5 Design Patterns

CERTIFICATION OBJECTIVES

- 5.01 Identify the Benefits of Using Design Patterns
- 5.02 Identify the Most Appropriate Design Pattern for a Given Scenario
- 5.03 State the Name of a Gamma et al. Design Pattern Given the UML Diagram and/or a Brief Description of the Pattern's Functionality
- 5.04 Identify Benefits of a Specified Gamma et al. Design Pattern
- 5.05 Identify the Gamma et al. Design Pattern Associated with a Specified Java EE Technology Feature
- ✓ Two-Minute Drill
- Q&A Self Test

D*esign patterns,* or patterns, are solutions to recurring problems in a given context amid competing concerns. They try to bring together and document the core solution to a given problem. They are identified and documented in a form that's easy to share, discuss, and understand. They are useful problem-solving documentation for software designers and are used repeatedly to help solve common problems that arise in the course of software engineering. Documentation for the design pattern should provide a discussion on the difficulties and interests surrounding the problem and arguments as to why the given solution balances these competing interests or constraints that are inherent in the issue being solved.

The value of the pattern is not just the solution to the problem; value can also be found in the documentation that explains the underlying motivation, the essential workings of the solution, and why the design pattern is advantageous. The pattern student will be able to experience all or at least some of the experience and insight that went into providing the solution. This will undoubtedly help the designer to use the pattern and possibly adapt it or adjust it further to address needs accordingly.

Patterns can be combined and used in concert to solve larger problems that cannot be solved with just one pattern. Once the pattern student has become more familiar with these patterns, their combined applicability to a new set of problems will become much easier to identify.

CERTIFICATION OBJECTIVE 5.01

Identify the Benefits of Using Design Patterns

Design patterns are beneficial because they describe a problem that occurs repeatedly, and then they explain the solution to the problem in a way that can be used many times over. Design patterns are helpful for the following reasons:

- They help designers quickly focus on solutions if the designers can recognize patterns that have been successful in the past.
- The study of patterns can inspire designers to come up with new and unique ideas.
- They provide a common language for design discussions.
- They provide solutions to real-world problems.
- Their format captures knowledge and documents best practices for a domain.
- They document decisions and the rationale that lead to the solution.

- They reuse the experience of predecessors.
- They communicate the insight already gained previously.
- They describe the circumstances (when and where), the influences (who and what), and the resolution (how and why it balances the influences) of a solution.

Nevertheless, patterns are not the be-all and end-all, they are by no means a "silver bullet" or panacea, and they cannot be universally applied to all situations. You can't always find the solution to every problem by consulting the pattern playbook. Patterns have been excessively hyped and have been used by designers to make them appear knowledgeable.

Design Patterns by Gamma et al., Also Known as the Gang of Four (GoF)

The Gang of Four (or GoF, which consists of Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, authors of the classic reference *Design Patterns: Elements of Reusable Object-Oriented Software* [Addison-Wesley, 2005]) described patterns as "a solution to a problem in a context." These three elements—problem, solution, and context—are the essence of a pattern. As with all pattern creators, the GoF used a template to document patterns. Before we review the 23 patterns documented by the GoF, let's take a look at the format for these patterns.

Format for the GoF Design Patterns

Table 5-1 shows the elements and sections for the GoF Design Patterns format.

If you are new to patterns, Table 5-2 will be useful. It is a suggestion for the sequence in which you can easily study the GoF Design Patterns.

CERTIFICATION OBJECTIVE 5.02

Identify the Most Appropriate Design Pattern for a Given Scenario

We will now review each of the Gamma et al. design patterns, starting first with those that are used to create objects (Creational), and then moving on to those that are concerned with composition of classes and objects (Structural), and finally covering those that are concerned with the interaction and responsibility of objects (Behavioral).

TABLE 5-1 Gang of Four (GoF) Design Patterns Elements, Sections, Descriptions

Element/Section	Description
Name	Used to help convey the essence of the pattern.
Classification	Categories are **Creational** Patterns concerned with creation **Structural** Patterns concerned with composition **Behavioral** Patterns concerned with interaction and responsibility.
Intent	What problem does the pattern address? What does it do?
Also Known As	Other common names for the pattern.
Motivation	Scenario that illustrates the problem.
Applicability	Situations in which the pattern can be used.
Structure	Diagram representing the structure of classes and objects in the pattern. The GoF uses Object Modeling Technique (OMT) or Booch notation. Today, Unified Modeling Language (UML), a unification of OMT, Booch, and others, is commonly used.
Participants	Classes and/or objects participating in the design pattern along with their responsibilities.
Collaborations	How the participants work together to carry out their responsibilities.
Consequences	What objectives does the pattern achieve? What are the trade-offs and results?
Implementation	Implementation details (pitfalls, hints, or techniques) to consider. Are there language-specific issues?
Sample Code	Sample code.
Known Uses	Examples from the real world.
Related Patterns	Comparison and discussion of related patterns; scenarios where this pattern can be used in conjunction with another.

