Chapter 4
Classic, Language-Based Variability Mechanisms

After reading the chapter, you should be able to

- implement features with run-time parameters, white-box frameworks, black-box frameworks, and components,
- discuss trade-offs between these implementation techniques,
- select a suitable implementation technique for a given product line,
- choose suitable design patterns to implement variability,
- explain limitations of inheritance and possible solutions, and
- decide what size is appropriate for a component in a product line.

There are many ways to implement variable code; some have been used long before the advent of software product lines. Even a simple `if` statement offers a choice between different execution paths. To prevent cluttering of code with `if` statements, to enhance feature traceability, to provide extensibility without the need to change the original source code, and to provide compile-time (or load-time) variability, developers have identified many common programming patterns to support variability.

In this chapter, we discuss four implementation techniques in detail: parameters (Sect. 4.1), design patterns (Sect. 4.2), frameworks (Sect. 4.3), and components and services (Sect. 4.4). All of them can be encoded in mainstream programming languages. In Chap. 5, we discuss approaches based on configuration-management tools (version control systems, build systems, and preprocessors) that operate on top of source code (see Sect. 3.1.2, p. 49) to achieve and manage variability.

The language mechanisms and tools discussed in this and the next chapter are well-known and commonly used in practice. Most industrial software product lines are implemented with one or more of them. As we will see, each has distinguishing properties and gives rise to trade-offs, which we discuss in terms of the classifications (binding times, granularity, and so forth) introduced in Chap. 3.

S. Apel et al., *Feature-Oriented Software Product Lines*,
DOI: 10.1007/978-3-642-37521-7_4, © Springer-Verlag Berlin Heidelberg 2013

4.1 Parameters

A simple way to implement variability is to use conditional statements (such as if and switch) to alter the control flow of a program at run time. In our context, conditional statements are typically controlled by configuration parameters passed to a method or a module, or set as global variables in a system. Different parameter assignments lead to different program executions.

There are many ways to set configuration parameters. Often, command-line parameters (such as in "ssh -v") or configuration files (such as system.ini or httpd.conf) are read at startup and stored in global variables. Also, users can change parameters in preference dialogs, sometimes with immediate effects, other times requiring a restart. Furthermore, values of variables can also be hard-wired in source code, so changing them requires recompilation.

Of course, configuration parameters do not have to be stored in global variables. It is quite common to pass configuration parameters as method arguments. Sometimes, configuration parameters are propagated from method to method, or from class to class, across the entire source code. Global variables are convenient to access, avoiding the need for additional method parameters, but they also discourage a modular solution, in which each module or method describes the configuration parameters it expects as part of its interface.

In a feature-oriented setting, we expect one Boolean parameter per feature. In a disciplined implementation, the relationship between features and parameters is expressed and enforced by naming conventions and thus easy to trace (see Sect. 3.2.2, p. 54).

Example 4.1 In Fig. 4.1, we show our graph example with two configurable features, Weighted and Colored, implemented as global configuration parameters. Class Conf stores two parameters (public static is the Java way of specifying a global variable), possibly initialized during startup from command-line parameters or configuration files. These parameters are evaluated inside if statements in the classes Graph, Node, and Edge to trigger feature-specific behavior on demand. □

4.1.1 Discussion

Implementing variability with parameters is straightforward. For this reason, it is widely used in practice. Variation is evaluated at run time when conditional statements are executed (see *load-time and run-time binding* in Sect. 3.1.1, p. 48). The parameter approach shares the usual benefits and drawbacks of run-time binding:

- All functionality is included in all deployed products, even if it is statically known that a feature will never be selected. This has potentially negative implications for resource consumption, performance, and security: First, deactivated functionality is still delivered. Second, performing run-time checks requires additional

```
1  class Conf {
2    public static boolean COLORED = true;
3    public static boolean WEIGHTED = false;
4  }
5
6
7  class Graph {
8    Vector nodes = new Vector();
9    Vector edges = new Vector();
10   Edge add(Node n, Node m) {
11     Edge e = new Edge(n,m);
12     nodes.add(n);
13     nodes.add(m);
14     edges.add(e);
15     if (Conf.WEIGHTED)
16       e.weight = new Weight();
17     return e;
18   }
19   Edge add(Node n, Node m, Weight w) {
20     if (!Conf.WEIGHTED)
21       throw new RuntimeException();
22     Edge e = new Edge(n, m);
23     e.weight = w;
24     nodes.add(n);
25     nodes.add(m);
26     edges.add(e);
27     return e;
28   }
29   void print() {
30     for(int i=0; i<edges.size(); i++){
31       ((Edge) edges.get(i)).print();
32       if(i < edges.size() - 1)
33         System.out.print(" , ");
34     }
35   }
36 }
```

```
37 class Node {
38   int id = 0;
39   Color color = new Color();
40   Node (int _id) { id = _id; }
41   void print() {
42     if (Conf.COLORED)
43       Color.setDisplayColor(color);
44     System.out.print(id);
45   }
46 }
47
48
49 class Edge {
50   Node a, b;
51   Color color = new Color();
52   Weight weight;
53   Edge(Node _a, Node _b) {a=_a; b=_b;}
54   void print() {
55     if (Conf.COLORED)
56       Color.setDisplayColor(color);
57     System.out.print(" (");
58     a.print();
59     System.out.print(" , ");
60     b.print();
61     System.out.print(") ");
62     if (Conf.WEIGHTED) weight.print();
63   }
64 }
65
66
67 class Color {
68   static void setDisplayColor(Color c)...
69 }
70 class Weight {
71   void print() { ... }
72 }
```

Fig. 4.1 Graph library: Variability implemented with parameters

computing overhead. In our example of Fig. 4.1, even though we know that we never use feature Colored, the application still contains class Color, evaluates an additional if statement whenever a method print is called, and requires memory for an additional field of every node and edge object. Third, we cannot even prevent others from instantiating objects or invoking methods of deactivated feature code, other than throwing run-time errors (see method add in Lines 19–28 of Fig. 4.1). Finally, shipping unused code opens unnecessary potential targets for attacks, such as buffer-overflow attacks.

- It is possible to alter a feature selection without stopping the program. However, run-time changes are nontrivial in general, as a feature's code may depend on certain initialization steps or assume certain invariants. For example, in Fig. 4.1, we might run into a null-pointer exception, if we enabled Weight at run time, because the field weight of previously created edges was uninitialized. In such cases, it might be easier to require a restart of the program, when configuration parameters change.
- An advantage of passing configuration parameters as method arguments (in contrast to using global variables or using compile-time variability) is that different

parts of the control flow can be configured differently. For example, we could use colored graphs and uncolored graphs in the same program.

It is possible to specialize a program statically when some configuration parameters are known at compile time. Many compilers include optimizations that remove dead code. For example, if we know at compile time that a parameter is always deactivated (for instance, because it is defined as a constant in the source code), the compiler can remove corresponding conditional statements and their bodies. However, deciding when to remove entire methods or classes is less obvious and rarely implemented in contemporary compilers. Compilers also differ in the amount of analysis they perform to recognize dead code when configuration parameters are passed across method boundaries, are modified, or are assigned to other variables. Beyond simple dead-code optimizations, more sophisticated optimizations using partial evaluation can be applied to statically eliminate variability (Jones et al. 1993), but these are far from mainstream or easy to use yet. All in all, despite limited possibilities, the parameter approach is not well-suited for compile-time binding. We will see a specialized form of `if` statements for compile-time binding later in the context of preprocessors, in Sect. 5.3.

Dependencies between features must be checked when the parameters are configured at startup, or whenever parameters are changed at run time. In our experience, feature dependencies are rarely checked in a systematic way when using parameters. The parameter approach cannot statically guarantee invariants on feature selections (meaning that without considerable effort, it may be possible to activate features at run time that are not compatible with each other).

Adding conditional statements to the source code is a form of annotation (see *annotation versus composition* in Sect. 3.1.3, p. 50). Annotations with conditional statements are available at a fine granularity (see *granularity* in Sect. 3.2.5, p. 59). With `if` statements, we can change the program behavior at statement level, and many languages even provide conditionals at the expression level (such as, "`a?b:c`" in Java). Most languages do not provide conditionals at the level of methods or classes, because they are not necessary to influence the behavior of the program (they become relevant only for compile-time variability, which is not the goal of the parameter approach). Extensions are usually performed invasively, but therefore also do not require specific preplanning (see *preplanning* in Sect. 3.2.1, p. 53). Configuration parameters can be encoded in essentially all programming languages; however, they are usually not applicable to noncode artifacts, such as, design documents, grammars, and documentation (see *uniformity* in Sect. 3.2.6, p. 60).

Configuration parameters often lead to implementations that have a poor code quality. On the one hand, global parameters are tempting but reduce modularity (they violate separation of concerns and potentially breach information-hiding interfaces). On the other hand, propagating method arguments requires boilerplate code and may lead to methods with many parameters (considered as code smell by Fowler (1999, pp. 78f)). A typical recommendation is to pass a single configuration object that encapsulates multiple configuration options, known as parameter objects (see, Fowler 1999, pp. 295ff).

