Chapter 9
Feature Interactions

After reading the chapter, you should be able to

- understand the role of feature interactions in feature-oriented product lines, including intended and inadvertent interactions,
- identify reasons for feature interactions and the feature-interaction problem,
- characterize the nature of 2-way and higher-order interactions,
- outline techniques to detect feature interactions,
- select suitable solutions to implement coordination code for known feature interactions, and
- weigh their mutual strengths and weaknesses.

After a broad discussion of a diverse selection of techniques for implementing features in Part II, we now have a closer look at how features interact when combined with other features. The key idea of feature orientation is to make features explicit in design and code, either by annotating code belonging to a certain feature or by separating and modularizing feature code. But a feature is not an island. Features interact in various ways, both in positive and intended ways, as well as in critical and inadvertent ways. Features are often expected to interact: to exchange information, refine the behavior of other features, reuse the functionality of other features, and accomplish a task in cooperation. However, inadvertent interactions can cause unexpected erroneous behaviors and result in undesired and critical system states. Specifying and managing intended feature interactions as well as detecting and resolving unintended feature interactions is one of the key challenges of feature-oriented product-line development.

In this chapter, we take a closer look at interactions between features and how they manifest in program code and behavior, rather than at the features themselves and their implementations. We illustrate different kinds of feature interactions, discuss strategies to detect them, raise awareness of an instance of the feature-interaction problem, called optional-feature problem, and compare techniques to implement known feature interactions in a controlled manner.

S. Apel et al., *Feature-Oriented Software Product Lines*,
DOI: 10.1007/978-3-642-37521-7_9, © Springer-Verlag Berlin Heidelberg 2013

9.1 The Feature-Interaction Problem

A feature that works perfectly well in a given system may exhibit inadvertent behavior when combined with other features. The problem is that, when features are developed independently, it is difficult to predict their mutual interactions when combined. Typically, the behavior of the generated product that contains multiple independently developed features is not easily deducible from understanding the features in isolation—we have to identify and understand their interactions.

Definition 9.1 A *feature interaction* between two or more features is an emergent behavior that cannot be easily deduced from the behaviors associated with the individual features involved.

An *inadvertent feature interaction* occurs when a feature influences the behavior of another feature in an unexpected way (for example, regarding the expected control flow, program or data state, or visible behavior).

The *feature-interaction problem* is to detect, manage, and resolve (inadvertent) feature interactions among features. □

When features are combined, their interactions need to be coordinated, for example, by ordering their execution, synchronizing data access, defining precedence rules for action handling, and including missing behavior.

For illustration, we provide a list of examples of feature interactions and the corresponding problems:

Example 9.1 *Call forwarding and call waiting.* A canonical example of a feature interaction occurs in telecommunication networks, in which the two features CallForwarding and CallWaiting interact (Calder et al. 2003). CallForwarding forwards calls made to a busy line to another host. CallWaiting notifies the called party on a busy line of another incoming call and allows the user to switch between both calls. Both features work fine in isolation, but it is unspecified what happens with an incoming call on a busy line if both features are activated. Either feature could take precedence over the other or, even worse, both may attempt to act at the same time.
The interaction between CallForwarding and CallWaiting is undesired and can lead to race conditions and unexpected and inconsistent behavior. The interaction can be hard to predict, because it occurs only in specific conditions (a second call on a busy line). If the interaction is known, we can take measures to control it by giving explicit precedence to one feature (possibly even configurable by the user) or by making both features mutually exclusive (only one can be selected at a time). □

Example 9.2 *Fire and flood control.* In a building-automation system, as outlined by Kang et al. (2002), feature FireControl activates sprinklers when sensors detect a fire, and feature FloodControl cuts off water supply when water is detected on the floor. Individually, both features operate as desired, but they *interact* in inadvertent

and critical ways: When fire is detected, feature FireControl activates sprinklers; subsequently, feature FloodControl detects standing water, and turns off the water main; as a consequence, the building burns down.
Clearly, the interaction between FireControl and FloodControl is undesired, and inadvertent in the sense that it is hard to predict when planning and implementing the involved features independently. Only when the interaction is known, we can take corresponding steps to manage or resolve it, in this case, for example, by giving explicit priority to feature FireControl over feature FloodControl, controled by some coordination code. □

Example 9.3 Read-only data structures and indexes. As a further example, suppose we incrementally develop a simple data-management solution by starting with a simple read-only data structure and by extending it with two optional features Write and Index. Feature Write adds functionality to add, change, and remove data. Independently, feature Index is developed on top of the read-only data structure to speed-up data retrieval (say, by storing an index as a separate hash map, which is created at load time). Without the other, both features work well on top of the basic data structure; but when combined, changes in the data due to feature Write are not reflected in the index maintained by feature Index. As a consequence, the index can become inconsistent with the data structure such that queries return incorrect results.
Clearly, some coordinating behavior is missing when combining two features. Feature Write does not know about the index that needs to be updated, and feature Index is not aware that the data structure can be modified. When we understand their interaction, we can implement additional coordination code, such that the index is updated properly when the data are modified. □

Example 9.4 Database transactions and statistics. Similar to the previous example, the features Transactions and Statistics interact in a database system. Transactions ensures ACID properties (atomicity, consistency, isolation, durability) in the case of concurrent access to persistent data, and defines the granularity of recovery actions. Statistics collects information for tuning and optimizing data management (for example, number of tables).
Transactions and Statistics interact. On the one hand, Statistics collects information on transaction operations (for example, the number of transactions per second is measured and stored for self-tuning). On the other hand, feature Transactions provides transactional access to statistics data: we want to access data collected by feature Statistics under the umbrella of transactional control.
If we develop both features independently, Statistics would not know about transactions and could not collect statistics on them. Conversely, Transactions would not know about statistics and could not control access to statistics data. Only when we know about this interaction, we can implement corresponding coordination code to make them work correctly together. □

These examples show the breadth of possible feature interactions. Some feature interactions are undesired and inadvertent in the sense that they are hard to predict when planning and implementing features in isolation (see Example 9.2). Other

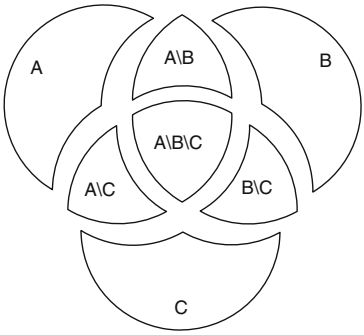

Fig. 9.1 Visualization of three features, three 2-way interactions and one 3-way interactions

feature interactions are desired and planned in advance (see Example 9.4). In any case, features need to be coordinated (more or less explicitly), and that may require additional coordination code, which we discuss in Sect. 9.3 in more depth. That also means, feature interactions may harm feature modularity (see Sect. 3.2.4, p. 57), because, besides the fact that each feature has its own code, there is additional code that does not belong to a single feature, but to a combination. We discuss this issue further in Sect. 9.4.

9.1.1 Higher-Order Interactions

All examples so far illustrate interactions between pairs of features. However, there can also be interactions between more than two features, which are called *higher-order interactions* or *n-way interactions*. The interaction between FireControl and FloodControl is a 2-way interaction or first-order interaction. An interaction that occurs when three features are selected, but not for feature selections of pairs of these features, is called a 3-way interaction or second-order interaction. In Fig. 9.1, we illustrate the possible interactions between the three features A, B, and C by overlapping circles. Three features can give rise to three 2-way interactions and one 3-way interactions (intersections between circles).