GoF Creational Design Patterns

Creational design patterns are concerned with the way objects are created. These patterns are used when a decision must be made at the time a class is instantiated. Typically, the details of the concrete class that is to be instantiated are hidden from (and unknown to) the calling class by an abstract class that knows only about the abstract class or the interface it implements. The following creational patterns are described by the GoF:

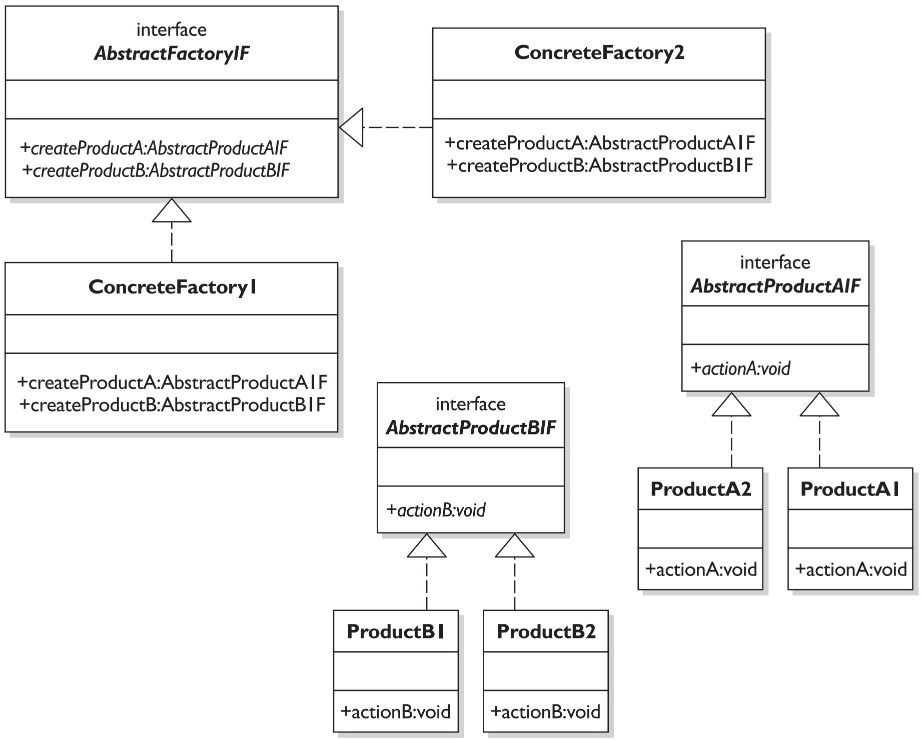
- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

TABLE 5-2 Study Sequence for Gang of Four (GoF) Design Patterns

Sequence	Design Pattern	Comment
1	Factory Method	Frequently used and also well utilized by other patterns.
2	Strategy	Frequently used, so early familiarity helps.
3	Decorator	Considered the “skin” to the “guts” of Strategy.
4	Composite	Often used along with Chain of Responsibility, Interpreter, Iterator, and Visitor.
5	Iterator	Looping through anything is widespread in computing, so why not through objects, too?
6	Template Method	Helps to reinforce your understanding of Strategy and Factory Method.
7	Abstract Factory	Create more than one type of a group of objects.
8	Builder	Another way to create, similar to Factory Method and Abstract Factory.
9	Singleton	You want only one copy of something.
10	Proxy	Controlled access to a service is needed.
11	Adapter	Gain access to a service with an incompatible interface.
12	Bridge	Decouples the function from the implementation.
13	Mediator	Yet another middleman.
14	Facade	Single interface simplifying multiple interfaces in a subsystem.
15	Observer	A form of the publish/subscribe model.
16	Chain of Responsibility	Passes the message along until it’s dealt with.
17	Memento	Backs up and restores an object’s state.
18	Command	Separates invoker from performer.
19	Prototype	Similar to cloning.
20	State	Object appears to change class and alter its behavior.
21	Visitor	Object that represents an operation that operates on elements of an object structure.
22	Flyweight	Allows you to utilize sharing to support large numbers of objects efficiently.
23	Interpreter	Defines a grammar and an interpreter that uses the grammar to interpret sentences.

FIGURE 5-1

UML for the Abstract Factory pattern



Abstract Factory

The Abstract Factory pattern's intent is to provide an interface to use for creating families of related (or dependent) objects without actually specifying their concrete classes. For a given set of related abstract classes, this pattern supplies a technique for creating instances of those abstract classes from an equivalent set of concrete subclasses. On some occasions, you may need to create an object without having to know which concrete subclass of object to create.

The Abstract Factory pattern is also known as *Kit*. The UML representation is shown in Figure 5-1.

Benefits Following is a list of benefits of using the Abstract Factory pattern:

- It isolates client from concrete (implementation) classes.
- It eases the exchanging of object families.
- It promotes consistency among objects.

Applicable Scenarios The following scenarios are most appropriate for the Abstract Factory pattern:

- The system needs to be independent of how its objects are created, composed, and represented.
- The system needs to be configured with one of a multiple family of objects.
- The family of related objects is intended to be used together, and this constraint needs to be enforced.
- You want to provide a library of objects that does not show implementations and only reveals interfaces.