Finally, with the parameter approach, variability-related code crosscuts the remaining implementation (see *crosscutting concerns* in Sect. 3.2.3, p. 55). Feature code is scattered across multiple files and modules, in variables, in method arguments, in if statements, and so forth. Furthermore, feature code is tangled with the base code and code of other features. Due to the crosscutting nature, it is difficult to encapsulate a feature's code behind an interface and to place all feature code in one cohesive structure (see *information hiding* in Sect. 3.2.4, p. 57). Due to the scattering and lack of cohesion, it can be nontrivial to trace a feature to all code fragments implementing it (see *feature traceability* in Sect. 3.2.2, p. 54): Unless specific conventions are used, one has to follow the control flow, possible assignments to other variables, and possible operations on the configuration parameters. The parameter approach can lead to undisciplined ad-hoc implementations that are difficult to analyze, maintain, and debug.

Summary parameters
Strong points:

- Easy to use, well-known.
- Flexible and fine grained (see Sect. 3.2.5, p. 59).
- First-class programming-language support (see Sect. 3.2.6, p. 60).
- Different configurations within the same program possible.

Weak points:

- All code is deployed, no compile-time configuration (see Sect. 3.1.1, p. 48).
- Often used in ad-hoc or undisciplined fashion.
- Boilerplate code or nonmodular solutions.
- Scattering and tangling of configuration knowledge (see Sect. 3.2.3, p. 55).
- Separation of feature code and information hiding possible, but left to the discipline of developers (see Sects. 3.2.3 and 3.2.4, p. 55 and 57).
- Extensions typically require invasive changes, but little preplanning though (see Sect. 3.2.1, p. 53).
- No support for noncode artifacts (see Sect. 3.2.6, p. 60).
- Nontrivial tracing between features and code (see Sect. 3.2.2, p. 54), and thus difficult to analyze statically (see Sect. 10.2.3, p. 257).

4.2 Design Patterns

A problem of the parameter approach is that variability is scattered and hard-wired in the source code, often in an undisciplined fashion. Consequently, many patterns have evolved on how variability can be separated and decoupled, among them many

well-known *design patterns* for object-oriented programming, such as *observer*, *template method*, *strategy*, *decorator* (Gamma et al. 1995).

Definition 4.1. *Design patterns* are descriptions of collaborating objects and classes that are customized to solve a general design problem in a particular context.

(Gamma et al. 1995) □

Because design patterns are already a part of many curricula, they are increasingly common in practice, and there are many excellent descriptions (most prominently, Gamma et al. 1995), we describe only four design patterns briefly that are well-suited for variability implementations.

4.2.1 Observer Pattern

The *observer pattern* (also known as publish/subscribe pattern) describes a common way to implement distributed event handling, in which a subject notifies all registered observers of changes to its state. The observer pattern decouples subject and observers and adds flexibility to add or remove observers later. For example, a data store (the subject) could notify user-interface elements such as tables, charts, and alerts (the observers) whenever its data changes, independent of how many user-interface elements currently display the data.

The observer pattern consists of three roles: (a) An *observer interface*, which contains one or more methods that are invoked by the subject upon state changes or other events. (b) *Concrete observers*, implementing the observer interface and reacting to changes by the subject. (c) A *subject*, to which observers can register themselves. A subject dispatches events to all registered observers. In Fig. 4.2, we illustrate the architecture and a small schematic implementation.

A subject broadcasts events to all registered observers. Instead of hardwiring the notification mechanism, developers can flexibly add and remove observers at runtime. The subject does not need to know all observers; in fact, the subject only knows the observer interface and has a (dynamically-changing) list of observers.

In product-line development, the observer pattern makes it easy to add and remove features, provided that a feature can be implemented as an observer. Each feature implements the observer interface and registers itself with the subject for relevant events, such as opening a file, sending a mail, committing a transaction, or printing the nodes of a graph. Variability is achieved by registering or not registering observers. Code of different features can be separated into distinct observer classes (see *separation of concerns* in Sect. 3.2.3, p. 55). This way, additional features can be added without changing the implementation of a subject (the part that is common to all products of the product line).

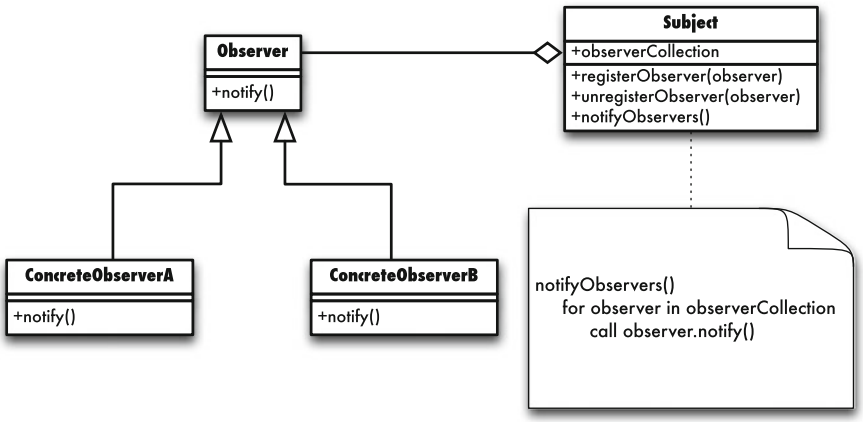


Fig. 4.2 Observer pattern

The observer pattern requires *preplanning* (see Sect. 3.2.1, p. 53). A developer needs to decide upfront where variation will be needed later and to prepare the code, by providing a registration facility and exposing relevant information through the observer interface. Extensions can only be added without invasive modifications of the base code, when the observer pattern was prepared in the base code. Furthermore, additional indirections (for example, calling notifyObservers) cause (a small) architectural run-time overhead, even if no observers are added.

In Fig. 4.3, we adapt the observer pattern to decouple feature code inside the print methods of our graph example. The base implementation of the graph acts as subject, and edges issue notifications when they are printed. These notifications are consumed by the observers WeightPrintObserver and ColorPrintObserver, which implement the parts of the features Weighted and Colored, respectively. Note that the observer mechanism is general; we could use it to implement also other features than printing at the same extension point without changing the base code.

4.2.2 Template-Method Pattern

The *template-method pattern* defines a skeleton of an algorithm in an abstract class, but leaves certain steps of the algorithm to be specified by a subclass. Different subclasses can provide different implementations of these steps by overriding one or multiple methods, and can thus influence program behavior. The pattern exploits subtype polymorphism and late binding in object-oriented programming to execute always the most specific implementation of each method. The template-method pattern is the core mechanism for implementing white-box frameworks, discussed later in Sect. 4.3.1.

```
1  class Graph {
2    ...
3    List<IPrintObserver> observers =
4          new ArrayList<IPrintObserver>();
5    void register(IPrintObserver o) {
6      observers.add(o);
7    }
8    void unregister(IPrintObserver o) {
9      observers.remove(o);
10   }
11   void notifyBeforePrint(Edge e) {
12     for (IPrintObserver o : observers)
13       o.beforePrint(e);
14   }
15   void notifyAfterPrint(Edge e) {
16     for (IPrintObserver o : observers)
17       o.afterPrint(e);
18   }
19 }
20
21 interface IPrintObserver {
22   void beforePrint(Edge edge);
23   void afterPrint(Edge edge);
24 }
```

```
25 class Edge {
26   Node a, b;
27   Color color = new Color();
28   Weight weight;
29   Graph graph;
30   Edge(Node _a, Node _b, Graph _g) {
31     a = _a; b = _b; graph = _g;
32   }
33   void print() {
34     graph.notifyBeforePrint(this);
35     System.out.print(" (");
36     a.print();
37     System.out.print(" , ");
38     b.print();
39     System.out.print(") ");
40     graph.notifyAfterPrint(this);
41   }
42
43 }
44
45 class WeightPrintObserver
46       implements IPrintObserver {
47   public void beforePrint(Edge edge) { }
48   public void afterPrint(Edge edge) {
49     edge.weight.print();
50   }
51 }
52
53 class ColorPrintObserver
54       implements IPrintObserver {
55   public void beforePrint(Edge edge) {
56     Color.setDisplayColor(edge.color);
57   }
58   public void afterPrint(Edge edge) { }
59 }
```

Fig. 4.3 Graph library: Variability in method print implemented following the observer pattern

Implementations of this pattern are straightforward in Java: We implement the algorithm skeleton as one or more methods in an abstract class. For invoking behavior that is subclass-specific, we call corresponding *abstract methods*. Alternatively, we could provide default behavior in virtual but concrete methods that may be overridden by a subclass. Subsequently, a subclass extends the abstract class and provides custom behavior. Different subclasses can provide different specific behaviors, but all share the overall implementation skeleton of the algorithm.

In product-line development, we can exploit this pattern and implement behavior of alternative features by means of different subclasses. Especially, if the algorithm differs only in minor details in each feature, we can share the common parts of the algorithm in a common abstract class. In Fig. 4.4, we illustrate how to implement weighted and unweighted graphs in an excerpt of our graph example. Note that, beyond just overriding existing methods, a subclass can also introduce additional behavior, as with the additional method add in class WeightedGraph.

