Applying Design Patterns to the Design Model

Objectives

Upon completion of this module, you should be able to:

- Define the essential elements of a software pattern
- Describe the Composite pattern
- Describe the Strategy pattern
- Describe the Observer pattern
- Describe the Abstract Factory pattern
- Describe the State pattern

Additional Resources

Additional resources – The following references provide additional information on the topics described in this module:

- Alexander, Christopher. *A Pattern Language: Towns Buildings Construction*. Oxford University Press, Inc., 1977.
- Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns.* West Sussex, England: John Wiley & Sons, Ltd., 1996.
- Gamma, Erich, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- Knoernschild, Kirk. *Java Design (Objects, UML, and Process)*. Reading: Addison Wesley Longman, Inc., 2002.
- Metske, Steven John. *Design Patterns Java Workbook*. Addison Wesley Professional, 2002
- Meyer, Bertrand. *Object-Oriented Software Construction (2nd ed)*. Upper Saddle River: Prentice Hall, 1997.
- Shalloway, Alan, and James Trott. *Design Patterns Explained: A New Perspective on Object-Oriented Design*. Addison-Wesley, 2001.
- Stelting, Stephen, and Olav Maassen. *Applied Java Patterns*. Palo Alto: Sun Microsystems Press, 2002.
- Vlissides, John, James Coplien, and Norman Kerth. *Pattern Language of Program Design (vol. 2)*. Reading: Addison Wesley Longman, Inc., 1996.

Note – There are also several good Web sites on Software Patterns: `http://hillside.net/patterns` and `http://gee.cs.oswego.edu/dl/pd-FAQ/pd-FAQ.html`

Process Map

This module describes the next step in the Design workflow: applying design patterns to the Design model. Figure 10-1 shows the activity and artifacts discussed in this module.

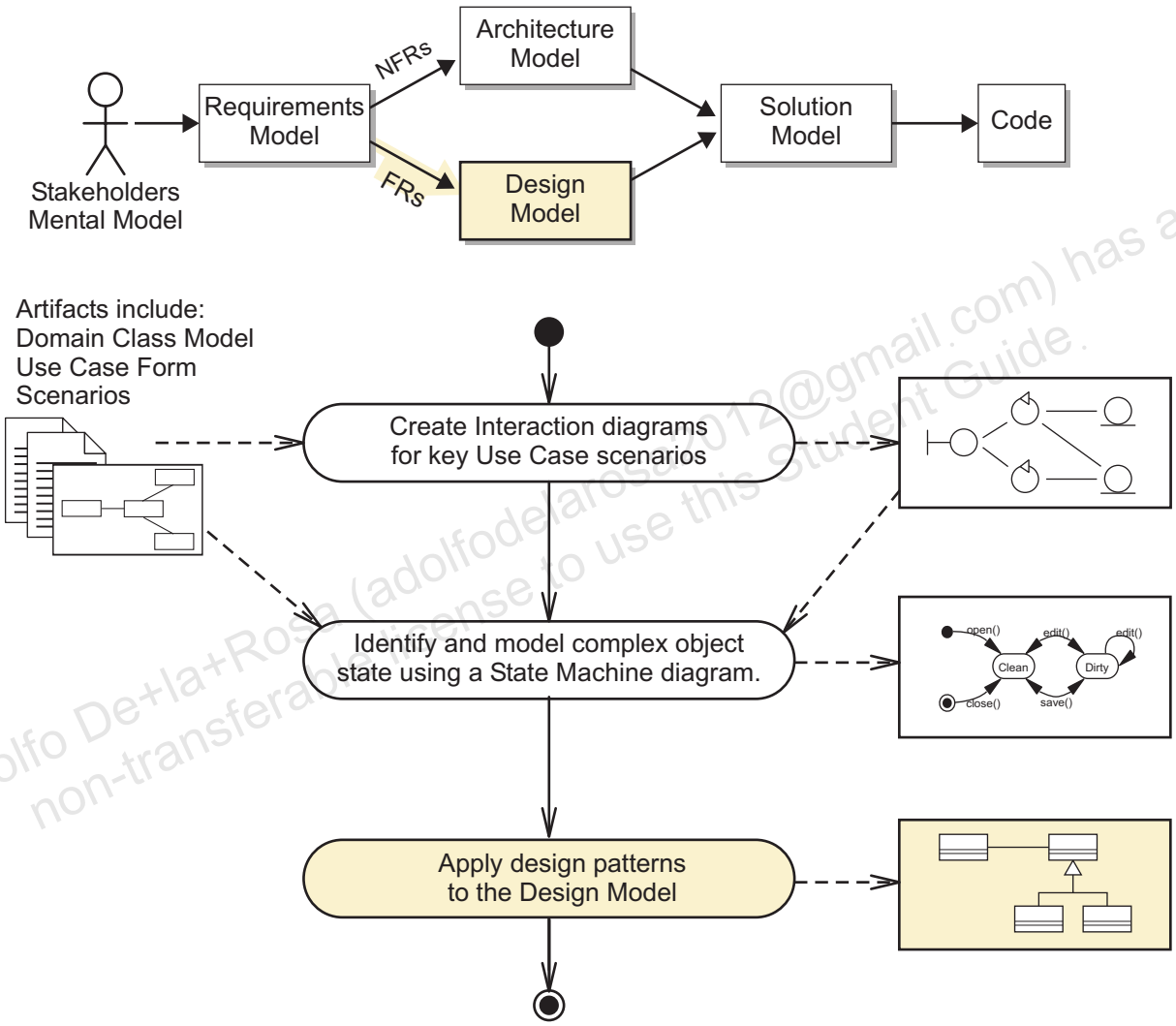


Figure 10-1 Design Workflow Process Map

Explaining Software Patterns

> "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice." (Alexander page x)

This statement was written about building architecture, but it is also true in object-oriented design. In software design, patterns appear over and over again.

A *software pattern* is a "description of communicating objects and classes that are customized to solve a general design problem in a particular context." (Gamma, Helm, Johnson, Vlissides page 3)

Note – The authors of the *Design Patterns* book Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides are commonly referred to collectively as the Gang of Four (GoF).

The discipline of software patterns formalizes these patterns by identifying four essential elements of the pattern:

- Pattern name

  This element provides a convenient and memorable term for the pattern. This element is useful when the development team communicates about the software design because it raises the level of abstraction in the conversation.

- Problem

  This element states the conditions under which the pattern applies.

- Solution

  This element states the structural and dynamic behavior of the components that make up the pattern.

- Consequences

  This element states the results of the pattern as well as any trade-offs incurred by applying the pattern.

Note – There are potentially dozens of elements that could be described about pattern; for example, the GoF use twelve elements in their pattern descriptions. However, the previous four elements are considered essential.

Levels of Software Patterns

Large software systems can be visualized at many levels of depth. Similarly, software patterns can be applied at many levels of depth. Patterns can be roughly grouped into the following three levels:

- Architectural patterns

  Patterns at this level manifest at the highest software and hardware structures within the system. For example, a Remote Proxy (GoF) is a pattern that enables a client component to access a remote service as if the client were accessing a local service. An RMI stub is an example of a Remote Proxy.

  Architectural patterns usually support the non-functional requirements of the system. For example, the Remote Proxy can increase the throughput of the Client (or Presentation) tier by moving an expensive service to another hardware node.

- Design patterns

  Patterns at this level manifest at the mid-level software structures within the system. These patterns apply to a small set of classes or components.

  Design patterns usually support the functional requirements of the system. For example, the Composite pattern directly supports a FR which states “the system contains a set of objects that all support the operations of X, Y, and Z; furthermore, the group of these object must also support those operations.”

- Idioms

  Patterns at this level manifest at the lowest software structures (classes and methods) within the system. These patterns apply either within a class definition or even with the code of a method. For example, Localized Ownership (Vlissides, Coplien, and Kerth page 5) is a pattern for managing object creation and destruction in C++.