Java EE Technology Features and Java SE API Association The Java EE technology features associated with the Abstract Factory pattern are

- Data Access Object (Sun)
- Transfer Object Assembler (Sun)

The Java Platform, Standard Edition (Java SE) API associated with the Abstract Factory pattern is *java.awt.Toolkit*.

Example Code Following is example Java code that demonstrates the Abstract Factory pattern:

```
package javaee.architect.AbstractFactory;
public class AbstractFactoryPattern {
  public static void main(String[] args) {
    System.out.println("Abstract Factory Pattern Demonstration.");
    System.out.println("--------------------------------------");
    // Create abstract factories
    System.out.println("Constructing abstract factories.");
    AbstractFactoryIF factoryOne = new FordFactory();
    AbstractFactoryIF factoryTwo = new GMFactory();
    // Create cars via abstract factories
    System.out.println("Constructing cars.");
    AbstractSportsCarIF  car1 = factoryOne.createSportsCar();
    AbstractEconomyCarIF car2 = factoryOne.createEconomyCar();
    AbstractSportsCarIF  car3 = factoryTwo.createSportsCar();
    AbstractEconomyCarIF car4 = factoryTwo.createEconomyCar();
    // Execute drive on the cars
    System.out.println("Calling drive on the cars.");
    car1.driveFast();
    car2.driveSlow();
    car3.driveFast();
    car4.driveSlow();
    System.out.println();
  }
}

package javaee.architect.AbstractFactory;
public interface AbstractFactoryIF {
  public AbstractSportsCarIF createSportsCar();
  public AbstractEconomyCarIF createEconomyCar();
}

package javaee.architect.AbstractFactory;
public interface AbstractSportsCarIF {
  public void driveFast();
}

package javaee.architect.AbstractFactory;
public interface AbstractEconomyCarIF {
  public void driveSlow();
}

package javaee.architect.AbstractFactory;
public class FordFactory implements AbstractFactoryIF {
  public AbstractSportsCarIF createSportsCar() {
    return new Mustang();
  }
  public AbstractEconomyCarIF createEconomyCar() {
    return new Focus();
  }
}

package javaee.architect.AbstractFactory;
public class GMFactory implements AbstractFactoryIF {
  public AbstractSportsCarIF createSportsCar() {
    return new Corvette();
  }
  public AbstractEconomyCarIF createEconomyCar() {
    return new Cavalier();
  }
}
```

```
package javaee.architect.AbstractFactory;
public class Mustang implements AbstractSportsCarIF {
  public void driveFast() {
    System.out.println("Mustang.driveFast() called.");
  }
}

package javaee.architect.AbstractFactory;
public class Focus implements AbstractEconomyCarIF {
  public void driveSlow() {
    System.out.println("Focus.driveSlow() called.");
  }
}

package javaee.architect.AbstractFactory;
public class Corvette implements AbstractSportsCarIF {
  public void driveFast() {
    System.out.println("Corvette.driveFast() called.");
  }
}

package javaee.architect.AbstractFactory;
public class Cavalier implements AbstractEconomyCarIF {
  public void driveSlow() {
    System.out.println("Cavalier.driveSlow() called.");
  }
}
```

Builder

The Builder pattern's intent is to separate the construction of a complex object from its representation so that the same construction process can create different objects. The Builder pattern is useful when several kinds of complex objects with similar rules for assembly need to be joined at runtime but result in different object types. It achieves this by separating the process of building the object from the object itself.

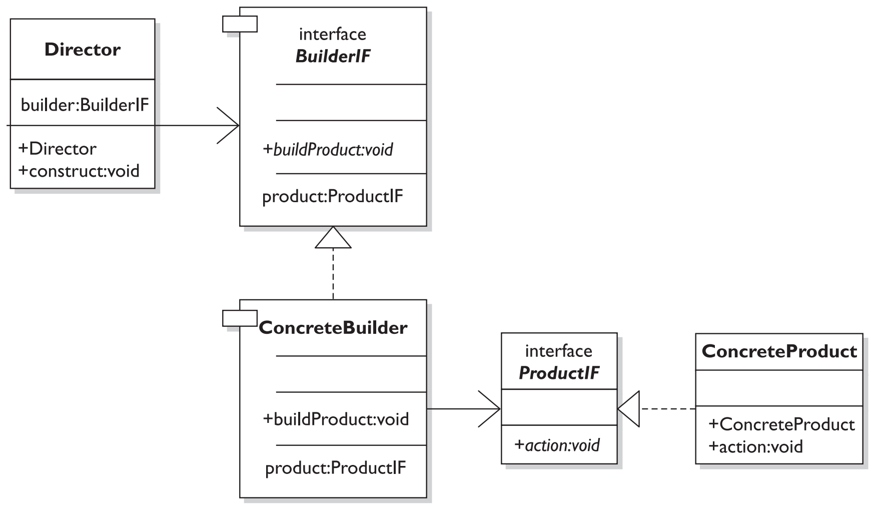
The Builder pattern creates complex objects in multiple steps instead of in a single step, as in other patterns. The UML is shown in Figure 5-2.

Benefits The following benefits are achieved when using the Builder pattern:

- It permits you to vary an object's internal representation.
- It isolates the code for construction and representation.
- It provides finer control over the construction process.