The template-method pattern is best suited for alternative features (that is, when only one feature out of a set of features can be selected at a time, see Sect. 2.3, p. 26). Also individual optional features can be implemented, when a default implementation is provided for each template method. However, the template-method pattern is not

```
1  abstract class Graph {
2    Vector nodes = new Vector();
3    Vector edges = new Vector();
4    Edge add(Node n, Node m) {
5      Edge e = createEdge(n, m);
6      nodes.add(n);
7      nodes.add(m);
8      edges.add(e);
9      return e;
10   }
11   protected abstract Edge createEdge(Node n, Node m);
12   ...
13 }
14
15 class UnweightedGraph extends Graph {
16   protected Edge createEdge(Node n, Node m) {
17     return new Edge(n, m);
18   }
19 }
20
21 class WeightedGraph extends Graph {
22   protected Edge createEdge(Node n, Node m) {
23     WeightedEdge e = new WeightedEdge(n, m);
24     e.weight=new Weight();
25     return e;
26   }
27   Edge add(Node n, Node m, Weight w) {
28     WeightedEdge e = (WeightedEdge) createEdge(n, m);
29     e.weight = w;
30     nodes.add(n);
31     nodes.add(m);
32     edges.add(e);
33     return e;
34   }
35 }
```

Fig. 4.4 Graph library: Variability between weighted and unweighted graphs with the template-method pattern

suited for combining multiple features, due to limitations of inheritance (see the discussion in Fig. 4.9, p. 78).

Similar to the other patterns, the template-method pattern separates feature code from base code (see *separation of concerns* in Sect. 3.2.3, p. 55). Feature code is placed in distinct classes and induces a moderate run-time overhead, due to additional invocations of virtual methods. Some authors classify variation through the template-method pattern as a distinct implementation strategy 'inheritance' or 'subtype polymorphism' (Anastasopoules and Gacek 2001; Muthig and Patzke 2002).

4.2.3 Strategy Pattern

The *strategy pattern* aims at variability in algorithms, similar to the template-method pattern. The strategy pattern is different in that it uses delegation instead of inheritance. Instead of writing an abstract method to be overridden by clients, a

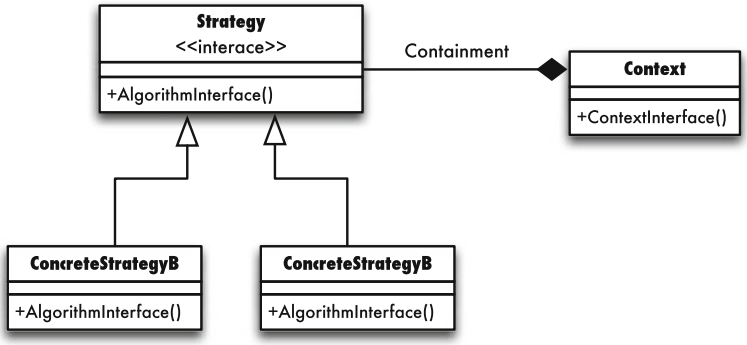


Fig. 4.5 Strategy pattern

developer specifies a strategy interface that is implemented by clients. The strategy pattern encodes a *callback* mechanism and is the core mechanism for implementing black-box frameworks, as discussed in Sect. 4.3.2.

The strategy pattern consists of three roles, as illustrated in Fig. 4.5: The *context* that implements the main algorithm (comparable to the abstract class in the template-method pattern, or the subject in the observer pattern); a *strategy interface* that describes the functionality that can be provided by clients (similar to the observer interface); and *concrete implementations* of the strategy. A strategy is passed to the context in some form, for example, as a constructor parameter, with a setter method, or as method argument; the context may call the strategy's methods.

Using the strategy pattern, a client can select which implementation of the strategy should be used during the execution. In this way, it is easy to add subsequently new implementations.

In product-line development, the strategy pattern is well-suited to implement alternative features, provided that features correspond to different implementations of methods. The pattern replaces ad-hoc conditional statements in the source code with polymorphic calls to the strategy interface. The language dispatches the method call to the concrete strategy implementation. Implementing features as a strategy encourages programmers to encapsulate features with interfaces in a disciplined form (see *information hiding* in Sect. 3.2.4, p. 57). Developers can precisely specify the interface for alternative implementations and future variations. In Fig. 4.6, we show the weighted-versus-unweighted decision from the template-method example, implemented using the strategy pattern.

Although the strategy pattern is best suited for alternative features, developers can also encode optional features. To that end, developers provide default or dummy implementations of strategies for deselected features or accept *null* as strategy parameter. Finally, combining multiple features is possible, if prepared accordingly: we could accept multiple strategies and execute them all (and potentially pass the result of one strategy as input for the next); we show an example later with filter plug-ins in Sect. 4.3.3.).

```
1  interface IEdgeStrategy {
2    Edge createEdge(Node n, Node m);
3  }
4
5  class Graph {
6    final private IEdgeStrategy strategy;
7    Vector nodes = new Vector();
8    Vector edges = new Vector();
9
10   Graph(IEdgeStrategy _strategy) {
11     strategy = _strategy;
12   }
13   Edge add(Node n, Node m) {
14     Edge e = strategy.createEdge(n, m);
15     nodes.add(n);
16     nodes.add(m);
17     edges.add(e);
18     return e;
19   }
20   ...
21 }
22
23 class WeightedEdgeStrategy implements IEdgeStrategy {
24   public Edge createEdge(Node n, Node m) {
25     WeightedEdge e = new WeightedEdge(n, m);
26     e.weight = new Weight();
27     return e;
28   }
29 }
30
31 class UnweightedEdgeStrategy implements IEdgeStrategy {
32   public Edge createEdge(Node n, Node m) {
33     return new Edge(n, m);
34   }
35 }
```

Fig. 4.6 Graph library: Weighted and unweighted graph variability with the strategy pattern

The strategy pattern encourages encapsulation and decoupling of features, and even enables distributed development and separate compilation. We discuss benefits and drawbacks of variability with the strategy pattern in more detail in the context of frameworks, in Sect. 4.3.5.

4.2.4 Decorator Pattern

The *decorator pattern* (also known as the wrapper pattern) describes a delegation-based mechanism to flexibly extend objects with additional behavior. Decorators enable objects of a (prepared) class to be extended with additional behavior at run time. Multiple extensions can be combined. The delegation-based decorator pattern can elegantly solve some composition problems that are problematic with inheritance. It can be seen as a dynamic and restricted form of mixin composition (see also Fig. 4.9 and Sect. 6.1.3).

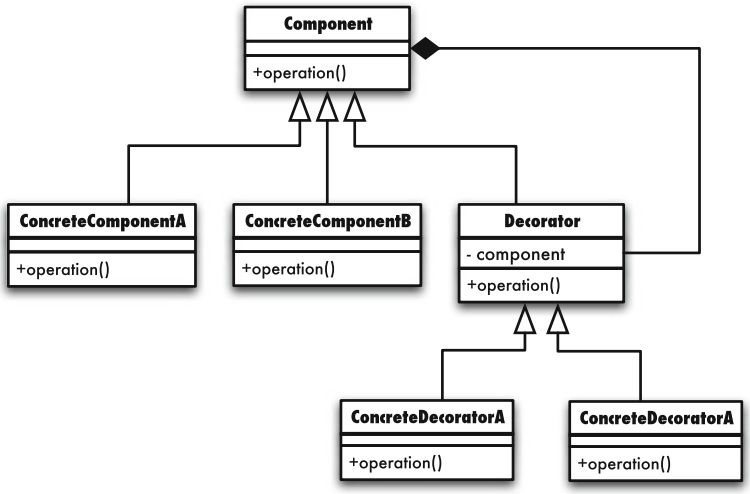


Fig. 4.7 Decorator pattern

According to the terminology of the decorator pattern, the extensible class is called a *component*. The decorator pattern consists of four roles, as illustrated in Fig. 4.7: The *component interface* that describes the (extensible) behavior of the component, a concrete implementation of the *component*, optionally, an abstract *decorator class*, and one or more concrete *decorator implementations*. The concrete component implementations and all decorators implement the component interface. The decorators receive a component as a constructor argument and forward all calls to that component, except for the intended changes to the behavior. For example, a decorator can intercept selected method invocations, whereas it forwards all remaining invocations to the decorated component. Decorators are added to a component object o by wrapping around it (for example, o = **new** DecoratorA(o);).

The strength of decorators is that additional behavior can be added incrementally at run time to existing classes. Furthermore, a series of decorators can wrap the same class, integrating the wrapper and class functionality. To the outside world, the class always provides the same interface, decorated or not. Probably the best known application of the decorator pattern in the Java world is streams in Java's standard library. Input streams all share a common interface, but there are multiple concrete implementations, such as ByteArrayInputStream and FileInputStream. The concrete implementations can be extended with several optional decorators, such as BufferedInputStream, CipherInputStream, and AudioInputStream. A developer can flexibly select core implementation and decorators, even at run-time. For example, in the following code fragment each decorator adds additional functionality to the methods of FileInputStream:

```
InputStream str = new  BufferedInputStream(
                              new CipherInputStream(
                                   new FileInputStream(file)));
```

```
1  class Graph {
2    Vector nodes = new Vector();
3    Vector edges = new Vector();
4    IEdge add(Node n, Node m) {
5      IEdge e = new Edge(n, m);
6      if (Conf.COLORED)
7        e=new ColoredEdge(e);
8      if (Conf.WEIGHTED)
9        e=new WeightedEdge(e);
10     nodes.add(n);
11     nodes.add(m);
12     edges.add(e);
13     return e;
14   }
15   ...
16 }
17
18 interface IEdge {
19   void print();
20 }
21
22 class Edge implements IEdge {
23   Node a, b;
24   Edge(Node _a, Node _b){a=_a; b=_b;}
25   public void print() {
26     System.out.print(" (");
27     a.print();
28     System.out.print(" , ");
29     b.print();
30     System.out.print(") ");
31   }
32 }
```

```
33 abstract class EdgeDecorator
34       implements IEdge {
35   protected IEdge edge;
36   EdgeDecorator(IEdge _edge) {
         edge=_edge; }
37   public void print() {
38     edge.print();
39   }
40 }
41
42 class ColoredEdge extends EdgeDecorator {
43   Color color = new Color();
44   ColoredEdge(IEdge edge) {
45     super(edge);
46   }
47   public void print() {
48     Color.setDisplayColor(color);
49     super.print();
50   }
51 }
52
53 class WeightedEdge extends EdgeDecorator{
54   Weight weight;
55   WeightedEdge(IEdge edge) {
56     super(edge);
57     weight=new Weight();
58   }
59   public void print() {
60     super.print();
61     weight.print();
62   }
63 }
```