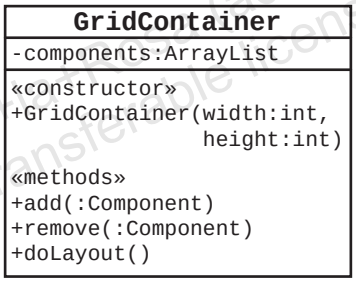
Idioms usually support language-specific features. For example, the Localized Ownership pattern supports the memory management model of C++ programs; it is not applicable to Java technology because the JVM software supplies automated memory management using garbage collection.

Design Principles

There are several design principles that support the solutions of software patterns:

- Open Closed Principle (OCP)
- Composite Reuse Principle (CRP)
- Dependency Inversion Principle (DIP)

The following sections discuss each of these principles using a common example. This example describes the design of a GUI `GridContainer` class; a GUI container that places each GUI component element in a grid starting from the top left and adding components to the right and to the bottom. Figure 10-2 shows this component and an calculator example.




```
GridContainer calcGUI = new GridContainer(4,4);
calcGUI.add(new Button("1"));
calcGUI.add(new Button("2"));
calcGUI.add(new Button("3"));
calcGUI.add(new Button("+"));
calcGUI.add(new Button("4"));
calcGUI.add(new Button("5"));
calcGUI.add(new Button("6"));
calcGUI.add(new Button("-"));
```

1	2	3	+
4	5	6	-
7	8	9	*
0	.	=	/

Figure 10-2 GUI Example that Demonstrates Design Principles

Open Closed Principle

> "Classes should be open for extension but closed for modification." (Knoernschild page 8)[1]

The point of this principle is that you should be able to change a class without affecting the clients of that class. There are two dimensions to this kind of change, change by extending the capabilities of the class and change by modifying the implementation of the class. Figure 10-3 illustrates this principle.

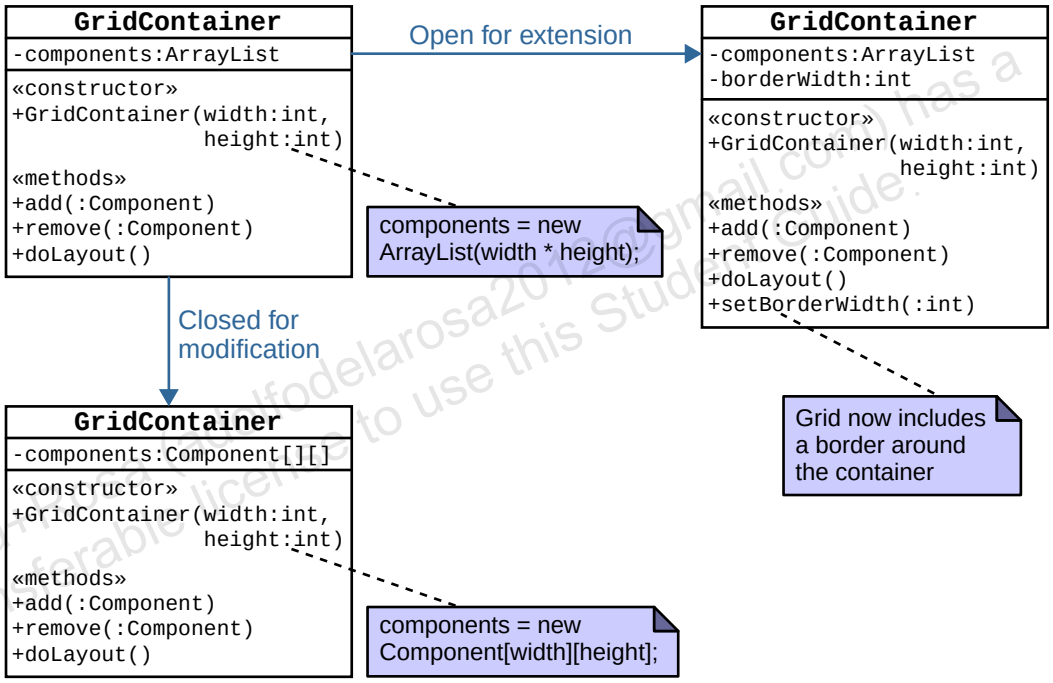


Figure 10-3 Example of the Open Closed Principle

The `GridContainer` class can be extended by adding a line border around the grid. This extension is made by adding the `setBorderWidth` method. An existing client of this class does not need to be changed because this behavior does not change the original behavior of the class. Also, the `GridContainer` class can be modified because the private implementation of the layout mechanism is hidden from the client. In particular, the data structure of the `components` attribute can be changed from a linear list to a matrix. An existing client of this class need not be changed due to this implementation change because the interface has not changed.

1. This principle was first introduced by Bertrand Meyer in OOSC page 57. However, Knoernschild's treatment is easier to understand.

Composite Reuse Principle

> "Favor polymorphic composition of objects over inheritance." (Knoernschild page 17)[2]

The point of this principle is that creating new behaviors by using inheritance is often not very flexible. In the GUI container example, you might need a new type of container layout: instead of a grid, you might want to layout a group of components into regions around the container screen. You could create an abstract `Container` superclass with two subclasses, `GridContainer` and `BorderContainer`. Figure 10-4 illustrates this.

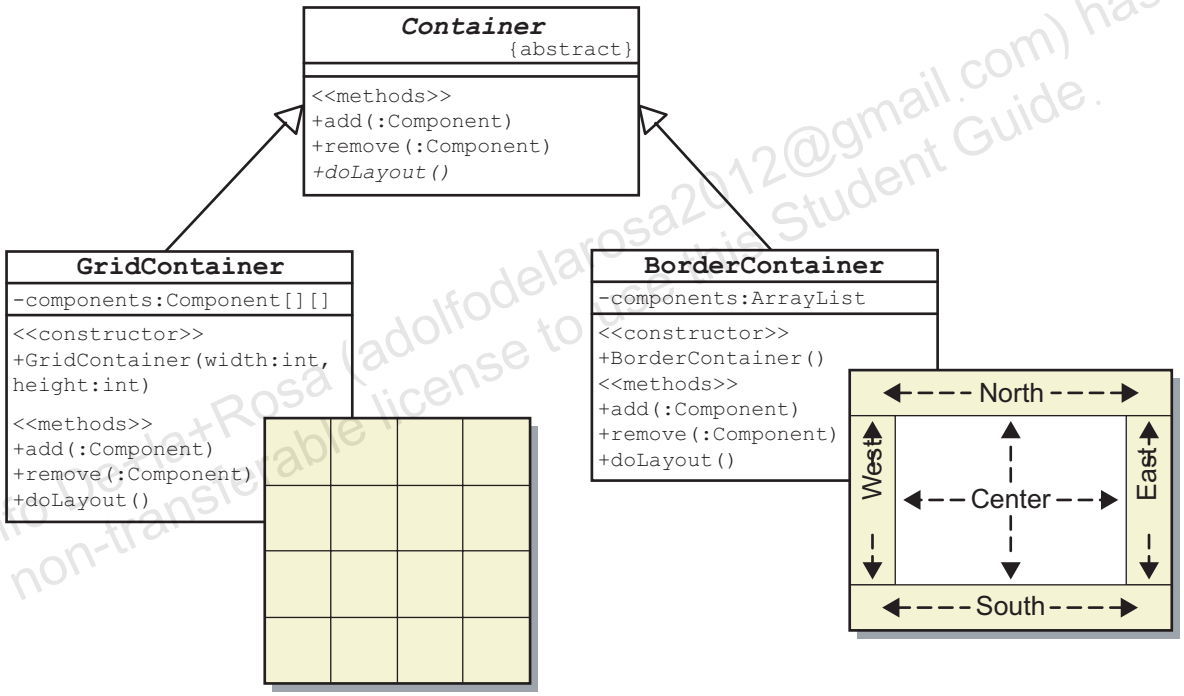


Figure 10-4 Using Inheritance to Provide Different GUI Layout Mechanisms

2. CRP was first introduced in the GoF *Design Patterns* book.

There is nothing inherently wrong with this strategy. However, GUI containers can have many different features. Suppose that you want to be able to create a border around the group of GUI components. You could create a flat border which draws a line around the boundary of the GUI components. Or you could draw an etched border around the components to give a 3D effect to the container. Conceptually, you have two independent hierarchies of functionality, one for container layout and one for drawing a border around the container on the screen. However, you cannot have two hierarchies, but rather a single hierarchy that is a cross product of the two features. Figure 10-5 illustrates this.