Definition 9.2 If n features interact, but none of their strict subsets, this is called an *n-way interaction*. □

Example 9.5 Higher-order interactions are difficult to illustrate in small examples. They emerge from the complex interplay of multiple features. Here, we have created a small but dense code example of a Stack to illustrate a specific case in which three features interact. In this example, variability is encoded using preprocessor directives (see Sect. 5.3, p. 110). The code that coordinates the features of Stack is implemented in the form of nested preprocessor directives (that is, its absence does not cause misbehavior like in the fire-and-flood-control example).

```
1  class Stack {
2
3    boolean push(Object o) {
4  #ifdef LOCKING
5      Lock lock = lock();
6      if(lock == null) {
7  #ifdef LOGGING
8        log("lock failed for: "+o);
9  #endif
10       return false;
11     }
12 #endif
13 #ifdef UNDO
14     rememberValue();
15 #endif
16     elementData[size++] = o;
17     /*...*/
18   }
19
20 #ifdef LOGGING
21   void log(String msg) { /*...*/ }
22 #endif
```

```
23 #ifdef UNDO
24   boolean undo() {
25 #ifdef LOCKING
26     Lock lock = lock();
27     if(lock == null) {
28 #ifdef LOGGING
29       log("undo-lock failed");
30 #endif
31       return false;
32     }
33 #endif
34     restoreValue();
35     /*...*/
36 #ifdef LOGGING
37     log("undone.");
38 #endif
39   }
40
41   void rememberValue() { /*...*/ }
42   void restoreValue() { /*...*/ }
43 #endif
44 }
```

Fig. 9.2 Implementing a stack data structure with preprocessor directives; coordination code is implemented by nesting preprocessor directives

In Fig. 9.2, we show the code of the features Locking, Logging and Undo. There is code that belongs to the individual features, and code that belongs to combinations of features (intersections in Fig. 9.1), which is recognizable from nested *#ifdef* directives (the code is only included if multiple features are selected). Note particularly Line 29, which is only included if and only if all three features are selected. If this line caused a failure (for example, threw a null-pointer exception), the failure would occur only in products with all three features, but not in products that select only strict subsets of these features. □

Empirical evidence, for example by Kolberg et al. (2000), Kuhn et al. (2004), and Reisner et al. (2010), indicates that (a) higher-order interactions occur in practice, but also that (b) higher-order interactions are rare compared to interactions between pairs of features; most failures are related to individual features or interactions between two features (such as Examples 9.1–9.4). This empirical evidence suggests to concentrate effort on detecting 2-way feature interactions by analyzing mostly pairs of features, an approach also frequently taken in product-line testing 10.3.2.

9.2 Detecting Feature Interactions

Especially when features are developed independently, detecting and identifying feature interactions is a challenging task, and is still one of the big open problems of product-line development. There are many different strategies, mostly pioneered in the domain of telecommunication systems since the 1990s (Calder et al. 2003; Nhlabatsi et al. 2008), but there is no single strategy that can be claimed as general, scalable, and production-ready, yet.

A key problem of detecting feature interactions is that inadvertent feature interactions may lurk behind any feature combination. In a product line with n features, there are $\binom{n}{2} = \frac{n(n-1)}{2}$ pairs of features that may potentially interact, and there are $\binom{n}{k}$ possibilities for k-way interactions. In short, the exponential number of potential interactions limits any systematic and complete search. Even investigating only 2-way interactions (which are empirically much more likely to occur than higher-order interactions) may be overwhelming for industrial-sized product lines.

Within this book, we will not go into details regarding feature-interaction detection, but refer to the corresponding research literature; good starting points are the surveys by Calder et al. (2003) and Nhlabatsi et al. (2008). A typical strategy to detect feature interactions is to make requirements and assumptions regarding features explicit and check them as part of systematic *requirements analysis* in the domain-analysis phase (see Sect. 2.2, p. 19):

- At the requirements level, a typical strategy is to systematically search for shared resources. Two features that share resources may potentially interact over this resource. For example, the features FireControl and FloodControl from Example 9.2 both affect the resource water supply. A typical strategy is to model all resources relevant for each feature and subsequently investigate manually all pairs of features that share a resource.
- The strategy applied to resources can be used also for events (and preconditions of operations). Two features that react to the same event (or that have overlapping preconditions) are potential candidates for feature interactions. For example, the features CallForwarding and CallWaiting from Example 9.1 both react to the same event (that is, an incoming call on a busy line). Again, modeling events allows us to manually investigate all pairs of features reacting to the same event.
- Inconsistent requirements and conflicting goals of features revealed during domain engineering can also be an indicator of potential interactions. For example, features Acceleration to increase the speed of a car and AdaptiveCruiseControl to automatically adjust the distance to other cars by decreasing speed have conflicting goals. Again, requirements and goals need to be made explicit, for example, by modeling them.
- Making assumptions (or invariants) of features explicit can help detecting when an assumption is violated by other features. For example, feature Index in Example 9.3 assumes that the data structure is immutable, an assumption violated by feature Write.

In addition to manual investigation of requirements documents during domain analysis, formal methods can be applied to feature-interaction detection. For example, if preconditions or goals are formally stated, automatic reasoners can detect overlaps, inconsistencies, and critical program states. Similarly, if the behavior of the system can be modeled and assumptions or invariants can be specified formally, model checkers and other reasoners can detect violations.

Example 9.6 Consider the example of an e-mail client that has two optional features for encryption and automatic forwarding (Hall 2005). Both features have been

developed and tested based on the basic e-mail client, independently of the respective other feature. As it happens, both features interact in an inadvertent way: The interaction occurs if one host sends an encrypted e-mail to a second host that forwards the e-mail automatically to a third host. If the second host does not have the public key of the third host, it forwards the e-mail in plain text (the forwarding feature has been developed independently and thus does not take encryption into account).

Apel et al. (2013) have shown that this situation contradicts the specification that encrypted e-mails must never be sent in plain text over the network (Hall 2005), and that product-line model-checking technology can be used to detect this situation automatically. □

For a comprehensive overview of formal approaches for feature-interaction detection, see the recent surveys by Calder et al. (2003) and Nhlabatsi et al. (2008). Formal methods have been successfully applied on core models of product lines (Heymans 2012), but to scale them to be able to analyze source code instead of requirements models or manually abstracted models remains an open problem.

Finally, excessive product-line testing can be employed, if suitable test cases are available or if the assumptions of features are specified as run-time assertions. We return to product-line analysis in Chap. 10, including combinatorial testing for feature interactions.

For the remainder of this chapter, we assume that we already know which features interact. We focus on how to implement features with a known interaction by means of coordination code.

9.3 The Optional-Feature Problem

As we have seen so far, interactions between features often require additional coordination code. This code has to be implemented somewhere, and it has to take action only if the corresponding features are present in a given product. The combination of the fact that features can be optional and the need of code to coordinate features give rise to the optional-feature problem.

Definition 9.3 The *optional-feature problem* is the mismatch between intended variability (as specified in the feature model) and the actual variability provided by the implementation, due to coordination code. It occurs when two (or more) optional features interact, and the presence of coordination code reduces the intended variability of the product line. □

Suppose, in our database example, feature Statistics counts the number of transactions per second. If the user configures a database without feature Transactions, the implementation of Statistics breaks (as code concerning transaction management is missing). That is, the *implemented variability* (Statistics requires

```
layer DFS;

refines class Graph {
  void search(Strategy s) { ... }
}
```

```
layer Cycle;

refines class Graph {
  void hasCycles() {
    ...
    search(...);
    ...
  }
}
```

```
layer Weighted;

refines class Edge {
  double weight;
  void setWeight(double w) { ... }
}
```

```
layer ShortestPath;

refines class Graph {
  List shortestPath(Vertex a, Vertex
        b) {
    Edge e1, e2;
    ...
    if (e1.weight > e2.weight) ...
    ...
  }
}
```

Fig. 9.3 Excerpts of the implementations of the features Weighted and ShortestPath of the graph implementation in Jak/AHEAD

Transactions) does not align with the *intended variability* (both features shall be independently selectable).

The optional-feature problem is a specific, but common implementation-level instance of the feature-interaction problem. From a developer's perspective it deserves special attention, because, although often simple to detect, it occurs frequently. When talking about feature interactions so far, we discussed problems regarding incorrect *behavior* due to missing coordination code. In contrast, the optional-feature problem is concerned with incorrect *implementations* of variability that reduce *intended variability* or that have other negative effects such as nonmodular code, as we discuss in the remaining chapter.

Let us illustrate the optional-feature problem further with an example from our graph library.