Fig. 4.8 Graph library: Decorators for the features Weighted and Colored

In product-line development, the decorator pattern is well-suited to implement optional features and feature groups of which multiple features can be selected. In Fig. 4.8, we illustrate decorators by implementing extensions of the features Weighted and Colored to class Edge (similar to the observer-pattern example, in Fig. 4.3). Note that we hard-code the installation of decorators in method addEdge depending on configuration parameters. Of course, we could use also the strategy pattern or other mechanisms to customize which decorators to install. We could even add decorators to existing objects in a running system.

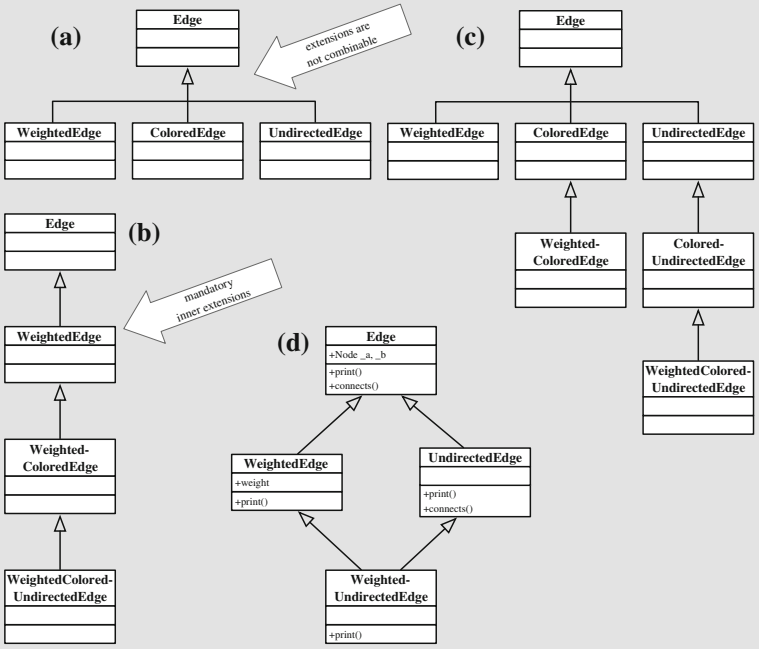
4.2.5 Discussion

Design patterns offer general solutions to reoccurring design problems. Implementing variability is a reoccurring problem, and several patterns provide guidance, as we have shown. Design patterns provide building blocks that are often adopted, used combined, or used in a larger context, for example, as part of frameworks, which we discuss in Sect. 4.3.

The advantages and drawbacks of design patterns are similar to those of the parameter solution, discussed in Sect. 4.1.1; here, we focus on the differences. In general,

Limitations of Inheritance: With inheritance, developers can decouple optional or alternative extensions from the base class. However, in product-line development, inheritance can quickly become limiting when multiple features should be combined.

Consider the following example from the graph library. We want to separate class `Edge` from its extensions `WeightedEdge`, `ColoredEdge`, and `DirectedEdge`. A first possibility (Example a) is to let all three extensions inherit directly from class `Edge`; however, now we cannot combine the extensions, for example, to create edges that are both colored and weighted. Alternatively, we could let all extensions form a linear inheritance hierarchy (Example b). Still, we cannot freely combine extensions; we have to create chains of subclasses for every combination of extensions (Example c), leading to massive code replication (Smaragdakis and Batory, 2002).



Multiple inheritance can offer a possible way out: A class can inherit from multiple superclasses. As illustrated in Example d, we could create a class `WeightedColoredEdge` that inherits both from `WeightedEdge` and `ColoredEdge`. However, if both superclasses have changed the same method, we run into the well-known *diamond problem* (Snyder, 1986): Which of the two possible super methods should be invoked. Due to the diamond problem, multiple inheritance is supported only in few languages.

Another solution is *mixin composition*: a restricted form of multiple inheritance. Mixin composition receives increasing attention with support in languages such as Scala and Ruby, but is not available in mainstream languages such as Java or C++. We will see work-arounds to the limitations of inheritance in Chapter 6.

Fig. 4.9 Limitations of Inheritance with regard to compositional flexibility

design patterns provide good-practice guidelines for disciplined implementations of variability. They are well-known and facilitate communication between developers. In contrast to native ad-hoc implementations with parameters, they encourage decoupling and encapsulation of features (see *separation of concerns* and *information hiding* in Sects. 3.2.3 and 3.2.4, p. 55 and 57) and support a clear tracing of features to observers, subclasses, strategies, decorators and others (see *feature traceability* in Sect. 3.2.2, p. 54). Design patterns enable noninvasive future extensions without changing the base implementation, at the cost of some preplanning effort (see *preplanning effort* in Sect. 3.2.1, p. 53). Since the patterns describe only designs—not concrete code snippets—they can be encoded in different programming languages, but not in noncode languages (*uniformity* in Sect. 3.2.6, p. 60). However, most implementations of design patterns add boilerplate code and architectural overhead, which may influence binary size and performance negatively.

All design patterns discussed here enable variability at run-time (see *binding times* in Sect. 3.1.1, p. 48). For some languages, there are also encodings of these patterns that allow compile-time specialization (such as inlining and static binding) for cases when the configuration choice is already known at compile-time. For example, Czarnecki and Eisenecker (2000, Chap. 7) discuss how to encode design patterns efficiently with generic-programming techniques in C++.

Summary design patterns. Similar to parameters (see Sect. 4.1.1, p. 66), but with the following distinctions.

Strong points:

- Well established, ease communication between developers.
- Guidelines for disciplined design.
- Separate feature code from base code (see Sect. 3.2.3, p. 55), possibly with clear interfaces (see Sect. 3.2.4, p. 57).
- Noninvasive extensions without modifying the base code, given a preplanning effort (see Sect. 3.2.1, p. 53).

Weak points:

- Boilerplate code and architectural overhead.
- Preplanning of extension points necessary (see Sect. 3.2.1, p. 53).

4.3 Frameworks

A *framework* is an incomplete set of collaborating classes that can be extended and tailored for a specific use case. It represents a reusable base implementation for a related set of problems, and thus perfectly fits the needs of product-line development.

A framework provides explicit points for extensions, called *hot spots*, at which developers can extend the framework. Often, extensions are called *plug-ins*. In the same manner as the template-method design pattern (see Sect. 4.2.2, p. 71) and the strategy design pattern (see Sect. 4.2.3, p. 73), a framework is responsible for the main control flow and asks its extensions for custom behavior, on demand; a principle called *inversion of control* (Johnson and Foote 1988).

Nowadays, frameworks with plug-ins are popular in end-user software, including web browsers, graphics-editing software, media players, and *integrated development environments (IDEs)*. For example, the Eclipse IDE is a framework (actually a set of many frameworks) that can be tailored with thousands of plug-ins (Gamma and Beck 2003). In Eclipse and all other frameworks, the basic application is extensible with specific plug-ins. Furthermore, plug-ins are often developed and compiled independently by third-party developers.

In feature-oriented product-line development, ideally, we develop one plug-in per feature and configure the application by assembling and activating plug-ins corresponding to the feature selection—a composition process (see *annotation versus composition* in Sect. 3.1.3, p. 50).

Definition 4.2. A *framework* is a set of classes that embodies an abstract design for solutions to a family of related problems, and supports reuse at a larger granularity than classes. A framework is open for extension at explicit *hot spots*.

(Johnson and Foote 1988) □

Although historically frameworks predate design patterns, technically, they can be best explained with the design patterns introduced in the previous section. Researchers distinguish between two kinds of frameworks: white-box and black-box (Johnson and Foote 1988). The latter are well-known for using plug-ins.

4.3.1 White-Box Frameworks

White-box frameworks consist of a set of concrete and abstract classes. To customize their behavior, developers extend white-box frameworks by overriding and adding methods through *subclassing*. A white-box framework can be best thought of as a class containing one or more template-methods (see Sect. 4.2.2, p. 71) that developers implement or overwrite in a subclass (actually, it consists of multiple classes).

The "white-box" in white-box framework comes from the fact that developers need to understand the framework's internals. Developers need to identify template methods and can additionally override all other accessible methods. Extensions in white-box frameworks can usually directly access the state of superclasses. All non-

private fields and methods can be regarded as hot spots of the framework. Extensions in white-box frameworks are usually compiled together with the framework code.

On the one hand, the ability to override existing behavior provides additional flexibility to implement unforeseen extensions. On the other, white-box frameworks require detailed understanding of internals and do not clearly encapsulate extensions from the framework; thus, they are criticized for neglecting modularity.