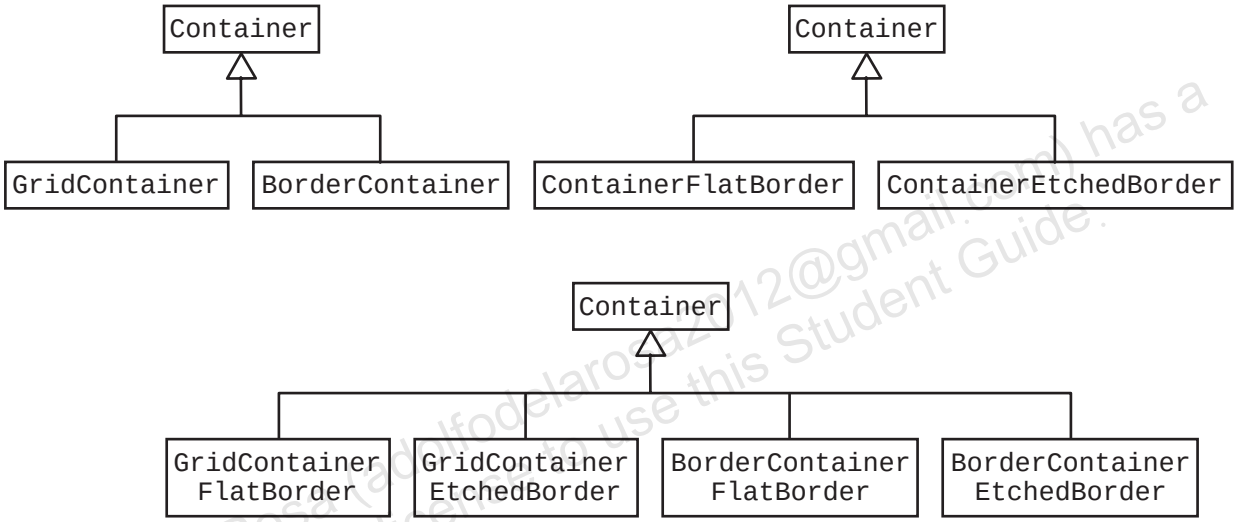


Figure 10-5 Variation in Multiple Features Leads to Combinatorial Explosion of Classes

This is a brittle class structure. Suppose that you now needed a new border visualization (for example, a rounded corner border). You would have to create two additional classes: `GridContainerRoundedBorder` and `BorderContainerRoundedBorder`. If you have *N* types of layout and *M* types of borders, then you potentially must create *N* * *M* container classes.

CRP provides an alternative to this problem. For each feature of a GUI container, you could create a separate class hierarchy that supports only that feature. The `Container` class would then use an implementation of a class that supports that feature. For example, you could have a single `Container` class that delegates the layout functionality to an independent object. Figure 10-6 illustrates this.

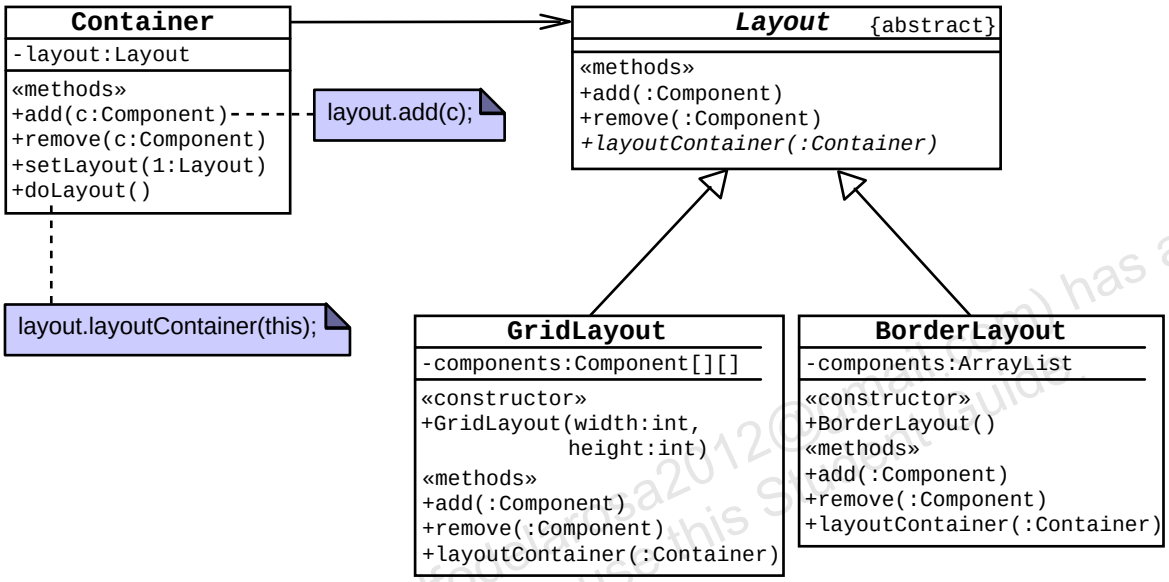


Figure 10-6 Example of the Composite Reuse Principle

The GUI border functionality can also be delegated by the `Container` class. This is not shown in the previous diagram, but it does make sense to use CRP to delegate this functionality to a separate object, with a class hierarchy independent of the `Container` class.

Dependency Inversion Principle

> "Depend on abstractions. Do not depend on concretions." (Knoernschild page 12)[3]

CRP provides a glimpse of the Dependency Inversion Principle. The point of DIP is that if you need to delegate to another component, then it is best to write the client code to an abstract class. Figure 10-6 on page 10-10 shows an example of DIP. The `Container` class uses an object of the `Layout` class hierarchy. The `Container` class does not need to know which layout manager is used, because the client of the container uses the `setLayout` method to specify the layout manager at runtime.

Figure 10-7 illustrates a generic version of this principle.

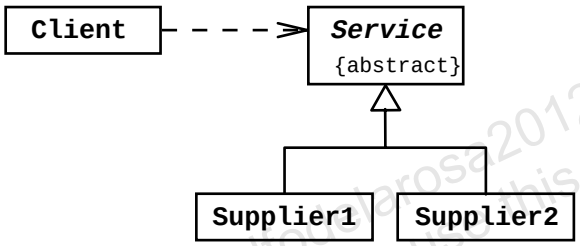


Figure 10-7 Dependency Inversion Principle Using an Abstract Class

Figure 10-8 shows DIP can be used with interfaces.

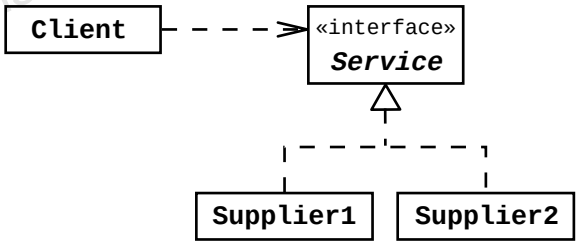


Figure 10-8 Dependency Inversion Principle Using an Interface

3. DIP was originally introduced by Robert Martin in a *C++ Report* article. This article is available online at `http://www.objectmentor.com/resources/articles/dip.pdf`)

Describing the Composite Pattern

> "Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly." (GoF page 163)

The Composite pattern deals with whole-part hierarchies. A GUI container is an example of a whole-part hierarchy: the container holds one or more GUI components. For example, a screen might have a group of GUI components to collect customer information, name, address, and so on. The screen (implemented by a `Frame` class) is the container and labels and text fields are the parts that make up the whole. Abstract Window Toolkit (AWT) includes a class called `Container` that enables the programmer to add and remove components to the container. The `Frame` class is a subclass of `Container`. Figure 10-9 illustrates this class hierarchy.