FIGURE 5-2

UML for the Builder pattern



Applicable Scenarios The following scenarios are most appropriate for the Builder pattern:

- The algorithm for creating a complex object needs to be independent of the components that compose the object and how they are assembled.
- The construction process is to allow different representations of the constructed object.

Example Code Following is some example Java code that demonstrates the Builder pattern:

```
package javaee.architect.Builder;
public class BuilderPattern {
  public static void main(String[] args) {
    System.out.println("Builder Pattern Demonstration.");
    System.out.println("------------------------------");
    // Create builder
    System.out.println("Constructing builder.");
    BuilderIF builder = new ConcreteBuilder();
    // Create director
    System.out.println("Constructing director.");
    Director director = new Director(builder);
    // Construct customer via director
    System.out.println("Constructing customer.");
    director.construct();
    // Get customer via builder
    CustomerIF customer = builder.getCustomer();
    // Use customer method
    System.out.println("Calling action on the customer.");
    customer.action();
    System.out.println();
  }
}

package javaee.architect.Builder;
public interface BuilderIF {
  public void buildCustomer();
  public CustomerIF getCustomer();
}

package javaee.architect.Builder;
public class ConcreteBuilder implements BuilderIF {
  CustomerIF customer;
  public void buildCustomer() {
    customer = new ConcreteCustomer();
    // You could add more customer processing here...
  }
  public CustomerIF getCustomer() {
    return customer;
  }
}

package javaee.architect.Builder;
public class ConcreteCustomer implements CustomerIF {
  public ConcreteCustomer() {
    System.out.println("ConcreteCustomer constructed.");
  }
  public void action() {
    System.out.println("ConcreteCustomer.action() called.");
  }
}

package javaee.architect.Builder;
public interface CustomerIF {
  public void action();
}

package javaee.architect.Builder;
public class Director {
  BuilderIF builder;
  public Director(BuilderIF parm) {
    this.builder = parm;
  }
  public void construct() {
    builder.buildCustomer();
  }
}
```

Factory Method

The Factory Method pattern's intent is to define an interface for creating an object but letting the subclass decide which class to instantiate. In other words, the class defers instantiation to subclasses. The client of the Factory Method never needs to know the concrete class that has been instantiated and returned. Its client needs to know only about the published abstract interface.

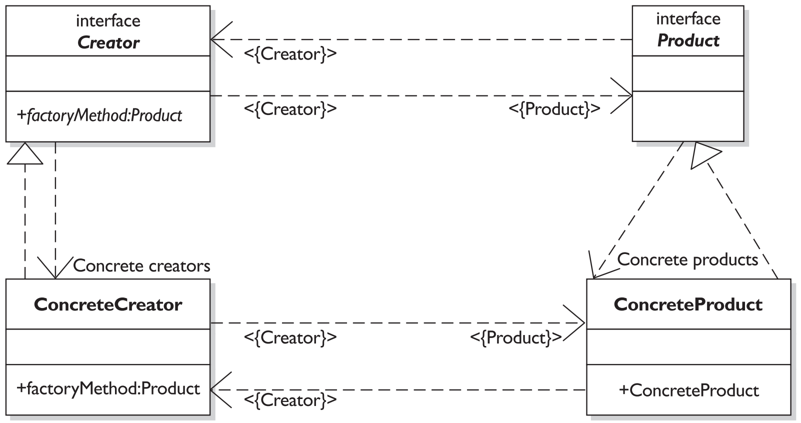
The Factory Method pattern is also known as *Virtual Constructor*. Figure 5-3 shows the UML.

Benefits Following is a list of benefits of using the Factory Method pattern:

- It removes the need to bind application-specific classes into the code. The code interacts solely with the resultant interface, so it will work with any classes that implement that interface.
- Because creating objects inside a class is more flexible than creating an object directly, it enables the subclass to provide an extended version of an object.

FIGURE 5-3

UML for the Factory Method pattern



Applicable Scenarios The following scenarios are most appropriate for the Factory Method pattern:

- A class is not able to anticipate the class of objects it needs to create.
- A class wants its subclasses to specify the objects it instantiates.
- Classes assign responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

Java EE Technology Features and Java SE API Associations The Java EE technology features associated with the Factory Method pattern are listed here:

- *javax.ejb.EJBHome*
- *javax.ejb.EJBLocalHome*
- *javax.jms.QueueConnectionFactory*
- *javax.jms.TopicConnectionFactory*

The Java SE APIs have many classes and interfaces that are associated with the Factory Method pattern. Here are some examples:

- *java.text.Collator*
- *java.net.ContentHandlerFactory*
- *javax.naming.spi.InitialContextFactory*
- *javax.net.SocketFactory*

Example Code Following is some example Java code that demonstrates the Factory Method pattern:

```
package javaee.architect.FactoryMethod;
public class FactoryMethodPattern {
  public static void main(String[] args) {
    System.out.println("FactoryMethod Pattern Demonstration.");
    System.out.println("--------------------------------------");
    // Create creator, which uses the FactoryMethod
    CreatorIF creator = new ConcreteCreator();
    // Create trade via factory method
    TradeIF trade = creator.factoryMethod();
    // Call trade action method
    trade.action();
    System.out.println();
  }
}
```