Example 9.7 The optional-feature problem in the graph example. Suppose a version of our graph example in which the features Weighted and ShortestPath are both optional and independent. The feature model also specifies that feature Cycle conceptually depends on feature DFS.

Now, let us consider implementations of these features. In Fig. 9.3, we show excerpts of implementations using feature-oriented programming using Jak/AHEAD, which are based on the graph implementation of Sect. 6.1. Notice how the implementation of feature Cycle refers to method search from feature DFS, and how the implementation of feature ShortestPath refers to field weight from feature Weighted. Due to these references, there are implementation dependencies from feature Cycle to feature DFS and from feature ShortestPath to feature Weighted.

The implementation dependency between features Cycle and DFS is acceptable and possibly even not avoidable, because the dependency is fundamental in the domain. The feature model already documents this intended dependency.

```
1  layer BasicGraph;
2
3  class Edge {
4    double weight = 1;
5  }
```

```
6  layer Weighted;
7
8  refines class Edge {
9    void setWeight(double w) { ... }
10 }
```

```
11 layer ShortestPath;
12
13 refines class Graph {
14   List shortestPath(Vertex a, Vertex b)
                {
15     Edge e1, e2;
16     ...
17     if (e1.weight > e2.weight) ...
18     ...
19   }
20 }
```

Fig. 9.4 Alternative implementation of Weighted and ShortestPath in Jak/AHEAD, without any implementation dependency between them

But, the optional-feature problem occurs between the features ShortestPath and Weighted (in this implementation). Although desired, we cannot generate a product for a feature selection with feature ShortestPath, but without feature Weighted: The generated code would contain a dangling reference to field weight. Conceptually, however, such a product should be possible to generate, because both features are optional and independent in the feature model and desired by stakeholders (finding the path with the fewest edges). Hence, we have an implementation dependency that reduces the variability of the product line beyond the domain expert's intention, only because of the coordination code that let the features Weighted and ShortestPath properly interact (Line 27).

Since implementation dependencies are specific to one implementation but not essential in the domain, we can usually provide an alternative implementation without that dependency. We sketch a naive, alternative implementation in Fig. 9.4, in which we can freely combine the features ShortestPath and Weighted: We move the field weight with a default value to the base code, so that feature ShortestPath works independently of feature Weight, without any dedicated coordination code that impairs variability (however, now the base code contains a field that is unused in products without feature Weighted). We discuss different implementation strategies to solve the optional-feature problem in Sect. 9.4. □

As illustrated in the example, the optional-feature problem arises from a mismatch between variability specified in the feature model and variability provided by a specific implementation. The problem occurs when coordination code is hard-wired inside a feature. The optional-feature problem often manifests in the form of type errors (for example, a dangling reference to a feature that is absent), which can be detected when actually compiling a derived product (see also Chap. 10 for mechanisms how to detect type errors in all products of a product line).

The optional-feature problem and the feature-interaction problem are highly related, but the goals and challenges are different. In the feature-interaction problem, the challenge is identifying missing behavior (and coming up with corresponding coordination code), whereas, in the optional-feature problem, the challenge is implementing the coordination code such that it does not impair variability.

9.4 Implementing Feature Interactions

After we have identified that two features interact, we need to find a strategy to deal with the interaction, preferably a strategy that does not introduce the optional-feature problem. As one possibility, we can change the feature model to prevent the interaction (or, alternatively, enforce it in all products). For example, we could declare interacting features as mutually exclusive. However, usually the goal is to generate all products properly as intended by domain experts, that is, generate products with both features combined and with each feature in isolation.

Essentially, all resolutions of feature interactions (and the optional-feature problem) can be abstracted to and described by the following pattern: There are two implementations of the individual features, and, to use them together, some coordination code is required to patch up both features (typically, by adding additional code, but potentially also by overriding behavior or removing code). This pattern applies to all examples throughout this chapter.

- *Call forwarding and call waiting (Example 9.1).* When both features are selected, additional code should coordinate them. Coordination code can give priority to one feature, invoke them in sequence, or provide a configuration dialog for users to configure the desired behavior.
- *Fire and flood control (Example 9.2).* When both features are selected, additional coordination code is required to specify that feature FireControl overrules feature FloodControl.
- *Read-only data structures and indexes (Example 9.3).* In the data-structure example, the need for coordination code is especially obvious. When we select both features Write and Index, we need additional code to update the index on write operations.
- *Database transactions and statistics (Example 9.4).* Similar to the previous example, we need coordination code to implement the missing behavior of collecting statistics about transactions and for synchronizing the access to statistics data.
- *Weighted graphs and shortest path (Example 9.7).* Finally, feature ShortestPath can be implemented independently of feature Weighted and vice versa; but, to use them together, we need to include coordination code, such that the shortest-path algorithm uses the correct weights.

For illustration, we use a graphical notation of two interacting features, shown in Fig. 9.5. Each feature is represented by a circle and the overlap between them represents the code that coordinates their interaction. With this graphical notation, we can also illustrate the desired products, as shown in Fig. 9.6: Either we want both features with their interaction properly coordinated, or we want each feature in isolation, without any coordination code. In Fig. 9.1, we already illustrated an equivalent picture for higher-order interactions. Now the question is how to implement feature code and coordination code properly inside a product-line implementation. We will use the graphical notation to illustrate and discuss six implementation strategies.

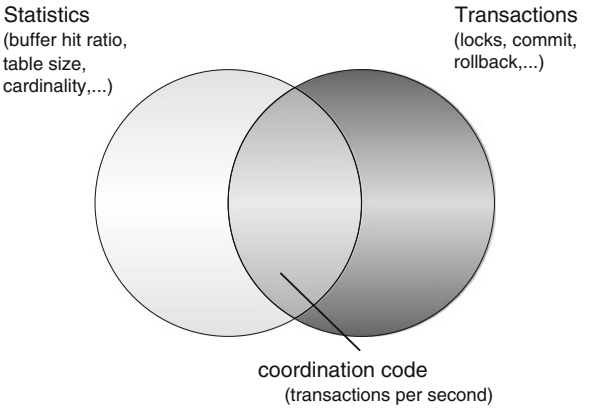

Fig. 9.5 The features Transactions and Statistics in concert

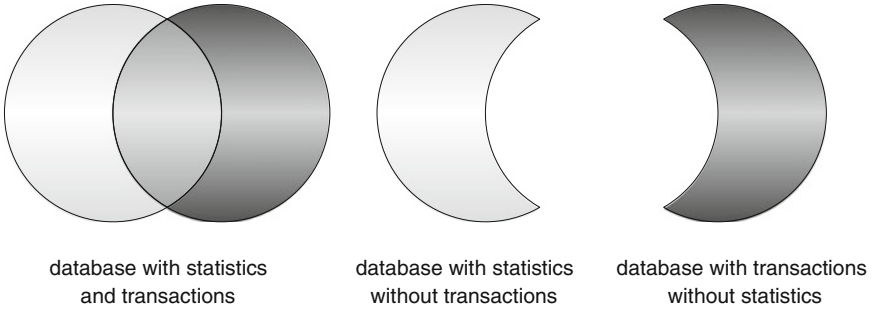

Fig. 9.6 Desired products using the features Transactions and Statistics

9.4.1 Implementation Strategies: Overview and Goals

What makes a good strategy to implement coordination code for feature interactions and to solve the optional-feature problem? There are at least four goals that we want to achieve.

1. *Variability.* The implementation strategy should allow the programmer to generate all products for all feature selections specified as valid in the feature model. That is, we do not want to reduce variability merely due to implementation issues, as described by the optional-feature problem (see Sect. 9.3).
2. *Implementation effort.* The implementation strategy should not require overwhelming implementation effort, because such implementation strategy would not be attractive to use in practice.
3. *Binary size and performance.* The implementation strategy should not increase binary size or decrease performance of products compared to an individual implementation of each product.

4. *Code quality.* Finally, the implementation strategy should not reduce code quality, which would make the product line harder to maintain. As discussed in Part II, there are many trade-offs, but the implementation strategy for interactions should fit to the interaction strategy chosen for features in the first place.