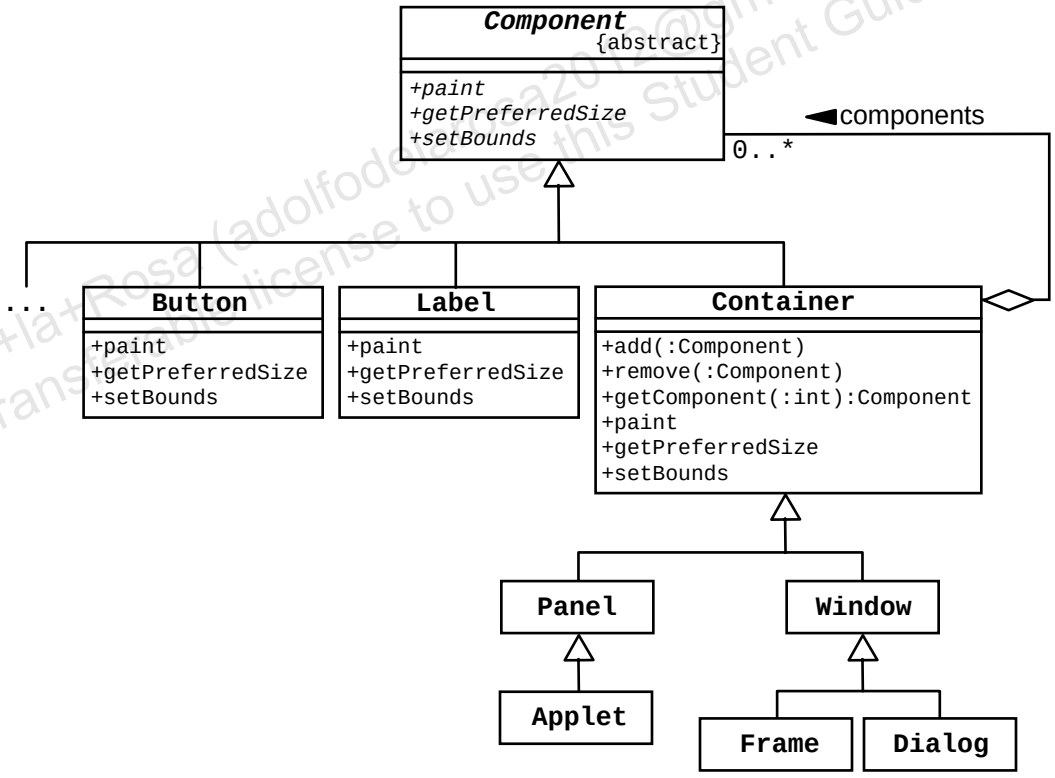


Figure 10-9 An Example of the Composite Pattern in AWT

Composite Pattern: Problem

The Composite pattern solves a problem having the following characteristics:

- You want to represent whole-part hierarchies of objects
- You want to use the same interface on the assemblies and the components in an assembly

Composite Pattern: Solution

The solution for the Composite pattern has the following characteristics:

- Create an abstract class, `Component`, that acts as the superclass for concrete "leaf" and `Composite` classes.
- The `Composite` class can be treated as a component because it supports the `Component` class interface.

Figure 10-10 illustrates the Composite pattern solution described by the GoF. In this solution, the `Component` class includes all of the composite methods: `add`, `remove`, and `getChild`.

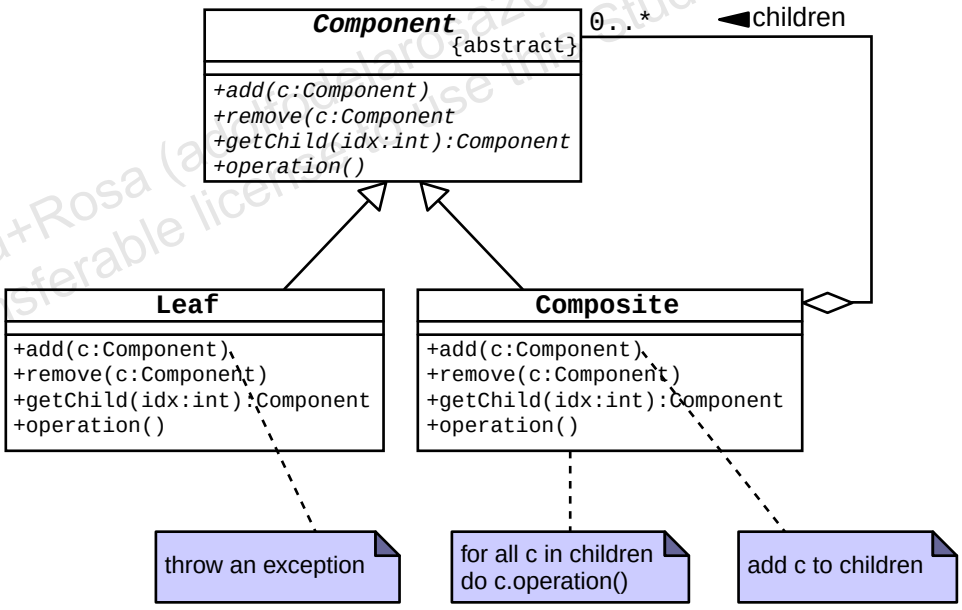


Figure 10-10 GoF Solution for the Composite Pattern

This might seem confusing at first because leaf classes would not have meaningful implementations of these methods. However, this solution provides the most flexibility for the client code because all subclasses of Component use the same interface. An alternate solution puts the composite methods only in the `Composite` class. Figure 10-11 illustrates this.

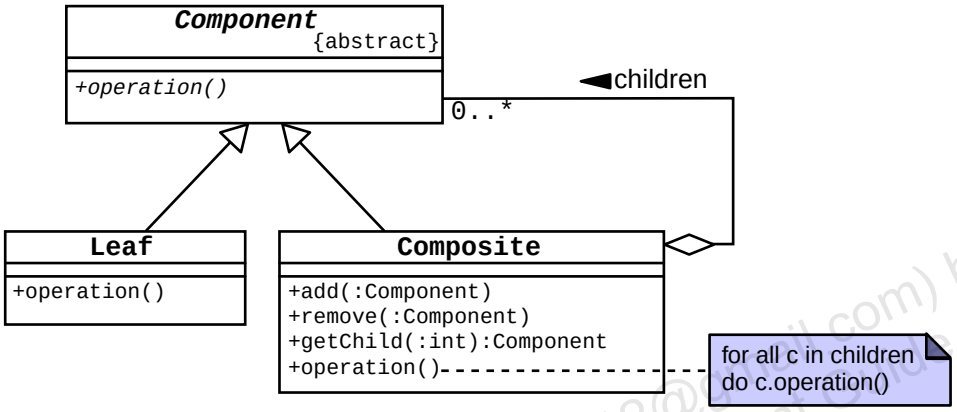


Figure 10-11 An Alternate Solution for the Composite Pattern

Composite Pattern: Consequences

The consequences of the Composite pattern include:

- Makes the client simple

  The client of a composite parts hierarchy does not need to care what type of component (including a composite component) it is dealing with; it can call any operation on the component object.

- Makes it easier to add new kinds of components

  You can add new types of components (and even new types of composites) to the component hierarchy. As long as the new class follows the same interface, then the client classes do not have to be modified.

- Can make the design model too general

  This pattern makes the component hierarchy very generic. A composite can contain *any type* of component. There might be cases in which you want to restrict the types of components a given composite can hold. This is not possible with this pattern.

Describing the Strategy Pattern

> "Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it." (GoF page 315)

GUI containers in AWT can be organized into several predefined layouts: grid, border, flow, and so on. AWT provides this mechanism by using a layout manager that is assigned to every GUI container. The container delegates the algorithm of laying out its components to this layout manager. This is a direct implementation of the Strategy pattern. Figure 10-12 illustrates this AWT example.