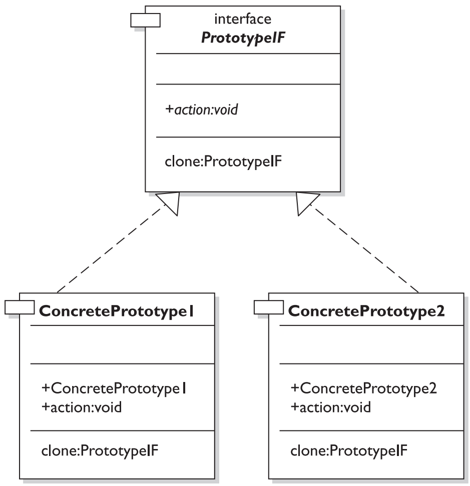
```
package javaee.architect.FactoryMethod;
public class ConcreteCreator implements Creator {
  public TradeIF factoryMethod() {
    return new ConcreteTrade();
  }
}

package javaee.architect.FactoryMethod;
public class ConcreteTrade implements TradeIF {
  public void action() {
    System.out.println("ConcreteTrade.action() called.");
  }
}

package javaee.architect.FactoryMethod;
public interface CreatorIF {
  public abstract TradeIF factoryMethod();
}

package javaee.architect.FactoryMethod;
public interface TradeIF {
  public void action();
}
```

Prototype

The Prototype pattern's intent is to specify the kinds of objects that need to be created using a prototypical instance, and then be able to create new objects by copying this prototype. The copying of objects in Java is typically done by the `clone()` method of *java.lang.Object*. The UML is shown in Figure 5-4.

FIGURE 5-4

UML for the Prototype pattern



Benefits Following are the benefits of using the Prototype pattern:

- It lets you add or remove objects at runtime.
- It lets you specify new objects by varying its values or structure.
- It reduces the need for subclassing.
- It lets you dynamically configure an application with classes.

Applicable Scenarios The following scenarios are most appropriate for the Prototype pattern:

- The classes to instantiate are specified at runtime.
- You need to avoid building a class hierarchy of factories that parallels the hierarchy of objects.
- Instances of the class have one of only a few different combinations of state.

Java SE API Association The Java SE API associated with the Prototype pattern is *java.lang.Object*.

Example Code The following is example Java code for demonstrating the Prototype pattern. There are two viewpoints on the Prototype pattern. The first is that it is there to simplify creating new instances of objects without knowing their concrete class. The second is it is there to simplify creating exact copies (or clones) of an original object.

The following example does not contain any state information in the objects prior to the call to `getClone()`. It demonstrates the first form of Prototype.

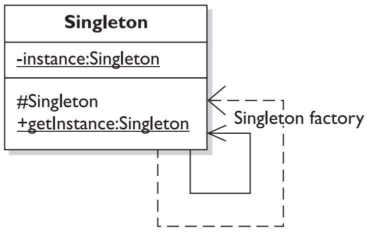
```
package javaee.architect.Prototype;
public class PrototypePattern {
  public static void main(String[] args) {
    System.out.println("Prototype Pattern Demonstration.");
    System.out.println("--------------------------------");
    // Create prototypes
    System.out.println("Constructing prototypes.");
    PrototypeIF prototype1 = new ConcretePrototype1();
    PrototypeIF prototype2 = new ConcretePrototype2();
    // Get clones from prototypes
    System.out.println("Constructing clones from prototypes.");
    PrototypeIF clone1 = prototype1.getClone();
    PrototypeIF clone2 = prototype2.getClone();
    // Call actions on the clones
    System.out.println("Calling actions on the clones.");
    clone1.action();
    clone2.action();
    System.out.println();
  }
}

package javaee.architect.Prototype;
public class ConcretePrototype1 implements PrototypeIF {
  public ConcretePrototype1() {
    System.out.println("ConcretePrototype1 constructed.");
  }
  public PrototypeIF getClone() {
    // if required, put deep copy code here
    return new ConcretePrototype1();
  }
  public void action() {
    System.out.println("ConcretePrototype1.action() called");
  }
}

package javaee.architect.Prototype;
public class ConcretePrototype2 implements PrototypeIF {
  public ConcretePrototype2() {
    System.out.println("ConcretePrototype2 constructed.");
  }
  public PrototypeIF getClone() {
    // if required, put deep copy code here
    return new ConcretePrototype1();
  }
  public void action() {
    System.out.println("ConcretePrototype2.action() called.");
  }
}

package javaee.architect.Prototype;
public interface PrototypeIF {
  public PrototypeIF getClone(); // as opposed to Object.clone()
  public void action();
}
```

Singleton

The Singleton pattern's intent is to ensure that a class has only one instance and provides a global point of access to it. It ensures that all objects that use an instance of this class are using the same instance. Figure 5-5 shows the UML.

FIGURE 5-5

UML for the Singleton pattern



Benefits Following are the benefits of using the Singleton pattern:

- It controls access to a single instance of the class.
- It reduces name space usage.
- It permits refinement of operations and representation.
- It can also permit a variable number of instances.
- It is more flexible than class methods (operations).