In the following, we discuss six implementation strategies. None of the strategies fulfills all goals; they have different trade-offs, as we will discuss. The discussed strategies are:

- *Change feature model*. Instead of a proper implementation, we exclude problematic feature combinations from the feature model.
- *Multiple implementations*. To account for configurations with and without coordination code, we implement the features separately for each combination.
- *Moving code*. Coordination code is moved to one of the interacting features or to a shared required feature.
- *Conditional compilation*. Using a preprocessor, the coordination code annotated and only complied if both features are present.
- *Optional weaving*. Coordination code is implemented as implicitly optional, using mechanisms inspired by aspect weaving.
- *Distinct module for coordination code*. A distinct module separates coordination code from feature code; the module is automatically included when both features are included.

We discuss these strategies separately, before we compare them in the end.

9.4.2 Change Feature Model

The simplest solution to resolve a known, undesired feature interaction is to *forbid* the problematic feature selection. Instead of solving the problem by adding proper coordination code or reimplementing features, we restrict the feature model to exclude problematic feature selections with an additional constraint. Similarly, we can declare an implementation dependency as domain dependency in the feature model. For example, we could mark the features FireControl and FloodControl from Example 9.2 as mutually exclusive, and we could enforce that feature ShortestPath cannot be selected without feature Weighted in Example 9.7.

This solution, of course, restricts variability (or at least acknowledges the reduced variability) compared to what *should* be valid products in the domain. On the positive side, this solution does not require to change the implementation and, thus, does not affect performance or code quality. Depending on the importance of the excluded products, the reduced variability can be acceptable or can have a serious impact on the strategic value of the product line.

When adopting this solution, we recommend documenting clearly which constraints in the feature model are driven by implementation dependencies. Such documentation helps to separate conceptual considerations in the domain from implementation issues.

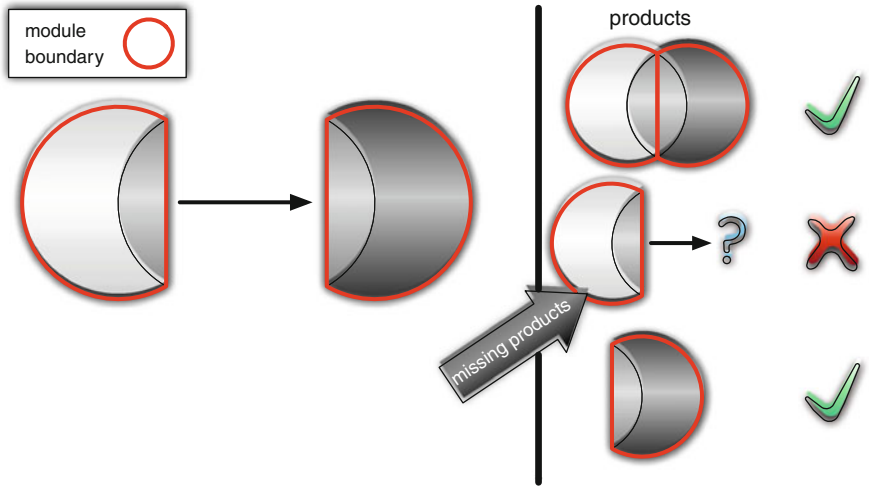

Fig. 9.7 Ignore feature interactions and restrict variability

In Fig. 9.7, we illustrate the solution graphically. We have two features that already contain coordination code, but the coordination code is encoded such that it causes an implementation dependency (as in Example 9.7 about the shortest-path algorithm). Here, we add the implementation dependency to the feature model, again disallowing the corresponding products. In a similar case (not shown graphically), we have two feature implementations but the necessary coordination code is missing (as in Example 9.2 about fire and flood control). We simply declare both features as mutually exclusive in the feature model, thus prohibiting products with both features.

9.4.3 Multiple Implementations

A simple strategy to handle interactions is to provide *multiple implementations* of a feature, one with and one without coordination code. For example, we can have two implementations of feature FloodControl from Example 9.2, one that always turns off water when flooding is detected and one that turns off water only after checking with feature FireControl. Similarly, we could provide an implementation of feature ShortestPath of Example 9.7 for weighted graphs and a second implementation for unweighted graphs. During product derivation, we would then include the suitable implementation, depending on which other features are selected.

Unfortunately, this strategy neglects code reuse, the prime benefit of product-line development, and encourages code replication instead. We need to implement a feature multiple times, one for each feature combination. Furthermore, the approach does not scale if a feature interacts with multiple other features. We need up to 2^n implementations of a feature that interacts with n other features.

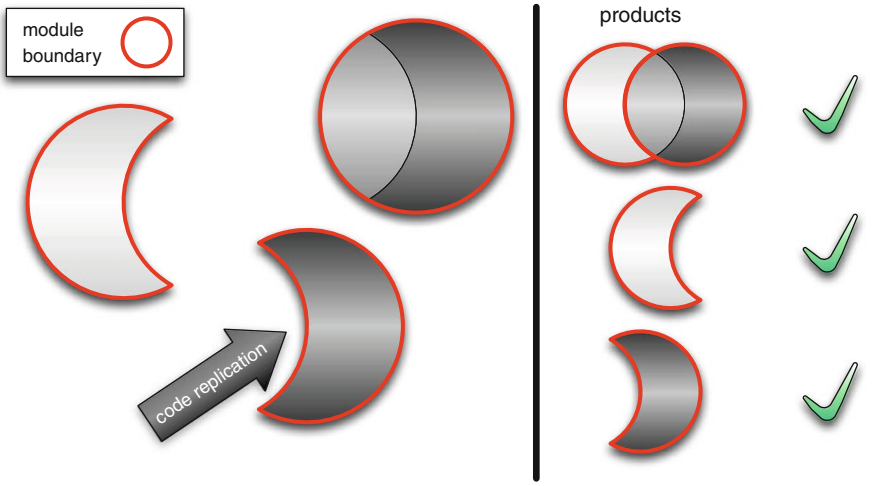

Fig. 9.8 Multiple implementations to make features optional

In Fig. 9.8, we visualize the multiple-implementations strategy. We provide two implementations of the dark-gray feature (say, feature ShortestPath), one with and one without code for coordination with other features. We can generate products for all feature combinations at the price of code replication and additional implementation effort.

9.4.4 Moving Code

In many cases, it is possible to implement the coordination code in one of the two features or in a third feature to which both features refer. For example, feature FloodControl from Example 9.2 could always include code for checking overriding conditions, independent of whether feature FireControl is selected. In the graph example in Fig. 9.4, we have already shown another instance of this solution: We have moved the field weight into the implementation of the base feature. The solution works, because we *move* the coordination code where it does not cause dependencies.

This solution has two drawbacks. The first drawback is a conceptual one: We violate the principle of *separation of concerns* (see Sect. 3.2.3, p. 55), because we move code to implementation units where it does not belong to. For example, feature FloodControl must now be aware of other overriding features as the fire sensor. Also, in our solution for feature ShortestPath in Fig. 9.4, we moved field `weight` that conceptually belongs to feature Weighted into the base code. With this implementation strategy, we give up the clear traceability from features to their implementation, as postulated in Sect. 3.2.2. The point is that coordination code does not belong

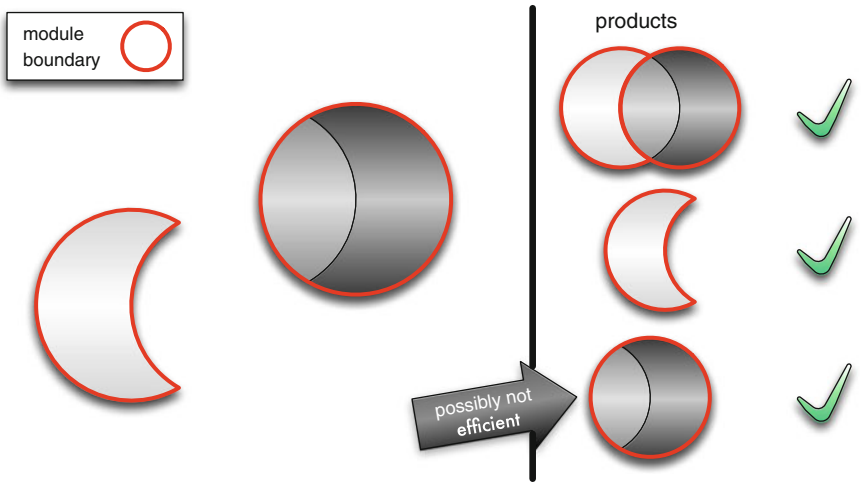

Fig. 9.9 Move code between features

to a single feature, but to combinations of features, so it is between modules that implement individual features.