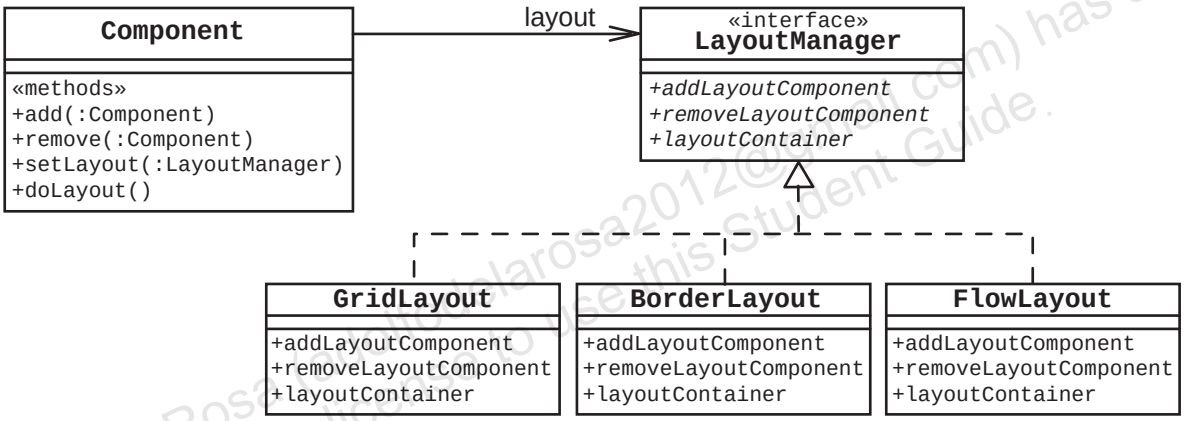


Figure 10-12 An Example of the Strategy Pattern in AWT

Strategy Pattern: Problem

The Strategy pattern solves a problem having the following characteristics:

- You have a set of classes that are only different in the algorithms that they use
- You want to change algorithms at runtime

Strategy Pattern: Solution

The solution for the Strategy pattern has the following characteristics:

- Create an interface, `Strategy`, that is implemented by a set of concrete "algorithm" classes.

- At runtime, select an instance of these concrete classes within the `Context` class.

Figure 10-13 illustrates the Strategy pattern solution described by the GoF.

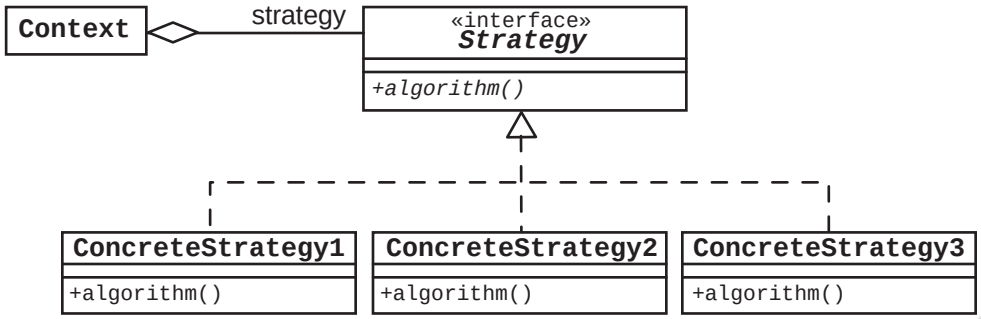


Figure 10-13 GoF Solution for the Strategy Pattern

Strategy Pattern: Consequences

The consequences of the Strategy pattern include:

- An alternate to subclassing

  As you saw with GUI containers in "Composite Reuse Principle" on page 10-8, it might seem reasonable to subclass a Context (such as a GUI container) into subclasses that supports specific algorithms, such as GUI layouts. This technique leads to an artificial hierarchy of classes (see Figure 10-4 on page 10-8) or a combinatorial explosion of classes if there are multiple algorithms that need to vary (see Figure 10-5 on page 10-9). The Strategy pattern eliminates the need to subclass to support multiple algorithms.

- Strategies eliminate conditional statements

  Some programmers might consolidate multiple algorithms into a single algorithm that uses conditional statements to branch between the differences in each algorithm. This solution leads to complex code that is hard to maintain. The Strategy pattern eliminates conditional statements by putting each algorithm in it own class.

- A choice of implementations

  Using the Strategy pattern, the context object can change the algorithm at runtime. This solution increases the flexibility of the software.

- Communication overhead between Strategy and Context patterns

On the downside, the Strategy pattern introduces additional communication overhead between the strategy objects and the context object. This could lead to complicated code if a given strategy requires private details of the context.

- Increased number of objects

  Using the Strategy pattern introduces the need for additional objects (strategy objects) at runtime. However, these objects might not require any attributes of their own; therefore, you could use a single instance of each concrete strategy class that all contexts share.

Describing the Observer Pattern

> "Define a one-to-many dependency between objects so that when one object changes, all its dependents are notified and updated automatically." (GoF page 293)

Suppose that you have a set of data (a collection of numbers) that you need to visualize using several different techniques using a pie chart, bar chart, or as a row in a spreadsheet. Figure 10-14 illustrates this situation.

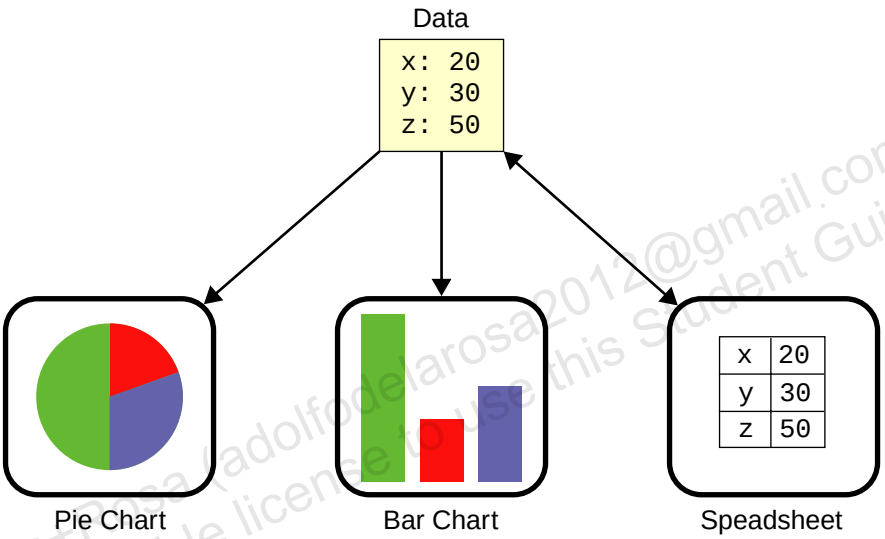


Figure 10-14 An Example of the Observer Pattern in a GUI Application

If you kept a copy of the data in each GUI component, then your software would have to modify the data in each of these components when the data changed in the system. This tight coupling between the UI components and the data model is brittle. A change in the data model would require changes to each UI component that must visualize that data.

Alternatively, you could keep the data in a separate object and then notify the UI components when that data has changed. Figure 10-15 illustrates this alternate approach in an Object diagram.

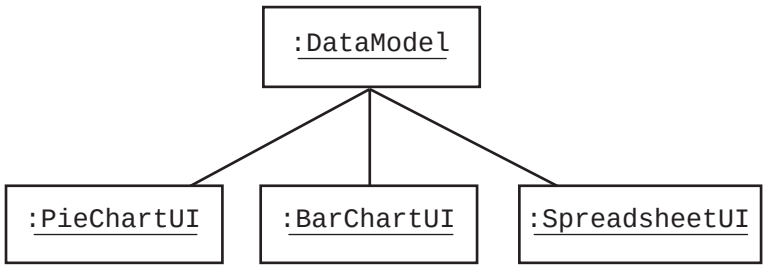


Figure 10-15 Object Diagram of the Observer Example

The UI classes must implement the `Observer` interface to perform the necessary changes to the UI representation when the `update` method is invoked. The `DataModel` class inherits from the `Observable` abstract class which provides the methods to attach and detach observers. The `DataModel` class is responsible for invoking the `notify` method whenever its data change. Figure 10-16 illustrates this design.