Applicable Scenario The scenario most appropriate for the Singleton pattern is when a single instance of a class is needed and must be accessible to clients from a well-known access point.

Java SE API Association The Java SE API associated with the Singleton pattern is *java.lang.Runtime*.

Example Code The following example Java code demonstrates the Singleton pattern:

```
package javaee.architect.Singleton;
public class SingletonPattern {
  public static void main(String[] args) {
    System.out.println("Singleton Pattern Demonstration.");
    System.out.println("--------------------------------");
    System.out.println("Getting Singleton instance (s1)");
    Singleton s1 = Singleton.getInstance();
    System.out.println("s1.getInfo()="+s1.getInfo());
    System.out.println("Getting Singleton instance (s2)");
    Singleton s2 = Singleton.getInstance();
    System.out.println("s2.getInfo()="+s2.getInfo());
    System.out.println("s1.setValue(42)");
    s1.setValue(42);
    System.out.println("s1.getValue()="+s1.getValue());
    System.out.println("s2.getValue()="+s2.getValue());
    System.out.println("s1.equals(s2)="+s1.equals(s2)
      + ", s2.equals(s1)="+s2.equals(s1));
    // The following will not compile
    // Singleton s3 = (Singleton) s1.clone();
    System.out.println();
  }
}

package javaee.architect.Singleton;
/*
 * Singletons really are "per classloader" and
 * in a Java EE application, many developers make
 * the mistake of assuming that a singleton really
 * is a singleton in a cluster of application servers.
 * This is not true!
*/
public final class Singleton {
  private static Singleton instance;
  private int value;
  private Singleton() {System.out.println("Singleton constructed.");}
  public static synchronized Singleton getInstance() {
  // if it has not been instantiated yet
  if (instance == null)
    // instantiate it here
    instance = new Singleton();
  return instance;
  }
  // remaining methods are for demo purposes
  // your singleton would have it's business
  // methods here...
  public String getInfo() {
    return getClass().getName() +
      // Uncomment line below to also see the loader
      //+", loaded by " + getClass().getClassLoader();
      ", id#" + System.identityHashCode(this);
  }
  public int getValue() {return value;}
  public void setValue(int parm) {value = parm;}
  public boolean equals(Singleton parm) {
    return (System.identityHashCode(this)
      == System.identityHashCode(parm));
  }
}
```

GoF Structural Design Patterns

Structural patterns are concerned with composition or the organization of classes and objects, how classes inherit from each other, and how they are composed from other classes.

Common Structural patterns include Adapter, Proxy, and Decorator patterns. These patterns are similar in that they introduce a level of indirection between a client class and a class it wants to use. Their intents are different, however. Adapter uses indirection to modify the interface of a class to make it easier for a client class to use it. Decorator uses indirection to add behavior to a class, without unduly affecting the client class. Proxy uses indirection transparently to provide a stand-in for another class.

The following Structural patterns are described by GoF:

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

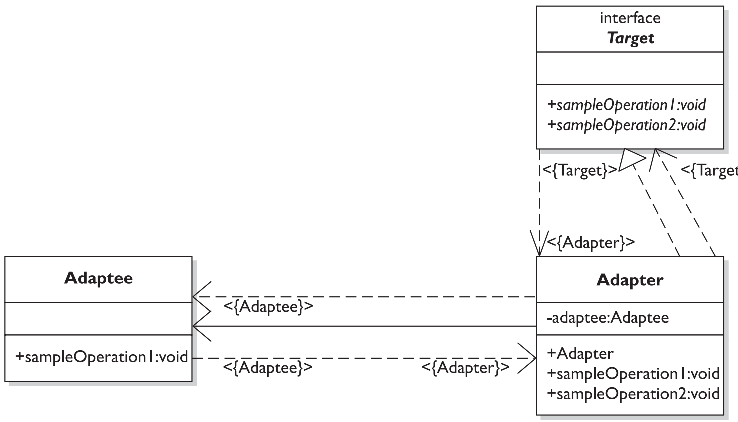
Adapter

The Adapter pattern converts the interface of a class into an interface that a client requires. It acts as an intermediary and lets classes work together that couldn't otherwise because of an incompatible interface.

The Adapter pattern is also known as *Wrapper*. The UML is shown in Figure 5-6.

FIGURE 5-6

UML for the Adapter pattern



Benefits Following are the benefits of using the Adapter pattern:

- It allows two or more previously incompatible objects to interact.
- It allows reusability of existing functionality.

Applicable Scenarios The following scenarios are most appropriate for the Adapter pattern:

- An object needs to utilize an existing class with an incompatible interface.
- You want to create a reusable class that cooperates with classes that don't necessarily have compatible interfaces.
- You need to use several existing subclasses but do not want to adapt their interfaces by subclassing each one.

Java EE Technology Feature and Java SE API Association The Java EE technology feature associated with the Adapter pattern is Java Connector Architecture (JCA), from an architectural viewpoint.

The Java SE API associated with the Adapter pattern is *java.awt.event.ComponentAdapter*.

