activated at run-time to detect fire, as this is required to activate both the home smoke detector and temperature sensors. Different classes provide such functionality that is used by the smart home system control to activate the right sensors in case of the presence of smoke and high temperature.

Implications. Some common functionality might be duplicated when the variants must select between different specializations.

4.7 Variant Component Implementation

Intent. The intent of variant component implementation is to support several implementations of one component architecture that can be chosen at any time dynamically.

Solution. Several design patterns (e.g., strategy pattern, broker pattern, SOA service-broker pattern, etc.) can be used to select between one or several components with high flexibility and changeability. Variability is defined at design time and variants cannot be added later. Variability is bound internally to the system.

Example. Several e-mail protocols like POP and IMAP using the same interface for connecting to the e-mail server.

Implications. The reusability of some code pieces may be low.

4.8 Condition on Constant

Intent. The intent of condition of constant is to support a way to enact one operation from several available. It constitutes a refined version of variant component specialization and is often used to select between different compilation options.

Solution. Conditional #ifdef compilation directives can be used to implement the variability at compilation time. The collection of variants depends on constants that are used to bind the variants at compilation time.

Example. Any software package that uses compilation directives that are selected before the package is installed in the system. Also, configuration executable files are often used to determine the system environment and to drive the selection of the compilation values.

Implications. Using #ifdef directives can be risky and difficult to maintain, in particular when the installation of a software package involves additional packages or modules, as the number of interdependencies may grow exponentially across releases (e.g., the Linux kernel). Also, flexibility of the variability implemented using this option decreases as the number of links and potential paths grow. Moreover, variation points tend to be scattered as it becomes difficult to track what parts of the system are affected by one variant.

4.9 Condition on Variable

Intent. The intent of condition on variable is to support several ways to perform an operation but the choice can be rebound at run-time.

Solution. It replaces the condition on constant by a variable that changes its value dynamically. In this particular case, new variants can be added during implementation and variability is bounded internally.

Example. Any program that wants to control the execution flow can use this technique. Another example may refer to the selection of different web services at run-time according to certain conditions that are stored in variables (i.e., variants in the system) which determine the selection of a particular web service.

Implications. This is a very flexible technique where variants can be instantiated dynamically. However, tracking the value of the variation points can be sometimes difficult if variation points are spread throughout the code.

5 Selecting a Realization Mechanism

This chapter summarizes different variability implementation techniques from a high-level point of view as different languages (e.g., object oriented versus nonobject oriented) and design patterns can be used to implement each technique. Hence, we did not restrict our description to a particular implementation technology. Object-oriented classes, inheritance, variables supporting system features, dynamic libraries, and so on, are examples of different ways to implement the system variability, but selection of a mechanism is driven by the binding time at which the variants are bound.

In general, multiple binding times are hard to combine, so we need to select carefully which binding times we want to support in order to choose the right variability implementation techniques that can be mixed in the code or supported by a specific platform.

The selection of a preferred realization technique is driven by three factors: the mapping to the problem domain variability, the need for late-stage openness, and the expected system evolution.

Ideally, there is a direct, one-to-one mapping between a problem domain variation and a variation point in the solution domain. This significantly simplifies the configuration process and it avoids complex defect detection and repair situations. For instance, in a case where a problem domain variation is mapped to #ifdef statements in every module of the system, it does not require much to make a mistake in one module and have the resulting system act in unpredictable ways due to misconfiguration. Deciding the variability realization technique needs one-to-one map to the problem domain variation.

Second, depending on the system domain, there may be a significant need for late-stage openness of the variation point to allow adding new variants. The selection of the realization technique should explicitly consider the ability to add variants at the required time as many realization techniques cause permanent binding during the compilation and linking stage.

Finally, expected system evolution is an important factor in the selection of the variability realization technique. In practice, the binding time of variation points tends to be delayed to later stages in the life cycle, meaning that even though a variation point may be bound permanently at compile time at this point in time, it is not unreasonable to assume that over time the binding will take place at installation, start-up, or run-time. Especially for variation points that have system-wide implications, the cost of replacing the selected variability realization technique may be very high and, consequently, it may be better to select a technique that allows for late binding.

6 Outlook

Writing adaptable and evolvable software using variability techniques is not always easy, as the modeling of large variability models is a complex and tedious task in itself. Because customers today push software developers to provide more and more configurable options, the *external variability* becomes more important, and this fact drives the realization of the variability times closer to configuration, run-time, and post-deployment times.

Systems that require run-time binding must implement the dynamic binding condition and use dynamic variability implementation mechanisms in a controlled manner to make the software more adaptable. However, only few variability implementation techniques can be used to realize binding and rebinding during execution time. Regarding the binding time of the variability realization mechanisms, one could think in a post-deployment realization mechanism, suitable for those systems that realize their variants once deployed. However, this new binding is quite similar to the run-time mechanism, and the slight difference between run-time and post-deployment perceived today is more subjective by software engineers because the variability realization mechanisms for architecture, component, and code are almost the same.

Finally, open variability models allow variants to be changed dynamically, but such high evolvability of the structural variability is hard to implement and requires additional codification to support the extensibility of the variability model.

References

1. Fritsch, C., Lehn, A., Strohm, T., Bosch, R.: Evaluating variability implementation mechanisms. In: Proceedings of International Workshop on Product Line Engineering (PLEES), pp. 59–64 (2002)
2. Bosch, J.: From software product lines to software ecosystems. In: Proceedings of the 13th International Software Product Line Conference (SPLC 2009), August 2009
3. Andrade, R., Ribeiro, M., Gasiunas, V., Sabatin, L., Rebêlo, H., Borba, P.: Assessing idioms for implementing features with flexible binding times. In: CSMR 2011, pp. 231–240 (2011)

4. Amin, F., Mahmood, A.K., Oxley, A.: An analysis of object oriented variability implementation mechanisms. ACM SIGSOFT Softw. Eng. Notes **36**(1), 1–4 (2011)
5. Myllymäki, T.: Variability management in software product lines. Tampere University of Technology. Software Systems Laboratory, ARCHIMEDES (2001)
6. Svahnberg, M., van Gurp, J., Bosch, J.: A taxonomy of variability realization techniques. Softw. Pract. Exp. **35**(8), 705–754 (2005)