Second, more technically, we include unnecessary code in some products. In the flood-control example, we would always include and execute code to check for a potential fire sensor, even in products without feature FireControl. In our solution for feature ShortestPath, all edges in all products now contain a field weight (which requires additional memory per object), even when neither feature ShortestPath nor feature Weighted is selected. Including unnecessary code potentially increases binary size, increases memory consumption, and decreases performance.

In Fig. 9.9, we visualize the implementation strategy. The coordination code is part of the implementation of one feature (or of some external feature that both features depend on; not shown here). The coordination code is implemented such that it does not cause a dependency; it remains as dead code in one feature, if the other feature is not selected. As a result, at least one product contains unnecessary code.

9.4.5 Conditional Compilation

If we use an annotative implementation strategy (see *annotation versus composition* 3.1.3, p. 50) for the product line, such as parameters (see Sect. 4.1) or preprocessors (see Sect. 5.3, p. 110), implementing glue code that is only executed if both features are selected is straightforward.

We simply place the coordination code that binds both features in nested *if* statements, nested *#ifdef* directives, or the like. Particularly compile-time approaches (see *binding times* in Sect. 3.1.1, p. 48), such as *conditional compilation* with

```
1  layer Weighted;
2
3  refines class Edge {
4    double weight;
5    void setWeight(double w) { ... }
6  }
```

```
7  layer ShortestPath;
8
9  refines class Graph {
10   List shortestPath(Vertex a, Vertex b)
           {
11     Edge e1, e2;
12     ...
13 #ifdef WEIGHTED
14     if (e1.weight > e2.weight) ...
15 #endif
16     ...
17   }
18 }
```

Fig. 9.10 Preprocessor-based implementation of Weighted and ShortestPath that implements coordination code as a conditional block

preprocessors, have the advantage that coordination code is only compiled and included when both features are selected. In Fig. 9.1 (p. 216), we have already illustrated interactions, including higher-order interactions, in terms of nested *#ifdef* directives.

Even when we use a primarily composition-based implementation strategy (such as components, frameworks, feature-oriented programming, and aspect-oriented programming, see Chaps. 4 and 6), we can use *#ifdef* directives inside composition units to conditionally remove unnecessary coordination code before compilation. We illustrate a possible solution for our shortest-path example (Example 9.7) in Fig. 9.10, where we eliminate unnecessary code from the composition unit at compile-time with a preprocessor.

This solution can implement all products without code replication and without compiling unnecessary code causing performance penalties. However, as already discussed in the context of parameters in preprocessors in Sects. 4.1 and 5.3, code quality is usually regarded as poor due to scattering and tangling of feature code and due to neglecting separation of concerns.

The preprocessors solution to known feature interactions is visualized in Fig. 9.11. It does not support an explicit separation of feature implementations. Consequently, in our graphics we have no module borders.

9.4.6 Optional Weaving

Instead of annotating optional coordination code inside the implementation *explicitly* with *#ifdef* directives or similar techniques, several researchers have explored more *implicit* mechanisms (Kästner 2007; Leich et al. 2005; Lohmann et al. 2011). These mechanisms are aimed at composition-based approaches (see *annotation versus composition* in Sect. 3.1.3, p. 50). The mechanisms, which we summarize under the name *optional weaving*, are inspired by the quantification mechanism in aspect-oriented programming (see Sect. 6.2, p. 141).

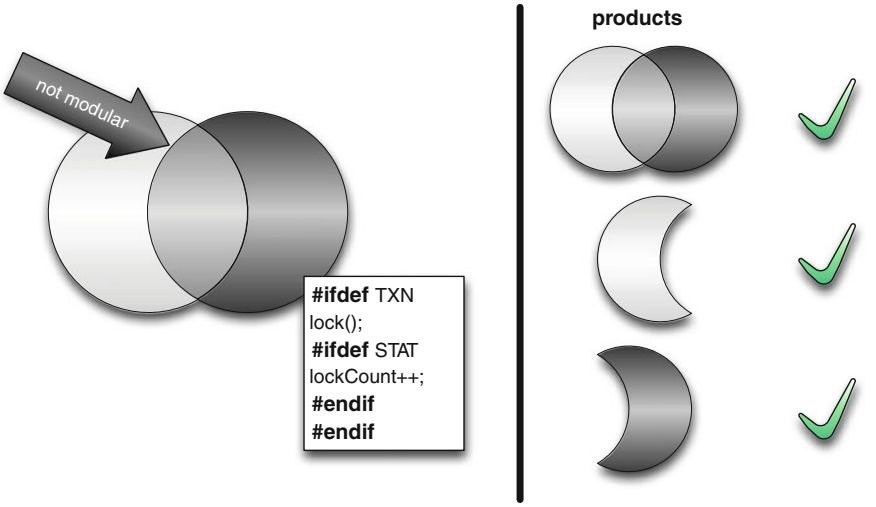

Fig. 9.11 Use of preprocessor annotations for implementing a feature interaction

In aspect-oriented programming, a developer declaratively specifies which code fragments to extend by means of a pointcut. The additional code is woven to all join points matched by the pointcut (possibly one, multiple, or even none).[1] Similarly, optional weaving declares where to add coordination code, but silently fails if the target is not present. As in the conditional-compilation strategy, coordination code is located in one of the features, but only included for compilation when both features are selected.

In Fig. 9.12, we illustrate the idea of optional weaving with a small code example for the fire-and-flood-control example. If the system does not contain methods startFireAlarm and endFireAlarm (when FireControl is not selected), the corresponding glue code in the advice body is simply never woven into the system.

Optional weaving is controversial and has not been fully explored yet. First, the weaving concepts of AspectJ are technically too restrictive for application of optional weaving at larger scale: Pointcuts cannot refer to class names that are possibly not present in the system, and optional weaving is not available for inter-type declarations. However, adopting the optional-weaving idea to other languages seems possible; AspectC is more flexible in this regard than AspectJ (Lohmann et al. 2011). Second, optional weaving depends on the silent failure to weave code when the target is not present. However, silent failure eliminates the chance to check or enforce weaving. For example, if a developer renames method startFireAlarm to beginFireAlarm without updating the pointcut, the coordination code is no longer woven into the system, but this failure is indistinguishable from correctly not weaving the coordination code if feature FireControl is not selected. Critics of optional weaving fear

[1] AspectJ issues a warning for a pointcut that does not match any join point shadow, but does not enforce any specific number of matches.

```
1  aspect FloodControl {
2    boolean isActive = true;
3
4    after(): execution(* *.floodingDetected()) {
5      if (isActive)
6        ...
7    }
8
9    before(): execution(* *.startFireAlarm()) {
10     isActive = false;
11   }
12   after(): execution(* *.endFireAlarm()) {
13     isActive = true;
14   }
15 }
```

Fig. 9.12 Example for optional weaving

that the mechanism is too implicit and will result in a large amount of optional code for which it remains unclear when exactly it is applied. However, optional weaving is a comparably recent solution, for which further research is needed.

9.4.7 Distinct Module for Coordination Code

In previous implementation strategies, we have discussed where to move coordination code. An alternative for composition-based implementations is to create yet another *module for code that coordinates features*. As illustrated in Fig. 9.14, we separate coordination code and compose it with the implementation of both features if and only if both features are selected. This strategy also scales for higher-order interactions with more additional modules for coordination code as illustrated in Fig. 9.1.