Example Code The following example Java code demonstrates the Adapter pattern:

```
package javaee.architect.Adapter;
public class AdapterPattern {
  public static void main(String[] args) {
    System.out.println("Adapter Pattern Demonstration.");
    System.out.println("------------------------------");
    // Create targets.
    System.out.println("Creating targets.");
    TargetIF target1 = new AdapterByClass();
    TargetIF target2 = new AdapterByObject();
    // Call target requests
    System.out.println("Calling targets.");
    System.out.println("target1.newRequest()->"+target1.newRequest());
    System.out.println("target2.newRequest()->"+target2.newRequest());
    System.out.println();
  }
}
```

```
package javaee.architect.Adapter;
public class Adaptee {
  public Adaptee() {
    System.out.println("Adaptee constructed.");
  }
  public String oldRequest() {
    return "Adaptee.oldRequest() called.";
  }
}

package javaee.architect.Adapter;
public class AdapterByClass extends Adaptee implements TargetIF {
  public AdapterByClass() {
    System.out.println("AdapterByClass constructed.");
  }
  public String newRequest() {
    return oldRequest();
  }
}

package javaee.architect.Adapter;
public class AdapterByObject implements TargetIF {
  private Adaptee adaptee;
  public AdapterByObject() {
    System.out.println("AdapterByObject constructed.");
  }
  public String newRequest() {
    // Create an Adaptee object if it doesn't exist yet
    if (adaptee == null) { adaptee = new Adaptee(); }
    return adaptee.oldRequest();
  }
}

package javaee.architect.Adapter;
public interface TargetIF {
  public String newRequest();
}
```

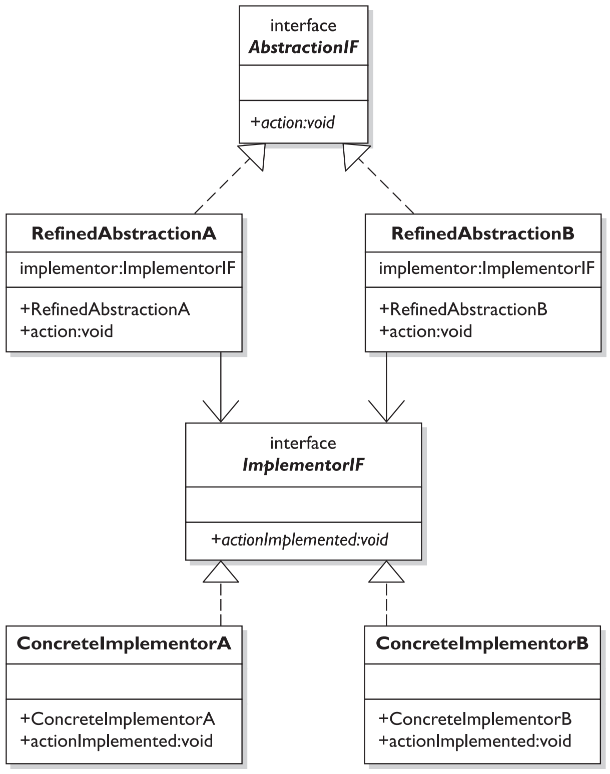
Bridge

The Bridge pattern's intent is to decouple the functional abstraction from the implementation so that the two can be changed and can vary independently.

The Bridge pattern is also known as *Handle/Body*. The UML is shown in Figure 5-7.

FIGURE 5-7

UML for the Bridge pattern



Benefits Following is a list of benefits of using the Bridge pattern:

- It enables the separation of implementation from the interface.
- It improves extensibility.
- It allows the hiding of implementation details from the client.

Applicable Scenarios The following scenarios are most appropriate for the Bridge pattern:

- You want to avoid a permanent binding between the functional abstraction and its implementation.
- Both the functional abstraction and its implementation need to be extended using subclasses.
- Changes to the implementation should not impact the client (not even a recompile).

Example Code The following example Java code demonstrates the Bridge pattern:

```
package javaee.architect.Bridge;
public class BridgePattern {
  public static void main(String[] args) {
    System.out.println("Bridge Pattern Demonstration.");
    System.out.println("-----------------------------");
    System.out.println("Constructing SportsCar and EconomyCar.");
    AbstractionIF car1 = new SportsCar ();
    AbstractionIF car2 = new EconomyCar();
    System.out.println(
      "Calling action() on SportsCar and EconomyCar.");
    car1.action();
    car2.action();
    System.out.println();
  }
}

package javaee.architect.Bridge;
public interface AbstractionIF {
  public void action();
}

package javaee.architect.Bridge;
public class SportsCarImplementor implements ImplementorIF {
  public SportsCarImplementor() {
    System.out.println("SportsCarImplementor constructed.");
  }
  public void actionImplemented() {
    System.out.println("SportsCarImplementor.actionImplemented() called.");
  }
}

package javaee.architect.Bridge;
public class EconomyCarImplementor implements ImplementorIF {
  public EconomyCarImplementor() {
    System.out.println("EconomyCarImplementor constructed.");
  }
  public void actionImplemented() {
    System.out.println("EconomyCarImplementor.actionImplemented() called.");
  }
}
```

```
package javaee.architect.Bridge;
public interface ImplementorIF {
  public void actionImplemented();
}

package javaee.architect.Bridge;
public class SportsCar implements AbstractionIF {
  ImplementorIF implementor = new SportsCarImplementor();
  public SportsCar() {
    System.out.println("SportsCar constructed.");
  }
  public void action() {
    implementor.actionImplemented();
  }
}

package javaee.architect.Bridge;
public class EconomyCar implements AbstractionIF {
  ImplementorIF implementor = new EconomyCarImplementor();
  public EconomyCar() {
    System.out.println("EconomyCar constructed.");
  }
  public void action() {
    implementor.actionImplemented();
  }
}
```

Composite

The Composite pattern's intent is to allow clients to operate in a generic manner on objects that may or may not represent a hierarchy of objects.