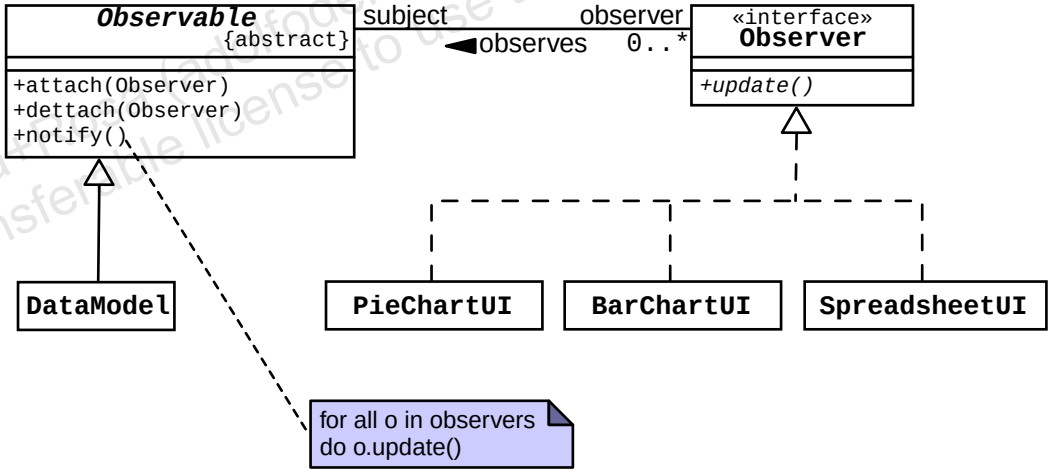


Figure 10-16 Class Diagram of the Observer Example

Observer Pattern: Problem

The Observer pattern solves a problem having the following characteristics:

- You need to notify a set of objects that an event has occurred.
- The set of observing objects can change at runtime.

Observer Pattern: Solution

The solution for the Observer pattern has the following characteristics:

- Create an abstract class `Subject` that maintains a collection of `Observer` objects.
- When a change occurs in a subject, it notifies all of the observers in its set.

Figure 10-17 illustrates the Observer pattern solution described by the GoF.

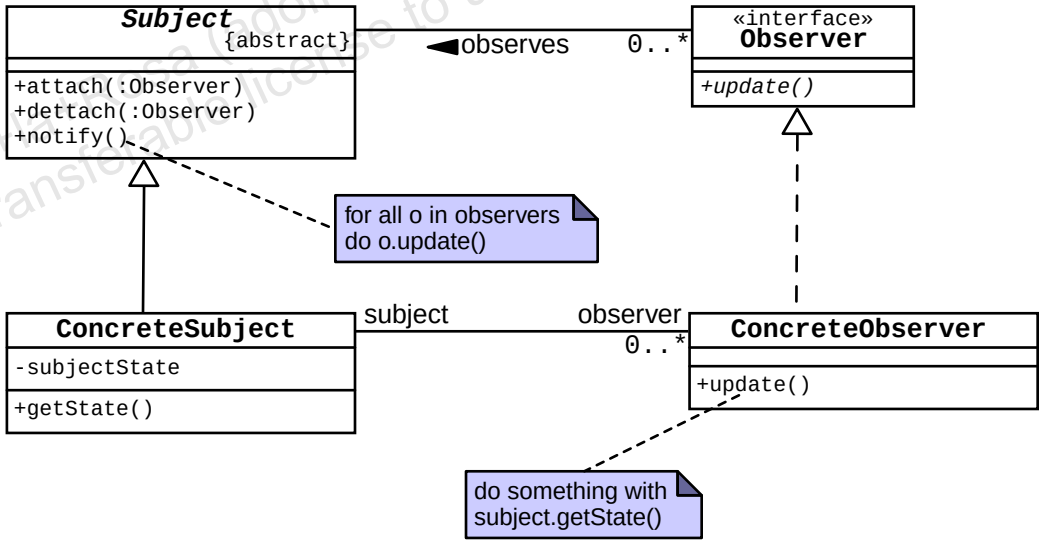


Figure 10-17 GoF Solution for the Observer Pattern

Observer Pattern: Consequences

The consequences of the Observer pattern include:

- Abstract coupling between Subject and Observer

  The coupling between the subject object and the observer object is abstract. This coupling is abstract because the subject does not know the concrete class of each observer; it only knows that the observer object supports the `Observer` interface (the `update` method).

- Support for multicast communication

  Observer objects can dynamically attach to a subject. Whenever the subject changes, all attached observers are notified in a broadcast of `update` messages to ever Observer.

- Unexpected updates

  A downside to this pattern is that the communication mechanism is coarse-grained. Observers might not care if one part of the data model is changed, but there is no mechanism for the subject to know which Observer is interested in which data changes. It is up to the Observer to query the data model and decide if the changes need to be processed.

Java technology uses the Observer pattern in several different ways. In AWT, each GUI component can listen for user input events, such as typing a character in a text components, selecting a menu item, or clicking on a button. You can create and attach listener objects to GUI components that observe various user events for that component. These listener are observers in the Observer pattern and the GUI component is the subject.

Describing the Abstract Factory Pattern

> "Provide an interface for creating families of related or dependent objects without specifying their concrete classes." (GoF page 87)

Java technology is based on using the Java virtual machine to provide platform-independence. Creating platform-independent GUIs was particularly challenging. The design of the AWT required the separation of abstract UI components, such as `java.awt.Button`, from the actual operating system components that physically presented the UI component on the screen. The real components are called peers of the abstract UI components. Therefore, there is a set of peers for UI components on the Motif platform as well as a set of peers on the Microsoft Windows platform. The abstract AWT component uses the AWT `Toolkit` class to create the appropriate peer at runtime. When the J2SE platform Software Development Kit (SDK) is created for a specific platform, then the SDK includes a concrete subclass of `Toolkit` which provides the methods to generate the relevant peers for that platform. Figure 10-18 illustrates this design.

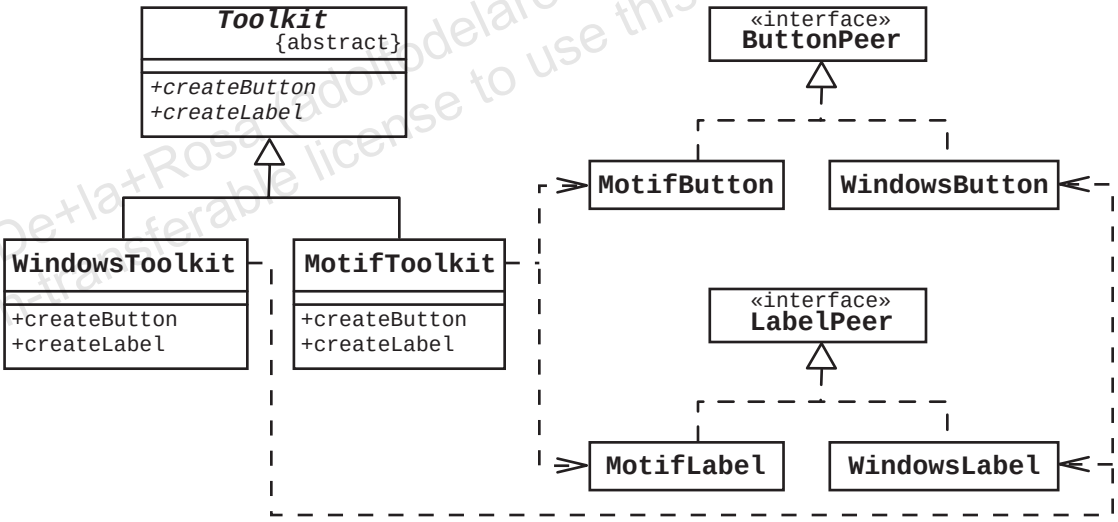


Figure 10-18 An Example of the Abstract Factory Pattern in AWT

Note – Swing is the second generation GUI framework for Java technology that provides true platform-independence. Swing GUI components are not based on peers.

Abstract Factory Pattern: Problem

The Abstract Factory pattern solves a problem having the following characteristics:

- A system has multiple families of products.
- Each product family is designed to be used together.
- You do not want to reveal the implementation classes of the product families.