Typical examples of white-box frameworks are libraries of graphical user interfaces, such as *Swing* or *MFC*, and extensible compilers, such as *abc* or *Polyglot*. We provide a concrete code example, after discussing black-box frameworks.

When using white-box frameworks for product-line variability, we can only add one subclass at a time to a given class, but not mix and match multiple extensions (as explained for the *template-method design pattern* in Sect. 4.2.2). Hence, as the template-method design pattern, white-box frameworks are best suited for implementing alternative features.

4.3.2 Black-Box Frameworks

Black-box frameworks separate framework code and extensions through interfaces. An extension of a black-box framework can be separately compiled and deployed and is typically called a *plug-in*. In feature-oriented product-line development, ideally, each feature is implemented by a separate plug-in.

Definition 4.3. A *plug-in* extends hot spots of a black-box framework with custom behavior. A plug-in can be separately compiled and deployed. □

Whereas white-box frameworks can be understood in terms of the template-method pattern, black-box frameworks follow the strategy and observer patterns (see Sect. 4.2, p. 69). The framework exposes explicit hot spots, at which plug-ins can register observers and strategies. That is, instead of subclassing, black-box frameworks register objects and callback functions. As discussed for the corresponding design patterns, it is possible to provide hot spots that can be extended with multiple plug-ins.

Black-box frameworks are called "black-box" because, ideally, developers need to understand merely their interfaces, but not the internal implementation of the framework. In contrast to a white-box framework, the interface (set of hot spots) of a black-box framework is explicit. Extensions can access only state of the framework that exposed in the interface. Developers can add extensions only to hot spots foreseen (or preplanned) by the framework developer. Although restricting extensions to an interface may limit flexibility, it enables a strict decoupling of framework code and extension code. Furthermore, it can make the framework easier to understand and use, because only a comparably small amount of interface code must be understood.

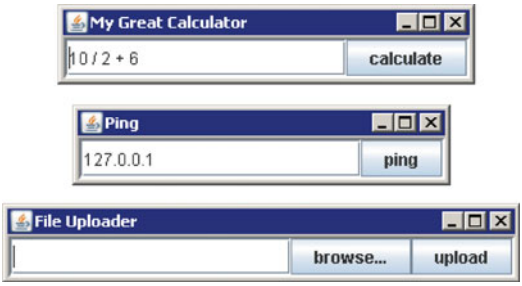


Fig. 4.10 Three similar applications (calculator, ping, and file loader) that can be implemented on *top* of a common framework

The decoupling of extensions encourages separate development and independent deployment of plug-ins, as known from many application-software frameworks, including web-browsers or development environments. As long as the plug-in interfaces remain unchanged, framework and plug-ins can evolve independently.

4.3.3 An Implementation Example for Frameworks

We illustrate the implementation of white-box and black-box frameworks in Java by means of a small example. In Fig. 4.10, we show screenshots of three applications that perform different tasks (calculator, ping, and file loader), but have a similar (trivial) user interface. Their implementations share a relatively large amount of source code for initializing the user interface (fields, buttons, and layout), for starting and stopping the application, and so forth. From the code of the calculator application in Fig. 4.11, only the underlined code fragments differ between the applications, the rest is shared. We demonstrate how to implement the common behavior in a framework and extend it with specific plug-ins, for example, to get the three applications.

A white-box framework is shown in Fig. 4.12: We replace variable code fragments by abstract methods (or overridable methods with default implementations), following the template-method pattern (see Sect. 4.2.2, p. 71). For each extension, we create a subclass and implement the abstract methods to specify custom behavior. The extensions can directly access protected and public methods of the framework, such as getInput in Line 44.

A black-box framework of the same design is listed in Fig. 4.13. Here, we decouple framework and extensions (plug-ins) with an interface Plugin. The extension does not subclass the framework, but implements only the interface. Note the similarity to the strategy pattern (see Sect. 4.2.3, p. 73): The interface Plugin represents a strategy interface called from the context in class App, whereas class CalcPlugin is a concrete strategy.

```
1  class Calc extends JFrame {
2    private JTextField textfield;
3    public static void main(String[] args) { new Calc().setVisible(true); }
4    public Calc() { init(); }
5    protected void init() {
6      JPanel contentPane = new JPanel(new BorderLayout());
7      contentPane.setBorder(new BevelBorder(BevelBorder.LOWERED));
8      JButton button = new JButton();
9      button.setText("calculate");
10     contentPane.add(button, BorderLayout.EAST);
11     textfield = new JTextField("");
12     textfield.setText("10 / 2 + 6");
13     textfield.setPreferredSize(new Dimension(200, 20));
14     contentPane.add(textfield, BorderLayout.WEST);
15     button.addActionListener(/* code to calculate */);
16     this.setContentPane(contentPane);
17     this.pack();
18     this.setLocation(100, 100);
19     this.setTitle("My Great Calculator");
20     // code for closing the window
21   }
22 }
```

Fig. 4.11 Example code of the calculator application. Specifics for the calculator are highlighted

```
1  abstract class App extends JFrame {
2    protected abstract String
         getApplicationTitle();
3    protected abstract String
         getButtonText();
4    protected String getInititalText() {
5      return "";
6    }
7    protected void buttonClicked() { }
8    private JTextField textfield;
9    public App() { init(); }
10   protected void init() {
11     JPanel contentPane =
12         new JPanel(new BorderLayout());
13     contentPane.setBorder(new
14         BevelBorder(BevelBorder.LOWERED));
15     JButton button = new JButton();
16     button.setText(getButtonText());
17     contentPane.add(button,
           BorderLayout.EAST);
18     textfield = new JTextField("");
19     textfield.setText(getInititalText());
20     textfield.setPreferredSize(
21         new Dimension(200, 20));
22     contentPane.add(textfield,
           BorderLayout.WEST);
23     button.addActionListener(
24         ... buttonClicked(); ...);
25     this.setContentPane(contentPane);
26     this.pack();
27     this.setLocation(100, 100);
28     this.setTitle(getApplicationTitle());
29     // code for closing the window
30   }
31   protected String getInput() {
32     return textfield.getText();
33   }
34 }
```

```
35 class Calculator extends App {
36   protected String getButtonText() {
37     return "calculate";
38   }
39   protected String getInititalText() {
40     return "(10 - 3) * 6";
41   }
42   protected void buttonClicked() {
43     JOptionPane.showMessageDialog(this,
44         "The result of " + getInput() +
45         " is " + calculate(getInput()));
46   }
47   private String calculate(String input){
48     ...
49   }
50   protected String getApplicationTitle(){
51     return "My Great Calculator";
52   }
53   public static void main(String[] args){
54     new Calculator().setVisible(true);
55   }
56 }
```

```
57 class Ping extends App {
58   protected String getButtonText() {
59     return "ping";
60   }
61   protected String getInititalText() {
62     return "127.0.0.1";
63   }
64   ...
65   public static void main(String[] args){
66     new Ping().setVisible(true);
67   }
68 }
```

Fig. 4.12 White-box framework for single-button applications and two extensions of the framework

```
1  interface Plugin {
2    String getAppTitle();
3    String getButtonText();
4    String getInititalText();
5    void buttonClicked() ;
6    void register(InputProvider app);
7  }
8  interface InputProvider {
9    String getInput();
10 }
```

```
11 class App extends JFrame
12           implements InputProvider {
13   private JTextField textfield;
14   private Plugin plugin;
15   public App(Plugin p) {
16     this.plugin=p;
17     p.register(this);
18     init();
19   }
20   protected void init() {
21     JPanel contentPane =
22         new JPanel(new BorderLayout());
23     contentPane.setBorder(new
24         BevelBorder(BevelBorder.LOWERED));
25     JButton button = new JButton();
26     button.setText(plugin.getButtonText());
27     contentPane.add(button,
           BorderLayout.EAST);
28     textfield = new JTextField("");
29     textfield.setText(
30         plugin.getInititalText());
31     textfield.setPreferredSize(
32         new Dimension(200, 20));
33     contentPane.add(textfield,
           BorderLayout.WEST);
34     button.addActionListener(
35         ... plugin.buttonClicked(); ...);
36     this.setContentPane(contentPane);
37     //...
38   }
39   public String getInput() {
40     return textfield.getText();
41   }
42 }
```

```
43 class CalcPlugin implements Plugin {
44   private InputProvider ip;
45   public void register(InputProvider i) {
46     this.ip = i;
47   }
48   public String getButtonText() { return
         "calculate"; }
49   public String getInititalText() {
         return "10 / 2 + 6"; }
50   public void buttonClicked() {
51     JOptionPane.showMessageDialog(null,
52       "The result of " +
53       ip.getInput() + " is " +
54       calculate(ip.getInput())); }
55   public String getAppTitle() { return
         "My Great Calculator"; }
56   private String calculate(String m) ...
57 }
```

```
58 class CalcStarter {
59   public static void main(String[] args){
60     new App(new CalcPlugin()).
61         setVisible(true);
62   }
63 }
```

Fig. 4.13 Black-box framework for single-button applications and a plug-in

Recall, in a black-box framework, the extension cannot access internals of the framework. To allow extensions accessing information from the framework, we need to provide a callback mechanism. In our example, the framework registers itself to the extension (Line 17), such that the extension can access methods from the framework (Line 53). We even decouple the callback with an additional interface InputProvider to protect what information the framework exposes. Such a callback is not always needed and must not necessarily be implemented with an additional interface as in our example.