In the literature, the additional modules for glue code are well known, but have many different names. They are called *lifters* (by Prehofer (1997), because they lift the implementation of one feature to the implementation of another feature), *tiles* (by Kühne (1999), because they connect features from different dimensions shown as a matrix), *derivatives* (by Liu et al. (2006), because they are derived from two features), *connectors* (terminology in Eclipse, because they connect two other plug-ins), and so forth.

In our fire-and-flood-control example (Example 9.2), we could implement both features in separate modules and add the coordination code (which overrides one feature with the other) as a separate module. The separate module is *automatically* included when both features are selected. For our shortest-path example (Example 9.7), we have exemplified one possible solution in Fig. 9.13: The implementation of feature ShortestPath calls a method isLonger with a default implementation that is overridden by coordination code in a separate module (ShortestPath_Weighted).

The key to this implementation strategy is that the additional module for the coordination code is automatically included during generation in the product derivation process, if and only if all participating features are selected. Some automation should

```
1  layer ShortestPath;
2
3  refines class Graph {
4    List shortestPath(Vertex a, Vertex b)
               {
5      Edge e1, e2;
6      ...
7      if (isLonger(e1, e2)) ...
8      ...
9    }
10   boolean isLonger(Edge e1, Edge e2) {
11     return false;
12   }
13 }
```

```
14 layer Weighted;
15
16 refines class Edge {
17   double weight;
18   void setWeight(double w) { ... }
19 }
```

```
20 layer ShortestPath_Weighted;
21
22 refines class Graph {
23   boolean isLonger(Edge e1, Edge e2) {
24     return e1.weight > e2.weight;
25   }
26 }
```

Fig. 9.13 Alternative implementation of Weighted and ShortestPath with an additional module for the glue code between them

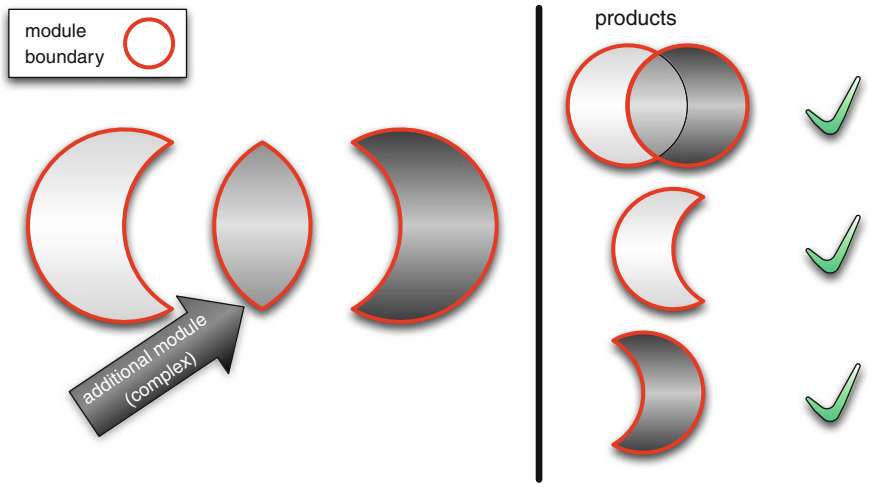

Fig. 9.14 Distinct modules implement coordination code

make sure that we cannot forget the coordination code. In the simplest case, we can create a new feature for the coordination code in the feature model and use constraints to enforce consistent selection (for example, `ShortestPath_Weighted` ⇔ `(ShortestPath` ∧ `Weighted)`). More sophisticated support in the generation step can help to hide the additional modules. Liu et al. (2006) and Batory et al. (2011) discuss a conceptual and theoretical framework that includes also a concept for naming and automatic selection.

In some cases it can be debatable whether the added module should be hidden and automatically selected, or whether it should be exposed as an extra feature. In our statistics-and-transactions example, we could argue that collecting statistics about transactions is another optional feature, but we could also argue that it belongs conceptually to the features transactions and statistics and should be selected auto-

matically. In most cases, though, the extra module clearly does not constitute a domain abstraction that should be modeled as feature, but mere coordination code that should be included automatically. For example, the coordination code of the fire-and-flood-control example and the write-and-index example should not be offered as optional feature; not including the coordination code when both features are included would be considered as interaction bug in these scenarios.

An interesting insight about this strategy is that it can also work in open-world scenarios (see *software ecosystems* in Sect. 4.3.5, p. 86) where features are provided by independent developers without central authority. For example, in Eclipse, when two plug-ins interact (or should interact), we can add the corresponding coordination code as another plug-in, which is typically called *connector* in this domain. However, a yet unsolved problem in open-world scenarios is to detect the interaction and make sure that a corresponding connector is provided, because there is no central product derivation mechanism that could automatically include the required coordination code.

The use of distinct modules for coordination code is a way of handling interactions, but there are also drawbacks. The number of derivatives may explode in cases where many interactions exist. This may lead to a high number of additional but potentially very small modules that can be overwhelming for developers and hard to understand in isolation. In the future, this increased complexity may be more manageable by tools supporting the automatic refactoring of existing coordination code into distinct modules (see Chap. 8), and their tool-driven maintenance throughout the whole lifetime of a software product line (see Chap. 7)—but more research is needed.

9.4.8 Comparison of Solutions

After we have discussed the implementation strategies in isolation, we can now take a look at the complete picture and discuss how the strategies perform with regard to our four goals of variability, implementation effort, binary size and performance, and code quality.

- Regarding variability, all implementation strategies, except mere changes to the feature model, support the full variability.
- The implementation effort differs significantly. Creating multiple implementations per feature requires significant overhead, and also creating distinct modules for coordination code requires significantly rewriting existing code and creating additional modules.
- Potential overhead regarding binary size, memory consumption, and performance is a problem when moving the code. For optional weaving, we do not yet have sufficient experience.
- Code quality can be discussed controversially. However, code replication of the multiple-implementation strategy is obviously a problem. Also, suboptimal sepa-

Table 9.1 Comparison of implementation approaches for the feature interaction problem

Implementation strategy	Variability	Implementation effort	Binary size and performance	Code quality
Change the feature model		✓	✓	✓
Multiple implementations	✓		✓	
Moving code	✓	✓		✓
Conditional compilation	✓	✓	✓	
Optional weaving	✓	✓	✓?	?
Distinct modules for glue code	✓		✓	✓

ration of concerns and scattering and tangling of code associated with conditional compilation is typically regarded as poor quality. Also, the implicit mechanisms of optional weaving potentially threaten code quality.

We summarize the discussion in Table 9.1.

As the table shows, there is no clear generally preferable strategy. Merely the multiple-implementation strategy seems never to be a good idea. From the remaining strategies, we need to select depending on the context and on which goal is currently most important to developers. Changing the feature model is the easiest solution, but decreases variability. Moving code is also simple, but potentially produces overhead. The main criticism of conditional compilation is its effect on code quality. Creating distinct modules for coordination code seems elegant, but can require significant additional effort from developers. Optional weaving is a new research approach for which not much experience is available. In practice, developers typically will mix and match the approaches according to their needs.

9.5 Experience

From the previous discussion, it seems that developers have to decide for the lesser evil when selecting an implementation strategy for feature interactions. All strategies have different trade-offs and none is without drawbacks. To gain experience, we conducted two case studies on the database systems Berkeley DB (both the Java edition and C edition) and FAME-DBMS (C++ implementation). In all cases, we observed and analyzed product-line development, counted instances of the optional-feature problem, and discussed and explored different implementation strategies. The results were published in Kästner et al. (2009), but here we repeat the key results to provide some context for the different strategies.

The case studies followed different implementation strategies. In the case of Berkeley DB, we decomposed an existing system into a product line. We started with legacy code that already contained many features, without making them explicit or configurable. Interactions between features were already hard-coded

(all features were hard-coded as part of the mandatory base code, so was the coordination code between them). We subsequently extracted features and made them optional, a process in which we found coordination code and needed to decide how to implement it. In the case of FAME-DBMS, we developed the product line from scratch. However, since the domain of database systems is well known, we could easily anticipate and plan interactions between features. In both case studies, we focused on the optional-feature problem, that is, how to implement known feature interactions without restricting the intended variability.