Abstract Factory Pattern: Solution

The solution for the Abstract Factory pattern has the following characteristics:

- Create an abstract creator class that has a factory method for each type of product.
- Create a concrete creator class that implements each factory method which returns a concrete product.

Figure 10-19 illustrates the Abstract Factory pattern solution described by the GoF.

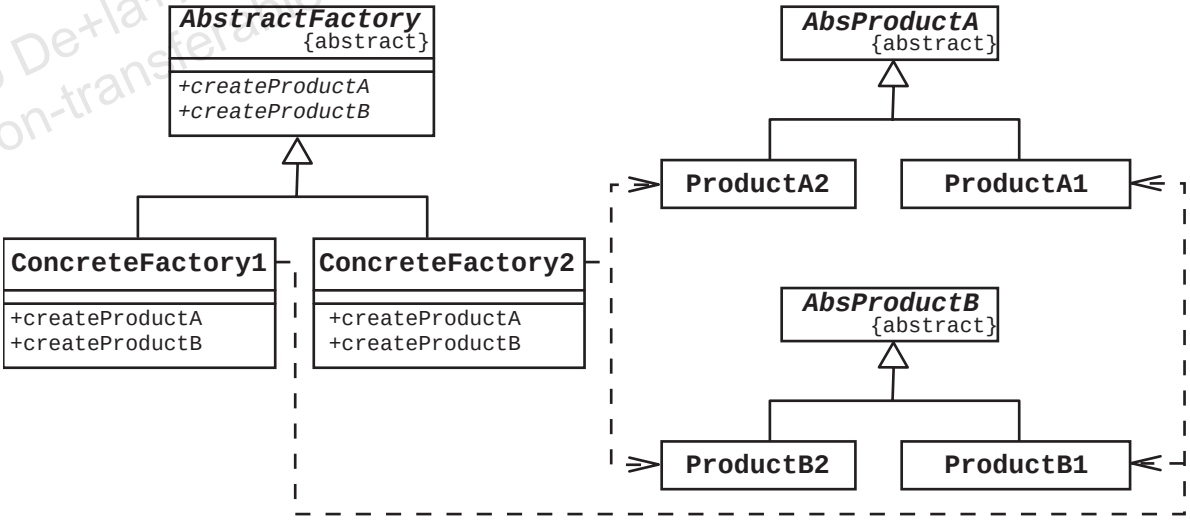


Figure 10-19 GoF Solution for the Abstract Factory Pattern

Abstract Factory Pattern: Consequences

The consequences of the Abstract Factory pattern include:

- Isolates concrete classes

  This pattern hides the concrete product classes, so the clients of these products must use the abstract class (or interface) thus enforcing the client into using the DIP.

- Makes exchanging product families easy

  You can add and remove whole product families without changing the client code, because it depends on the abstract product classes rather than on the concrete classes.

- Promotes consistency among products

  "When product objects in a family are designed to work together, it is important that an application use objects from only one family at a time. Abstract Factory makes this easy to enforce." (GoF page 90)

  The best example of this in Java technology is a variation on Abstract Factory in the JDBC framework. The factory methods start at the `DriverManager` class level. The `getConnection(URL)` method returns a `Connection` object for the specific JDBC driver that matches the URL. The `Connection` interface supplies the `createStatement` factory method. This method returns a `Statement` object containing a `executeQuery` method that creates a `ResultSet` object. In the JDBC framework, the factory methods are scattered among multiple interfaces: `Connection`, `Statement`, and `ResultSet`. These are the products for which every JDBC vendor must provide an implementation (a concrete product). These products are built by the vendor to work together; the Abstract Factory variation in the JDBC framework is built to enforce that consistency.

- Supporting new kinds of products is difficult

  On the downside, changing the set of products is rather difficult because you have to update all of the product families that are currently used.

Programming a Complex Object

This section discusses the problems with programming a complex object and suggests a design pattern to simplify this development.

Problems With Coding a Complex Object

Before introducing the State pattern, you should understand why the pattern exists at all. Objects with complex state often are very difficult to code. A naive approach is to use the attributes of the object to determine the state of the object which performs the actions of the state. This can lead to very complex, and hard to maintain code because such code usually includes significantly complex conditional statements. Figure 10-20 shows an example implementation of the `setRealTemp` method in the `HVAC` class.

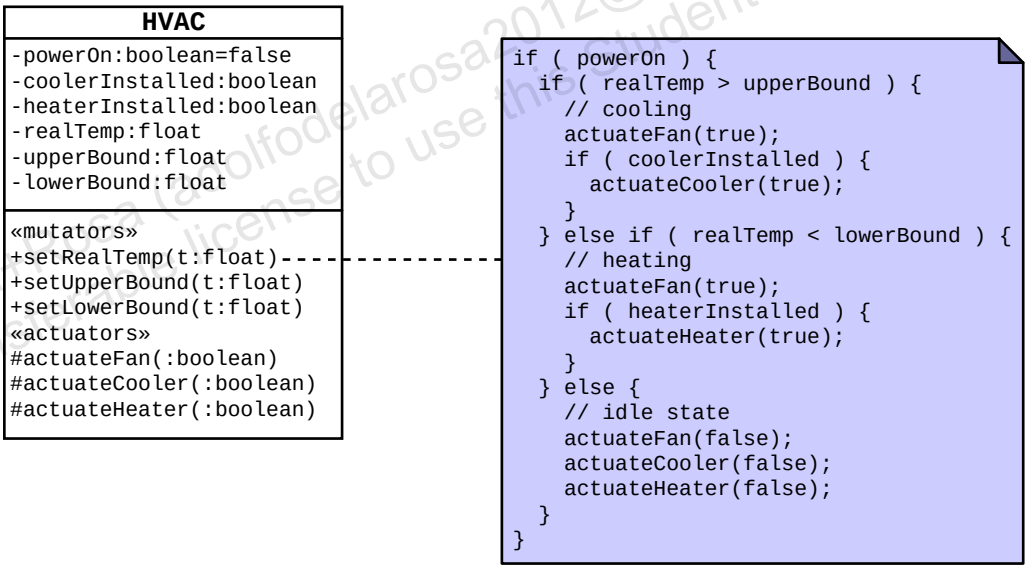


Figure 10-20 HVAC Class With Complex State Code

Not only is the code complex, but the code might not truly capture the state behavior of the object. The `setRealTemp` method has several if and if-else constructs to determine if the HVAC object should be cooling, heating, or idle. In the else-clause for the idle condition, the method calls three actuator methods. If these methods are expensive operations, then this code is wasting time being idle. To code this method correctly would require even more conditional logic.

Another issue is: How do you change this code if a new state is introduced to the HVAC system? Creating new states or changing the behavior of existing states can be a maintenance nightmare because you will have to review *all* of the code in the `HVAC` class.

The State pattern solves all of these problems.

Describing the State Pattern

> "Allow an object to alter its behavior when its internal state changes. The object will appear to change its class." (GoF page 305)

The State pattern introduces a set of state objects within which the behavior of the primary object is encapsulated in the state class methods. For the HVAC class, you would create four state classes: `InitialState`, `IdleState`, `CoolingState`, and `HeatingState`. Each of these classes implements a `State` interface that is specifically designed for the primary object (each primary class has its own interface for its state objects).

The primary class keeps a `state` attribute that holds the particular state object for the state of the primary object. The primary class includes a protected `setState` method which performs the state transitions for the primary object. This method calls the exit action before changing state, changes the `state` attribute, and then calls the entry actions for the new state. The state-dependent methods of the primary class delegate the behavior to the current state object. In the HVAC class, the `setRealTemp` method stores the new temperature value and then calls the corresponding method on the state object. Figure 10-21 illustrates the HVAC class using the State pattern.