To provide a hot spot that can be extended with multiple plug-ins, we store a list of plug-ins instead of a single plug-in (similar to the observer pattern, see Sect. 4.2.1, p. 70). Further, instead of providing a single plug-in interface for all variability, we provide multiple specialized plug-in interfaces. We illustrate both extensions

```
1  public class App {
2    private List<EncoderPlugin> encoders;
3    private List<FilterPlugin> filters;
4    public App(List<EncoderPlugin> encoders,
           List<FilterPlugin> filters) {
5      this.encoders=encoders;
6      for (EncoderPlugin plugin: encoders)
7        plugin.register(this);
8      this.filters=filters;
9    }
10   public Message processMsg (Message msg) {
11     for (EncoderPlugin plugin: encoders)
12       if (plugin.canProcess(msg))
13         msg = plugin.encode(msg);
14     boolean isVeto = false;
15     for (FilterPlugin plugin: filters)
16       isVeto = isVeto || plugin.veto(msg);
17     ...
18     return msg;
19   }
20 }
```

Fig. 4.14 Framework that supports multiple plug-ins of different kinds

in Fig. 4.14: Encoders and filters have different plug-in interfaces and the framework accepts a list of both plug-ins; all plug-ins work together to encode and filter messages.

Observe the *inversion of control* that is typical for frameworks. The framework controls the execution and only calls the extension when it requires information.

4.3.4 Loading Plug-Ins

A final question is how to load extensions, especially, plug-ins in black-box frameworks. In white-box frameworks, we simply pass the desired subclass or invoke its main method. In our black-box framework example, we passed the desired plug-in as constructor parameters to the framework from a separate starter class (class CalcStarter in Fig. 4.13). In practice, separate plug-in loaders are common.

In Fig. 4.15, we illustrate a simple plug-in loader for our black-box framework. The loader expects a command-line parameter naming the plug-in class. The loader then uses Java's reflection mechanism to dynamically load the class and instantiate the framework with it.

Beyond this simple example, a plug-in loader typically searches for plug-ins in a certain directory or loads plug-ins listed in a configuration file. Subsequently, the plug-in loader sets up the framework with the corresponding loaded plug-ins. The loader decouples framework and plug-ins even further, as the plug-in loader identifies and loads plug-ins during startup. No further code must be written to activate a plug-in, in the simplest case it is just copied to a specific directory.

Plug-in loaders may additionally check that the plug-in implements required interfaces and check dependencies or ordering constraints between plug-ins. Thus, invalid

```
1  public class Starter {
2    public static void main(String[] args) {
3      if (args.length != 1)
4        System.out.println("Plugin name not specified");
5      else {
6        String pluginName = args[0];
7        try {
8          Class<?> pluginClass = Class.forName(pluginName);
9          Plugin plugin = (Plugin) pluginClass.newInstance();
10         new App(plugin).setVisible(true);
11       } catch (Exception e) {
12         System.out.println("Cannot load plugin " + pluginName + ", reason: " + e);
13       }
14     }
15   }
16 }
```

Fig. 4.15 Simple plug-in loader using the Java reflection API

plug-in combinations can be rejected at load-time. End-user applications also often provide sophisticated mechanisms to install, update, deactivate, or configure plug-ins, often with graphical front-ends.

4.3.5 Discussion

Frameworks, especially, black-box frameworks, are a suitable way to implement variability in product lines. Using typical plug-in loaders, individual products are created by *composing* plug-ins at *load-time*, but in principle run-time changes are possible (see *binding times* and *annotation versus composition* in Sects. 3.1.1 and 3.1.3, p. 48 and 50). Much like design patterns, frameworks are general in that they can be implemented in most programming languages, but not in noncode languages (XML, documentation, and so forth; see *uniformity* in Sect. 3.2.6, p. 60).

When features are implemented as plug-ins, we can trace them directly (see *traceability* in Sect. 3.2.2, p. 54). With plug-ins in black-box frameworks, we can encode alternative as well as optional features in a disciplined way. We can even combine multiple optional features, as illustrated in Fig. 4.14. Developing one plug-in per feature allows us to flexibly select other features and load the corresponding plug-ins. The entire process from a feature selection to a tailored program can be automated. Plug-ins for deselected features do not need to be deployed, so we can potentially reduce binary size. Also white-box frameworks can be used to implement alternative features or a single optional feature, but combining multiple optional features is problematic due to the limitations of subclassing, as discussed in Fig. 4.9 (p. 78).

In contrast to the parameter approach (Sect. 4.1) and an ad-hoc use of design patterns (Sect. 4.2), black-box frameworks facilitate modularity by following well-defined conventions (see *separation of concerns* in Sect. 3.2.3, p. 55). Plug-ins are encapsulated from the framework implementation through clear interfaces. Ideally, an interface is designed not only for a specific extension in mind, but for a whole set of potential extensions. Framework and plug-ins can be changed independently as long as they still adhere to the common interface. In the best case, all code related

to a feature is encapsulated in a single plug-in, and it is possible to understand and maintain the feature by looking only at this plug-in's code (see *information hiding* in Sect. 3.2.4, p. 55).

Modularity allows developers to provide third-party plug-ins that can be compiled and deployed independently. This is especially important for *software ecosystem* (Bosch 2009), in which a community of specialized companies or independent developers provides additional features. Such a development model works both for open-source projects and closed-source projects. Well-known examples of are web browsers and development environments, such as *Eclipse* and *Visual Studio*, both consisting of multiple (mostly black-box) frameworks. For example, users of Eclipse can select from many independently developed open source and commercial plug-ins.

However, frameworks are not without difficulties. Creating and maintaining frameworks is a challenging task. The framework designer must anticipate (or preplan) where hot spots are needed and design corresponding template methods or plug-in interfaces. They must *design for change*, which requires an upfront investment (see *preplanning effort* in Sect. 3.2.1, p. 53). If a framework designer chooses not to expose information that extensions need, these extensions are difficult or impossible to build (without invasive refactoring the framework). Designing a framework is often handed to senior developers, because it requires substantial experience and a deep understanding of the domain.

Once hot spots and interfaces have been fixed, they are hard to evolve: Although developers can add new hot spots to the framework, it is not possible to change the plug-in *interface* without invasively changing all existing plug-ins (some of which might be provided by third parties and not available with source code or not even known). The inflexibility to change a framework may slow down future evolution of the product line. Hence, frameworks are better suited for proactive adoption of product lines than for reactive or extractive adoption (see Sect. 2.4, p. 39).

Plug-ins may be reused in many different instantiations of a framework, but, in contrast to components (which we discuss next), they are not intended to be reused across different frameworks. It is highly unlikely that plug-ins for one framework (or product line) can be plug-ins for other frameworks (or product line). The reason is simple: every framework encodes structures, architectural conventions, and implementation details that are specific to it, and that are unlikely to be shared verbatim by any other framework.

Furthermore, frameworks induce both development and run-time overhead. Developers need to write additional code to decouple extensions from the framework, such as interface Plugin in Fig. 4.13. Even if a hot spot is not extended, additional code for the extension point is required. Hence, frameworks often require more source code, result in a larger binary size, and perform slower due to additional indirections. Often the overhead is acceptable, but not always: In high-performance and embedded scenarios, unnecessary overhead can not be tolerated. Furthermore, when a framework exhibits too many hot spots for potential extensions that are never used (a problem named *speculative generality* by Fowler (1999); a common overreaction to experiencing a too narrow framework), artificial overhead can become problematic.

Fine-grained and crosscutting features (see *granularity* and *crosscutting* in Sects. 3.2.5 and 3.2.3, p. 59 and 55) are hard to implement with frameworks. Fine-grained extensions require hot spots for minimal extensions and crosscutting features require many hot spots; both lead to disproportionally complex designs. Whereas, in our calculator example, an extension modified only four code locations, features such as the transaction subsystem in a database system affect many parts of the framework. For fine-grained and crosscutting features, the framework must expose many details of the framework. This is hard to do: interfaces become bloated making new plug-ins harder to understand and build. Then there is the problem of speculative generality mentioned earlier. When there are simply too many hot spots, the benefits of modularity diminish and other variability techniques must be considered.

Overall, frameworks are better suited for coarse extensions that extend few well-defined points in the control flow (see *granularity* in Sect. 3.2.5, p. 59). There is no technical limitation, but the overhead for implementing fine-grained and crosscutting features can become overwhelming. Features such as transaction management in a database system (crosscutting the entire implementation and changing the behavior in many locations in nontrivial ways) are rarely separated into plug-ins.

Summary frameworks. Like the parameter approach, frameworks are language-based (see Sect. 3.1.2, p. 49). However, frameworks differ from parameters in that they are primarily composition-based, not annotation-based (see Sect. 3.1.3, p. 50), and in that variability is usually decided at load-time, not run-time (see Sect. 3.1.1, p. 48).
Strong points:

- Well-suited for implementing variability.
- Automated product derivation by plug-in loading.
- Static tailoring, deploying only selected features (see Sect. 3.1.1, p. 48).
- Modularity by separating features, hiding feature internals, and enabling feature traceability (especially in black-box frameworks; see Sects. 3.2.2–3.2.4, p. 54–57).
- Suitable for open-world development (black-box frameworks only).
- Disciplined implementation, well-known.