9.5.1 Decomposition of Berkeley DB

Oracle's *Berkeley DB*[2] is an open-source database engine implemented in approximately 70,000 lines of code, that can be embedded into applications as a library. In two independent endeavors, we decomposed both the Java and the C version of Berkeley DB into features (described in more detail by Kästner et al. (2007) and Rosenmüller et al. (2008)). We pursued a composition-based implementation strategy with the goal of separating each feature in a distinct module, using aspect-oriented programming with AspectJ in the Java version (see Sect. 6.2, p. 141) and feature-oriented programming with FeatureC++ in the C version (see Sect. 6.1, p. 130).

In the Java version, we identified 38 features. Almost all features are optional and there are only 16 domain dependencies; in theory, we should be able to derive 3.6 billion different products. However, implementation dependencies occurred much more often than domain dependencies. With manual and automated source-code analysis, we found 53 implementation dependencies corresponding to 2-way interactions that were not covered by domain dependencies. We did not find higher-order interactions. We show an excerpt of features and corresponding dependencies between their implementation modules in Fig. 9.15 (implementation dependencies marked with 'x'; there are no domain dependencies between the shown features). Overall, in Berkeley DB, the optional-feature problem occurred between 53 pairs of features, which are independent in the domain, but not in their implementation.

Changing the feature model to simply document all implementation dependencies is not acceptable, because this would restrict the ability to generate tailored products drastically. In pure numbers the reduction from 3.6 billion to 0.3 million possible products may appear acceptable, considering that still many products can be generated. Nevertheless, when having a closer look, we found that especially in the core of Berkeley DB, there are many implementation dependencies. Important features regarding statistics, transactions, memory management, and database operations shown in Fig. 9.15 must be included in virtually every valid feature selection. The remaining variability of 0.3 million products is largely due to several small independent debugging, caching, and IO features. Considering all implementation dependencies, essentially all intended variability is lost.

[2] http://www.oracle.com/database/berkeley-db

	11.	12.	13.	20.	27.	29.	30.	31.	33.
11. Atomic Transactions									
12. FSync	x								
13. Latch	x	x							
20. Statistics	x	x	x						
27. IN Compressor	x	x	x	x					
29. DeleteDbOperation	x		x	x	x				
30. TruncateDbOperation	x		x			x			
31. Evictor			x	x		x			
33. Memory Budget	x		x	x		x		x	

Fig. 9.15 Implementation dependencies ('x') in Berkeley DB (excerpt)

In the C version, which has a very different architecture and was independently decomposed into a different set of features, we identified 24 features (see Rosenmüller et al. (2008) for details). But, the overall picture is similar: With only 8 domain dependencies almost 1 million products are conceptually possible, but only 784 products can be generated considering all 78 implementation dependencies between feature pairs that we found. Again, important features were de facto mandatory in every feature selection.

These numbers give a first insight into the impact of the optional-feature problem. We found more instances of feature interactions than there are features (as explained in Sect. 9.2, there can be a quadratic number of interactions between pairs of features, and even an exponential number considering also higher-order interactions). In Berkeley DB, the strategy to merely change the feature model reduces variability to a level that makes the product line almost useless.

Exploring Implementation Srategies

After the analysis revealed that changing the feature model is not a general option, we explored different solutions to eliminate implementation dependencies. Focusing on a clean composition-based implementation, and following the principle of separation of concerns, we started with creating distinct modules for coordination code.

In the Java version, we first created nine distinct modules to encapsulate coordination code of all nine interactions of the feature Statistics. These nine modules alone required over 200 additional pieces of advice or inter-type declarations with AspectJ. Of 1867 lines of code of the statistics feature, we rewrote 76 % as modules (which would also be the amount of code we needed to move into different features for the moving-code strategy). This shows that most of the functionality of feature Statistics is in its interactions with other features. In the C version, we created 19 distinct modules for coordination. In both versions, we experienced the necessary rewrites as rather tedious. We needed between 15 min and 2 h for each new module, depending on the amount of code. Due to the high effort, we refrained from creating distinct modules for all implementation dependencies.

Next, we experimented with conditional compilation. In the C version, we used *#ifdef* statements inside FeatureC++ modules, as illustrated in Sect. 9.4.5. In the Java

version, we used a preprocessor-like environment *CIDE* to eliminate all implementation dependencies (see *virtual separation of concerns* in Sect. 7.4 p. 184). Using conditional compilation was significantly faster than implementing distinct modules, because no changes to the code were necessary except for introducing annotations. However, we deviated from our original goal of a clean composition-based implementation. As a result, feature code is scattered and tangled, with up to 300 annotated code fragments in 30 classes per feature.

In Berkeley DB, both creating distinct modules for coordination code and conditional compilation were acceptable despite their drawbacks. While we prefer a clean separation of concerns, we felt that the required effort was overwhelming. In this project, a mixture of additional modules and conditional compilation felt as a good compromise to us.

9.5.2 Design and Implementation of FAME-DBMS

The question remains of whether the high number of implementation dependencies is caused by the design of Berkeley DB and our subsequent refactoring, or whether they are inherent in the domain. In the latter case, they should also appear in a database product line that was designed from scratch.

FAME-DBMS is a prototype of a database product line implemented with FeatureC++ (see *feature-oriented programming* in Sect. 6.1 p. 130). FAME-DBMS was designed specifically for small embedded systems. Its goal was to show that product-line technologies are appropriate to tailor data management for special tasks in even small embedded systems (for example, BTNode with Nut/OS, 8 MHz, and 128 kB of memory). FAME-DBMS is minimalistic and provides only essential data management functionality to store and retrieve data using an API. Advanced functionality such as transactions, set operations on data, or query processing was not part of the prototype. The initial development that we describe here was performed in a project by a group of four graduate students at the University of Magdeburg, after our experience with Berkeley DB.

Design

FAME-DBMS was designed after careful domain analysis and analysis of scenarios and existing embedded database engines. The initial feature model of FAME-DBMS as presented in the kick-off meeting of the project, is depicted in Fig. 9.16 (only layout and feature names were adapted for consistency). It contains 14 concrete features (grayed features were not linked to code). To customize FAME-DBMS, we can choose between different operating systems, between a persistent and an in-memory database, and between different memory-allocation mechanisms and paging strategies. Furthermore, index support using a B^+-tree is optional, so is debug logging, and finally it is possible to select from three optional operations *get*, *put*, and *delete*.

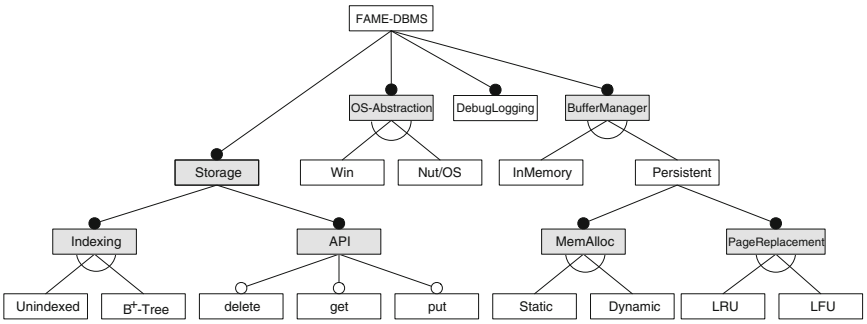

Fig. 9.16 Initial feature model of FAME-DBMS

Fig. 9.17 Domain dependencies ('o') and implementation dependencies ('x') in FAME-DBMS

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1. Nut/OS														
2. Win	o													
3. InMemory														
4. Persistent			o											
5. Static	x	x	o	o										
6. Dynamic	x	x	o	o	o									
7. LRU			o	o										
8. LFU			o	o			o							
9. Unindexed														
10. B+-tree									o					
11. Put	x	x		x					x	x				
12. Get	x	x							x	x	x			
13. Delete	x	x		x					x	x	x	x		
14. Debug	x	x	x	x	x	x	x	x	x	x	x	x	x	

The intended variability, captured by the feature model, describes 320 valid feature selections.