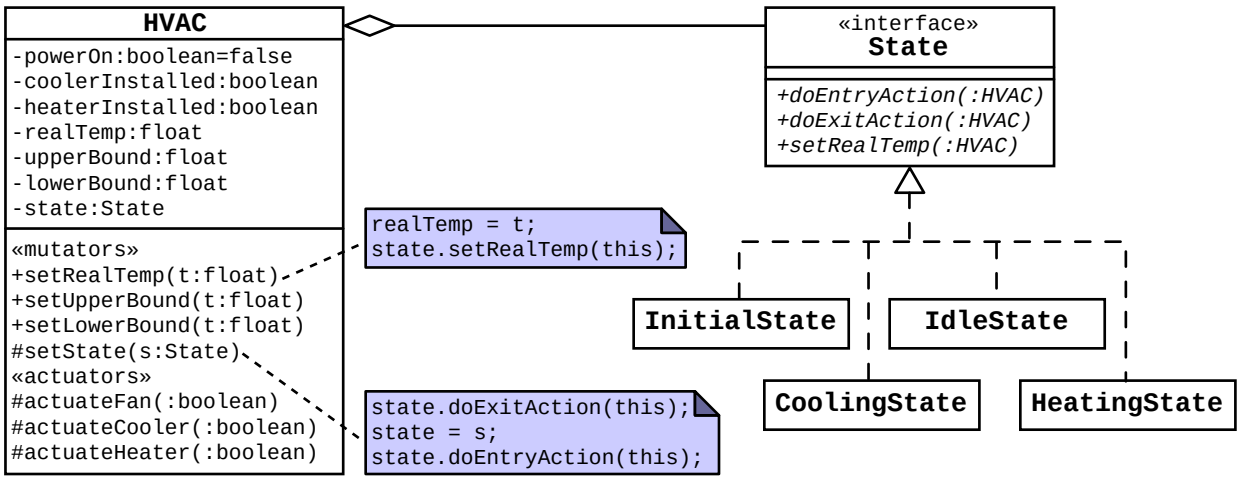


Figure 10-21 HVAC Class Using the State Pattern

Each state class focuses only on the behavior of that state. For example, the `CoolingState` class includes an entry action of turning on the AC subsystem, an exit action of turning the AC off, and a `setRealTemp` method that changes the state of the HVAC object to Idle if the temperature has dropped below the upper bound value. Figure 10-22 illustrates this.

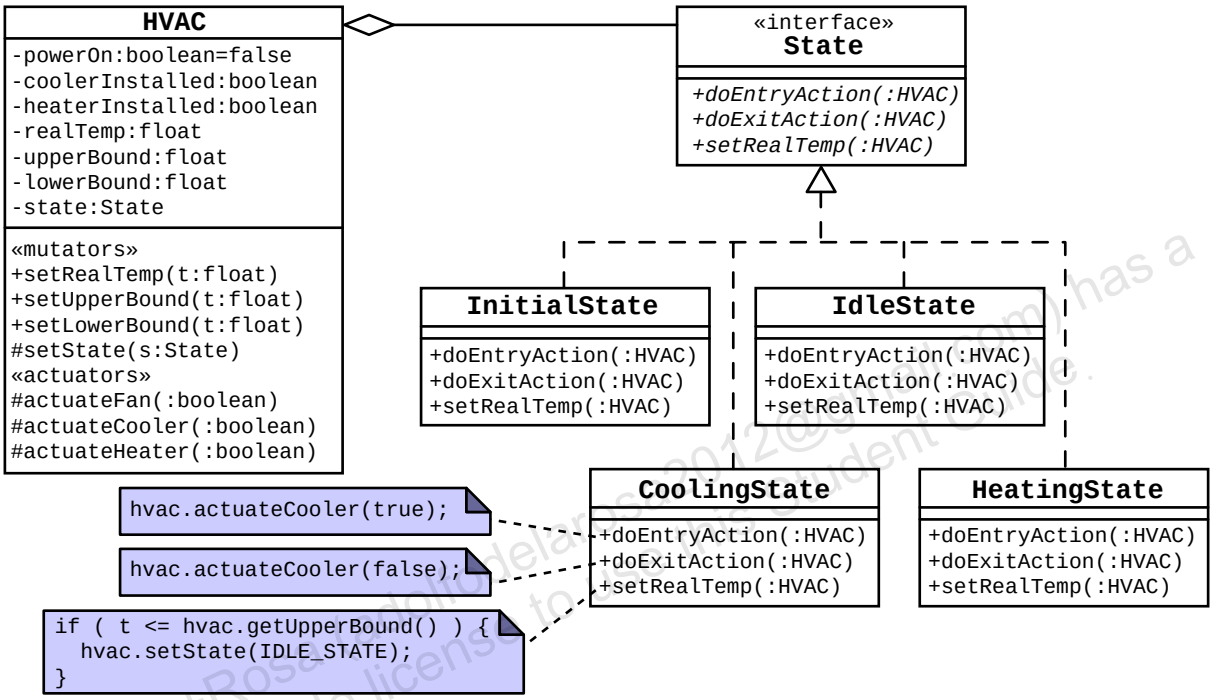


Figure 10-22 HVAC Class Using the State Pattern

State Pattern: Problem

The State pattern solves the problem having the following characteristics:

- An object's runtime behavior depends on its state.
- Operations have large, multipart conditional statements that depend on the object's state.

State Pattern: Solution

The solution for the State pattern has the following characteristics:

- Create an interface that specifies the state-based behaviors of the object.
- Create a concrete implementation of this interface for each state of the object.

- Dispatch the state-based behaviors of the object to the object's current state object.

Figure 10-23 illustrates the State pattern solution described by the GoF.

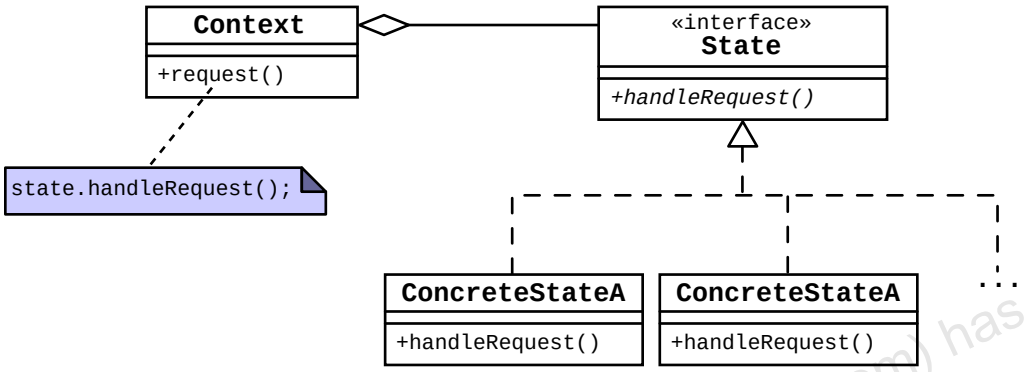


Figure 10-23 GoF Solution for the State Pattern

State Pattern: Consequences

The consequences of the State pattern include:

- Localizes state-specific behavior

  State-specific behavior is localized to the class that models that particular state. This solution greatly simplifies the development of state objects. The State pattern easily permits the addition of new state classes or the modification of the behavior of existing states without affecting the code in the other state classes.

- Reduces conditional statements

  Without using the State pattern, the methods of the primary class tend to use complex conditional code to keep track of the internal state of the object. The State pattern greatly reduces this type of conditional code.

- Makes state transitions explicit

  Without using the State pattern, the methods of the primary class often hide the details of when the object transitions from one state to another. Using the State pattern makes the state of the object and its state transitions explicit.

- Increased the number of objects

  Using the State pattern introduces the need for additional objects (state objects) at runtime. However, these objects might not require any attributes of their own; in that case, you could use a single instance of each concrete state class that all objects share.

- Communication overhead between the State and Context objects

The State pattern introduces additional communication overhead between the state objects and the context object. This could lead to complicated code if a given state requires private details of the context. One solution to this problem is to make the state classes as inner classes of the context class.

Summary

In this module, you were introduced to a few classic Design Patterns. Here are a few important concepts:

- Software patterns provide proven solutions to common problems.
- Design principles provide tools to build and recognize software patterns.
- Patterns are often used together to build more flexible and robust systems and frameworks (such as AWT and JDBC).