Weak points:

- High upfront design effort (see Sect. 3.2.1, p. 53).
- Difficult evolution.
- Potential development, run-time, and size overhead.
- No reuse outside the framework.
- Unsuited for fine-grained and crosscutting features (see Sects. 3.2.5 and 3.2.3, p. 59 and 55).
- No support for noncode artifacts (see Sect. 3.2.6, p. 59).

4.4 Components and Services

As a last classic language-based implementation approach, we discuss components and services (including the notion of web service). Although component-based implementations are common in product-line practice, they lack the automation potential of feature orientation that we aim at. However, as we will see, components can be integrated to some degree with other implementation approaches. Hence, we introduce components briefly and discuss their benefits and limitations.

Definition 4.4. A *software component* is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

(Szyperski 1997) □

The key idea of a component is to form a modular, reusable unit. A component provides its functionality through an interface, whereas its internal implementation is encapsulated (also for components, there is a white-box versus black-box discussion (Szyperski 1997, Chap. 4); here, we assume black-box components). To reuse components, they are *composed* with other components in different combinations (see *annotation versus composition* in Sect. 3.1.3, p. 50). A class can be seen as a small component that can be reused in many applications; a library of graph algorithms is an example of a larger component, consisting of many classes.

Already more than three decades ago, Parnas (1979) proposed to implement product lines (back then called program families) by encapsulating changing parts and hiding their internals (see *information hiding* in Sect. 3.2.4, p. 57), so that those parts could be exchanged and removed easily. Domain analysis is essential to *design for change* and to identify and separate those parts that differ between products of a product line.

Proponents motivate the use of components for another reason: *building markets*. The idea is that developers can implement and deploy components independently, and compose components from different sources. As a consequence, developers can decide whether to implement their own components or whether to buy and reuse third-party components. Component markets open a new perspective: Developers can focus on their expertise and develop, perfect, and sell an individual component. Others can buy the component and use it in their software, instead of reimplementing the functionality or buying an entire software product that contains the desired functionality, but that is otherwise not tailored to their needs. Components facilitate a best-of-breed approach, in which developers can decide, for each subsystem, whether to buy the best (or cheapest) component from the market or to implement the functionality individually (Szyperski 1997).

```
1  package modules.colorModule;
2
3  //public interface
4  public class ColorModule {
5    public Color createColor(int r, int g, int b) { /* ... */ }
6    public void printColor(Color color) { /* ... */ }
7
8    public void mapColor(Object elem, Color col) { /* ... */ }
9    public Color getColor(Object elem) { /* ... */ }
10
11   //just one module instance
12   public static ColorModule getInstance() { return module; }
13   private static ColorModule module = new ColorModule();
14   private ColorModule() { super(); }
15 }
16 public interface Color { /* ... */ }
17
18 //hidden implementation
19 class ColorImpl implements Color { /* ... */ }
20 class ColorPrinter { /* ... */ }
21 class ColorMapping { /* ... */ }
```

Fig. 4.16 Simple example of component managing colors

Of course, the fallacy here is that components are plug-compatible only if they and their interfaces are designed to existing standards. In fact, reusing components from markets is often problematic, because their architectural assumptions mismatch (Garlan et al. 1995). Unless planned together, components are likely incompatible and require significant engineering effort to compose. Domain engineering improves this picture, as we will discuss shortly.

Services, as discussed in the context of service-oriented architecture (Erl 2005), are a special form of software components. A service encapsulates functionality behind an interface, just like a component. Similarly, proponents envision a market of services. Services typically emphasize standardization, interoperability, and distribution. Especially in the popular form of web services, a service is reachable over a standardized Internet protocol and may run on remote servers; that is, it is not necessary to install and integrate a service locally. Even the lookup of a service can occur at run-time, through the use of service registries. In addition, communication between services is standardized, so services written in different languages can exchange messages. To connect services, called *service orchestration*, several specialized tools and (graphical) languages exist, which simplify the composition process. For our discussion of product-line development, we make no further distinction between components and services.

Example 4.2 In Fig. 4.16, we exemplify a small component from our graph library. Assume that storing and printing colors is nontrivial and might be reused in another project outside the product line. We could extract color management into a reusable component (or package). The component's interface exposes a class ColorModule with several public methods and a Java interface Color, whereas other implementation details are hidden. We use Java's scoping and package mechanism to enforce

encapsulation. That is, to ensure that other components may not access private implementation details, we use package visibility. Classes ColorImpl, ColorPrinter, and ColorMapping are not public and visible only inside the package, so other components in other packages can only interact with public classes and methods.

A component is independent of a specific application or product line. Developers can reuse it when implementing the color feature for the graph library, but also for other applications. Note that developers need to write custom code to connect the implementation with the component, for example, extra code to call the component's methods. To reuse the component, only the public interface is of interest; implementation details are not accessible. In Java, we can deploy the component separately as class or jar files.[1] □

4.4.1 Sizing Components

Deciding when to build a reusable component and what to include in that component is a difficult design decision. There is a well-known trade-off between reuse and use (Biggerstaff 1994; Szyperski 1997): A large component that provides plenty functionality is easy to integrate and use, but there might be only few applications in which the component fits. In contrast, one might be tempted to build small reusable components that can be combined flexibly (in an extreme case, put every method into a distinct component), but, then, the overhead of using the component becomes discouraging. The smaller the component, the more programming is left to the user that has to connect the components with the base code and with each other; in extreme cases, components are so small that little remains to reuse and to hide behind their interfaces. Szyperski (1997, Chap. 4) rephrases this as the maxim "maximizing reuse minimizes use."

Example 4.3 Consider the common scenario that a developer searches the web for a piece of software. Suppose you find two choices: One that is small (less than 1000 lines of code) with only a fraction of the functionality that you want; and one that is huge (100 000 lines of code) with many extra functionalities that are highly intertwined with the desired functionality. Which one should be selected? Just on intimidation alone, many developers would select the small application as there is a higher potential to understand what it does, while the rest could still be added manually. In contrast, the big one likely makes incompatible architectural assumptions. It would require considerable integration effort and pose considerable risks. That is, most developers would likely reuse less and implement more themselves. □

[1] In our example, we use the *facade* design pattern to provide a concise interface and the *singleton* design pattern to ensure that only one instance of the component exists at at time (Gamma et al. (1995) discuss these patterns in more detail). A developer would invoke a method somewhat like this: ColorModule.getInstance().createColor(...).

In the tension between reuse and use, developers need to strive for a balance of a component that is large enough to provide useful functionality but small enough to be reused in many contexts. Unfortunately, without knowing when and how a component will be reused, even experienced developers may only guess suitable component size. Unsuitable component size has often been claimed a reason for the limited success of market places for components and services.

In product-line development, *domain analysis* solves this dilemma and guides us in how to size a component to balance reuse and use. As described in Sect. 2.2, during domain analysis, a domain expert investigates potential products within the domain and decides which products are in the scope of the product line. With this information, we can decide upfront which functionality will be reused *within* the product line; thus, we can size components accordingly and coordinate their architectural assumptions. That is, in product-line development, developers solve the sizing problem by *preplanning reuse systematically*. For example, when we know that certain functionality is always used together, we can combine it in the same component and make it easier to use; in contrast, when we know that functionality is only used in few products, it should be separated into its own component. In short, *domain analysis helps us to decide how to divide code into components*.

4.4.2 Composing Components

Developing product lines by constructing and composing reusable components was a common strategy, especially, in the early era of product-line engineering (Bass et al. 1998; Krueger 2006). With domain analysis, developers decided which functionality should be reused across multiple products of the product line and designed components accordingly. In application engineering, developers then composed the corresponding components. In principle, one could create one component per feature or map features to components in some other form.

In contrast to plug-ins in frameworks, components are generally *not* designed to be composed automatically. Component-based software engineering pursues a different mind set, in which building units of functionality is the goal; being plug-compatible with other components is typically not a priority. A common goal of the design-for-change strategy is to (potentially) exchange a component by another one with the same interface, but again automation is not the focus.

Given this, to *derive a product* for a given feature selection during application engineering (see Sect. 2.2, p. 19), a developer selects suitable components and then manually writes *glue code* to connect components for every product individually. Module interconnection languages (DeRemer and Kron 1976) and service orchestration (Erl 2005) aim at providing high-level scripting approaches to gluing together components. Although definitely a manual process, implementing product lines with components still offers a huge benefit over constructing each product from scratch. When considering the economical motivation of product lines in Fig. 1.4 (p. 9) from the first chapter, manual effort during application engineering increases costs per

product (and hence the slope), but does not change the overall expected benefit of product lines.

Building product lines with components is suitable if feature selection is performed by developers (not customers) and the number of products is low. For example, Phillips builds software for consumer electronics from reusable components (van Ommering 2002). In this case, product-line developers can construct new products from reusable parts. If required, they can also perform customizations beyond what was preplanned as product-line feature (and without propagating the customization back to the feature model). Especially, when publishing only a few distinct preconfigured products or when implementing tailor-made solutions for few customers, the manual effort in application engineering may be negligible or acceptable.

The goal of feature-oriented product-line development is to entirely *automate* product derivation after selecting features in application engineering (see *push-button approach* in Sect. 2.2.4, p. 26). In a component-based approach, there is no generator that would automatically build a product for a given feature selection; manual developer intervention is required. We argue that automated product derivation is essential for effective, large-scale product-line development. Krueger (2006) even claims "application engineering considered harmful" to emphasize the importance of automation.