Soon after the initial design, the students realized that many of the features would require code to coordinate interactions. In Fig. 9.17, we show the domain dependencies ('o') and the implementation dependencies ('x') that were expected in addition. The latter give rise to the optional-feature problem. For example, the debug logging feature has an implementation dependency with every other feature (it extends them with additional debugging code), but should be independent according to the feature model. Also the features Get, Put, Delete, Nut/OS, and Win interact with many other features. This analysis already shows that it is necessary to find suitable strategies for the implementation of coordination code. Again, merely changing the feature model was not an option, because this would almost entirely eliminate variability.

Implementation

We left the implementation up to the students. We recommended the solution with extra modules for coordination code, but did not enforce it. In the remainder of this section, we describe the final implementation at the end of the project and discuss choices and possible alternatives.

First, as expected, the students chose implementation strategies to implement coordination code without restricting variability. There were only two exceptions, in which they changed the feature model: They merged features Put and Delete, so they cannot be selected independently, and they marked feature Get as mandatory. With this choice, they reduced the number of interactions they needed to implement from 36 to 22. At the same time, they reduced the number of possible products from 320 to 80. The intension behind changing the feature model was the following: Although there are use cases for a database that can write but not delete data, or even for a write-only database (see Szewczyk et al. (2004)), these feature selections are so rarely demanded that the students considered the reduced variability acceptable.

Second, the 11 interactions of feature DebugLogging with other features have been implemented using conditional compilation. This scattered the debugging code across all implementation modules. Alternatively, some debugging code could have been moved into the base implementation causing a small run-time penalty, or 11 additional modules could have been created. The students decided to use conditional compilation despite the scattered code, because it required least effort.

Third, the implementation of feature B^+-tree always contains code to add and delete entries, even in read-only configurations. This is an instance of the moving-code implementation strategy. In read-only configurations with feature B^+-tree, the additional code is included but never called. The students choose this strategy, because it was simpler than creating distinct modules. An ex-post evaluation revealed that the unnecessary code increased binary size by 4–9 kB (5–13 %; depending on the remaining feature selection).

Fourth, the remaining 10 interactions were implemented using distinct modules, following our original recommendation. The students considered the additional effort as the lesser evil compared to a further reduction of variability, a further scattering of code with preprocessor annotations, or a further unnecessary increase in binary size. The multiple-implementations strategy was not considered at any time.

The implementation of FAME-DBMS used a combination of various strategies, but still increased the code size of some products, reduced variability, and required effort for creating 10 additional modules. Even in such small product line, the optional-feature problem pervades the entire implementation.

All our case studies are from the database domain and we believe that other developers may have chosen different trade-offs. The frequent occurrence of the optional-feature problem may be due to the domain or the used fine granularity, but we believe that observations will be possible in other product lines. In each case, developers have to make their own choices with regard to implementing coordination code, but we hope that sharing our experience helps with these trade-offs.

9.6 Further Reading

When the feature-interaction problem became a crisis in the telecommunications industry in the late 1980s (Bowen et al. 1989), researchers began to develop formalisms to enable automatic detection of feature interactions (Blom et al. 1994; Bruns et al. 1998; Felty and Namjoshi 2003; Lin and Lin 1994; Pomakis and Atlee 1996), architectures that avoid classes of interactions (Hay and Atlee 2000; Jackson and Zave 1998; Utas 1998; van der Linden 1994; Zave 2010), and techniques for resolving interactions at run-time (Griffeth and Velthuijsen 1994; Tsang and Magill 1998).

While the pioneering work on the feature-interaction problem in telecommunication systems was foundational and successful (see the surveys by Calder et al. (2003) and Nhlabatsi et al. (2008)), it is limited, as it is based on assumptions that hold for telecommunication systems, but not for other domains, for example, the enforcement of architectural styles or the need of explicit specifications of feature interactions.

Recently, researchers began to propose solutions (mostly based on verification techniques) to the feature-interaction problem that take the specific properties of software product lines into account, especially, the possibly exponential number of products (Apel et al. 2013; Classen et al. 2012; Lauenroth et al. 2009; Thüm et al. 2012).

The testing community has introduced the concept of interaction faults, which denotes implementation defects that are only triggered when multiple parameters are set to specific values. Garvin and Cohen (2011) provide a good definition of feature-interaction faults, which includes the possibility that a defect occurs only when a feature is *deselected* in combination with another feature. Several researchers have empirically investigated how defects and code paths in practical software systems are related to feature interactions (Kolberg et al. 2000; Kuhn et al. 2004; Reisner et al. 2010). The general insight is that the majority of bugs and code paths are triggered by individual features or n-way interactions with low values for n. This confirms a tendency in the testing community to search primary for 2-way interactions, in software product lines and any other software systems.

Our discussion of the optional feature problem and the trade-offs of different implementation strategies, as well as our experience report, is based on prior work (Kästner et al. 2009). Most implementation strategies are quite obvious and not discussed in-depth in the literature. However, the strategy of using additional modules was discussed implicitly and explicitly in many contexts (Kühne 1999; Liu et al. 2006; Lopez-Herrejon et al. 2005; Prehofer 1997). Also, optional weaving has been discussed repeatedly in recent years (Adler 2010; Kästner 2007; Leich et al. 2005; Lohmann et al. 2011).

Exercises

9.1. Collect a list of interactions that may appear between (a) common features of a mobile phone and (b) features identified in Exercise 2.4 (p. 43).
9.2. Hall (2005) has collected a list of possible interactions between basic features of an e-mail delivery system, of which we show two below. Discuss possible strategies how these two interactions could have been detected.

(a) Bob sends a signed message to Alice, who has no signing key provisioned. Yet Alice forwards the message to a third party. The message will arrive there signed, not by the sender (Alice), but by the originator (Bob). Thus, the signature will not verify, even if the third party has a verifying key for Bob, since the verification is defined to determine whether the message was signed by the sender of the message.
(b) Bob sets up forwarding to Alice. Alice has an auto-response feature enabled. A third party sends a message to Bob, which is forwarded to Alice. The auto-response is sent back to Bob and then forwarded to Alice. Thus, messages arriving at Alice via Bob are not effectively auto-responded.

9.3. Are all feature interactions undesired? Discuss this issue by means of the following feature interaction in a phone system:[3] Alice is forwarding calls to Bob, and Bob is forwarding calls to Carol. If Alice is called, should the call be forwarded to Carol?
9.4. Extend the graph library with an additional feature that introduces a feature interaction. Discuss the implementation strategies from Sect. 9.4; argue which implementation strategy is most suitable to resolve the optional-feature problem in this case.
9.5. Did any implementation of the chat system (Exercise 4.1 and following) give rise to feature interactions or the optional-feature problem? If yes, how did you handle the interaction?

Below is a list of extensions for the chat system. For each extension:

(a) Explain which interaction is triggered by the extension. What is the required coordination code in this case (if any)?
(b) Modify the chat system of Exercise 4.5, 6.2, or 6.4 accordingly.
(c) Observe whether the extension triggers an instance of the optional-feature problem. Illustrate the optional-feature problem in terms of intended and actual variability.
(d) Explore and compare all implementation strategies discussed in Sect. 9.4. Argue which implementation strategy is most suitable for this extension.

The extensions for this exercise are:

1. Document in the history of the server whenever a user tries to authenticate with an incorrect password (features History and Authentication).

[3] Adopted from: http://www2.research.att.com/~pamela/faq.html.

2. Ensure that authentication messages are encrypted and that the spam filter always works on decrypted messages (features Encryption, Authentication, and SpamFilter from Exercise 4.5, p. 96)
3. Ensure that a message is never encrypted twice (encryption features).
4. The dialog showing the history should display the color of the message (features History and Color).
5. Even when the user is busy, messages with red color are still delivered (features Color and BusyStatus from Exercise 6.4, p. 173).

9.6. Consider a product line with 20 optional features of which 10 participate in 2-way interactions (each feature participates only in one interaction).

(a) How many modules are necessary to implement all coordination code using the distinct-modules strategy?
(b) If we change the feature model to forbid interacting feature to be selected independently, how many valid products can be generated?