4.4.3 Components Versus Plug-Ins

Components and plug-ins share many similarities. They both pursue a modular implementation (ideally a module per feature) and hide implementation details behind an interface (see *traceability* and *information hiding* in Sects. 3.2.2 and 3.2.4, p. 54 and 57) and require similar preplanning effort (see Sect. 3.2.1, p. 53).

Their main difference lies in the automation potential and in reuse beyond a product line: A plug-in is always tailored for a specific framework, and, as such, has very specific requirements on its context. In contrast, components can be intended to be reused even outside a product line. At the same time, the tight integration of plug-ins into a framework allows loading plug-ins automatically without additional per-product development. Deriving a product for a feature selection in a framework requires only assembling the corresponding plug-ins, not writing additional glue code, as necessary for components.

Of course, components can be used within frameworks. We can write glue code that adapts a general-purpose component to the plug-in interface of a framework (see also the *adapter* and *bridge* design patterns as described by Gamma et al. (1995)). This way, we can reuse components within a framework and automate their integration based on a feature selection, while we can still reuse the component in different contexts outside the framework.

Components can be encoded in many languages. Ideally, the language should provide some encapsulation mechanism that can hide internals of the component behind an interface, and that enforces the interface mechanically. However, also weaker notions of encapsulation are possible that translate also to noncode artifacts. For example, we can simply separate grammars, design documents, or models into separate artifacts and let developers combine them manually during product derivation.

Apart from automation and uniformity, as components and plug-ins use similar modularity mechanisms, they share similar benefits and limitations. They modularize features and allow compile-time product derivation, deploying only selected functionality (see *binding times* in Sect. 3.1.1, p. 48). At the same time, both components interfaces and plug-in interfaces are difficult to evolve once they are fixed and other (potentially third-party) implementations rely on them. Both may add overhead due to additional indirections and boilerplate code. Most importantly, components have the same limitations regarding fine-grained and crosscutting extensions (see *granularity* and *crosscutting* in Sects. 3.2.5 and 3.2.3, p. 59 and 55). For example, integrating a crosscutting transaction subsystem provided as component into a database will require much glue code.

Summary components and services
Strong points:

- Well-known and established implementation technique.
- Static tailoring, deploying selected features only (see Sect. 3.1.1, p. 48).
- Separation of concerns, information hiding, and feature traceability (see Sects. 3.2.2–3.2.4, p. 54–57).
- Reuse within and beyond the product line.
- Reuse of third-party implementations.
- Reuse in distributed environments, even with features maintained and run by third parties (especially services).
- Uniformly applicable to many languages (see Sect. 3.2.6, p. 60).

Weak points:

- No automated product derivation, glue code is necessary
- Difficult evolution.
- Potential development, run-time, and size overhead.
- Preplanning necessary to size components (see Sect. 3.2.1, p. 53).
- Unsuited for fine-grained and crosscutting features (see *separation of concerns* in Sect. 3.2.5, p. 59).

4.5 Further Reading

The classical language-based implementation approaches discussed in this chapter are frequently used to implement product lines, but rarely discussed explicitly as such in literature.

Especially for the parameter approach, there is little literature. Modularity and the related concepts of encapsulation and coupling are discussed generally by Meyer (1997, Chaps. 3 and 4). Regarding the problem of methods with too many parameters, Fowler (1999) discusses a corresponding code smell and a solution with a refactoring toward parameter objects. More recently, Reisner et al. (2010) and Rabkin and Katz (2011) began exploring the use of configuration options in programs (not necessarily product lines). Reisner et al. (2010) found that in three analyzed programs many configuration options are orthogonal and rarely interact. Rabkin and Katz (2011) found that configuration options are often not consistently documented, which we interpret as an encouragement to use product-line technology to plan variability in a more systematic way.

The book of Gamma et al. (1995) is still the best reference on design patterns and explains patterns in much detail. Some product-line literature (for example, Muthig and Patzke 2002; Anastaspoules and Gacek 2001) distinguishes variability implementations further into techniques based on delegation, inheritance, parametric polymorphism, and so forth. Many of them and their best practices can be explained in terms of design patterns as well. Czarnecki and Eisenecker (2000, Chap. 7) discuss low-overhead design-pattern implementations for static configuration with C++.

Frameworks have a long tradition. The seminal paper "Designing Reusable Classes" (Johnson and Foote 1988) provides an excellent introduction and discusses trade-offs between white-box and black-box frameworks. An impressive example of framework design in practice is the Eclipse development environment. Several books (and web articles) describe Eclipse's architecture and how they make use of design patterns; even though not the newest book on Eclipse, we recommend the introduction by Gamma and Beck (2003).

Finally, Szyperski (1997) provides a detailed introduction into the concept of components, including a discussion of building markets, technology choices, and how to size components. Erl (2005) provides a broad introduction into the philosophy behind service-oriented architectures. High-level languages for composing components can be traced back to the idea of programming-in-the-large by DeRemer and Kron (1976) and have been evolved since with many module interconnection languages, and more recently service-orchestration approaches (Erl 2005). Czarnecki and Eisenecker (2000) discuss in detail how domain engineering helps to preplan and size components for reuse. If not planned accordingly, reuse can be very hard though: Garlan et al. (1995) discuss architectural mismatch, the reason why composing components that have not been designed together is so difficult.

Exercises

4.1. Implement a basic chat system consisting of a server and multiple clients (in the domain discussed in Exercise 2.4, page 43). The clients show messages received from the server and allow users to post messages to the server. The server broadcasts all received messages to all connected clients.

Subsequently, extend this implementation with the following features:

(a) *Colors:* Messages may have a text and background color. The color can be specified in the client when writing a message.
(b) *Authentication:* To connect to the server, the client must provide a username and password.
(c) *Encryption:* Messages are encrypted. Provide at least two optional encryption mechanisms.
(d) *History:* Server and clients keep a log of all received messages. The client can show the last 10 entries of the log in a dialog.

Hint: Reuse existing source code where possible.

4.2. Implement all features of the chat system (Exercise 4.1) using the *parameter* approach, such that all features can be configured at load-time with command-line parameters, with a configuration file, or even with a graphical dialog in the chat client's user front-end. Critically discuss code quality and implementation effort of the resulting system.

4.3. Discuss the potential of design patterns for implementing the chat system (Exercise 4.1 and 4.2). Change the implementation where appropriate. Discuss the influence of this change on the quality criteria introduced in Chap. 3.

4.4. Find an open-source project that can be configured using configuration files or command-line parameters (for example, command-line utilities, web servers, database engines). Study the end-user documentation to find three (Boolean) configuration options that could be considered as features and investigate how those are implemented.

(a) What is the binding time of the configuration option?
(b) Is the configuration option implemented cohesively or is it scattered throughout the source code?
(c) Are global variables used or are parameters propagated? Are design patterns used in the implementation? Discuss traceability, separation of concerns, and information hiding with regard to the implementation of the configuration option.
(d) Discuss whether design patterns could be used to improve the implementation and prepare it for future extensions.

4.5. Implement the chat system (Exercise 4.1) as a framework that can be extended with plug-ins. Provide a plug-in for each feature. Ensure that framework and features can be compiled separately. Provide a simple plug-in loader (see Sect. 4.3.4) so that the configuration can be changed without any modifications to source code. Critically reflect on code quality and implementation effort of the resulting system; consider also the quality criteria from Chap. 3.

4.6. Extend the implementation of Exercise 4.5 with an additional feature *Spam Filter* that rejects all messages that contain words from a blacklist. Review your implementation: Are new extension points necessary? Is it necessary to change the framework or other plug-ins? Can the new plug-in be understood in isolation? Would the same hold for feature *Command-Line Interface* (instead of a graphical user interface) or a feature *File Transfer*?

4.7. Find a software product that is extensible with plug-ins (for example Miranda-IM,[2] Netbeans,[3] or Mozilla Firefox[4]). Study the developer documentation or source code to find possible extension points.

(a) Is the framework implemented as black-box framework, or white-box framework, or as some combination of those? Are design patterns used for extensibility?
(b) Can plug-ins be compiled separately? What mechanism is used to load plug-ins? What is the binding time?
(c) Name three features that can be added noninvasively as plug-ins using existing extension points.
(d) Name three features that cannot be added with existing extension points but would require invasive changes to the framework.

4.8. Decompose the chat application from Exercise 4.1 into reusable components. Build three different chat products out of these components.

4.9. Discuss possible components that could be reused within and beyond a product line of (a) graph algorithms, (b) chat applications, and (c) the scenarios from Exercise 2.5 (page 43). Discuss suitable size of the components and the potential costs of using them.

4.10. Reconsider the scenarios of Exercise 2.9 (page 44). Which implementation approach would you recommend to the developers and why?

4.11. Compare all discussed implementation approaches in terms of (a) modularity, (b) suitability of distributed development with multiple developers and multiple companies, (c) possibility of buying and integrating parts or features developed by third parties, (c) overhead on run-time performance, (d) overhead on binary size, (e) development effort and required skill, and (f) maintainability.

Use your implementations of the chat example (Exercises 4.2, 4.3, 4.5, and 4.8) to support your analysis.

[2] http://www.miranda-im.org/

[3] http://netbeans.org/

[4] http://www.mozilla.org/