The UML is shown in Figure 5-8.

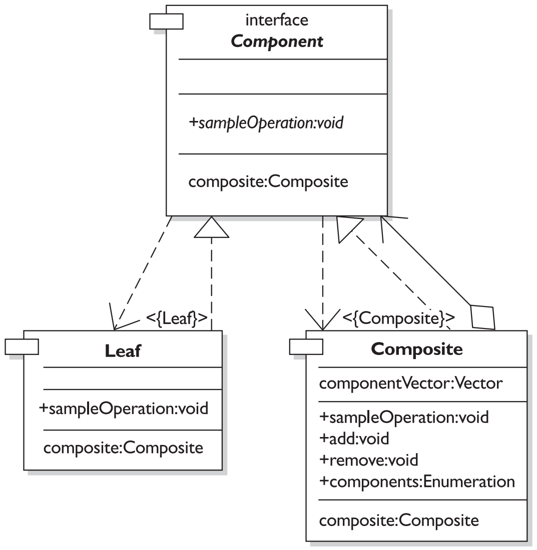
Benefits Following are benefits of using the Composite pattern:

- It defines class hierarchies consisting of primitive and complex objects.
- It makes it easier for you to add new kinds of components.
- It provides flexibility of structure with a manageable interface.

Applicable Scenarios The following scenarios are most appropriate for the Composite pattern:

FIGURE 5-8

UML for the Composite pattern



- You want to represent a full or partial hierarchy of objects.
- You want clients to be able to ignore the differences between the varying objects in the hierarchy.
- The structure is dynamic and can have any level of complexity: for example, using the Composite View from the J2EE Patterns Catalog, which is useful for portal applications.

Example Code The following example Java code demonstrates the Composite pattern:

```
package javaee.architect.Composite;
public class CompositePattern {
  public static void main(String[] args) {
    System.out.println("Composite Pattern Demonstration.");
    System.out.println("-------------------------------");
    System.out.println("Creating leaves, branches and trunk");
    // Create leaves
    Component leaf1 = new Leaf("    leaf#1");
    Component leaf2 = new Leaf("    leaf#2");
    Component leaf3 = new Leaf("    leaf#3");
    // Create branches
    Component branch1 = new Composite("  branch1");
    Component branch2 = new Composite("  branch2");
    // Create trunk
    Component trunk = new Composite("trunk");
    // Add leaf1 and leaf2 to branch1
    branch1.add(leaf1);
    branch1.add(leaf2);
    // Add branch1 to trunk
    trunk.add(branch1);
    // Add leaf3 to branch2
    branch2.add(leaf3);
    // Add branch2 to trunk
    trunk.add(branch2);
    // Show trunk composition
    System.out.println("Displaying trunk composition:");
    trunk.display();
    // Remove branch1 and branch2 from trunk
    trunk.remove(branch1);
    trunk.remove(branch2);
    // Show trunk composition now
    System.out.println("Displaying trunk composition now:");
    trunk.display();
    System.out.println();
  }
}

package javaee.architect.Composite;
public abstract class Component {
  public abstract void display();
  public void add(Component c) { // override in concrete class; }
  public void remove(Component c) { // override in concrete class; }
  public Component getChild(int index) { return null; }
  public String getName() { return null; }
}

package javaee.architect.Composite;
import java.util.*;
public class Composite extends Component {
  String name = null;
  List children = new ArrayList();
  public Composite(String parm) {
    this.name = parm;
    System.out.println(parm.trim()+" constructed.");
  }
  public String getName() { return name; }
  public Component getChild(int parm) {
    Component child;
    try {child = (Component) children.get(parm);}
    catch (IndexOutOfBoundsException ioobe) {child = null;}
    return child;
  }
  public void add(Component parm) {
    try {
      System.out.println("Adding "+parm.getName().trim()
        +" to "+this.getName().trim());
      children.add(parm);
    }
    catch (Exception e) {System.out.println(e.getMessage());}
  }
  public void remove(Component parm) {
    try {
      System.out.println("Removing "+parm.getName().trim()
        +" from "+this.getName().trim());
      children.remove(parm);}
    catch (Exception e) {System.out.println(e.getMessage());}
  }
  public void display() {
    Iterator iterator = children.iterator();
    System.out.println(this.getName()
      +(iterator.hasNext()?" with the following: ":" that is bare."));
    while (iterator.hasNext()) {((Component) iterator.next()).display();}
  }
}

package javaee.architect.Composite;
public class Leaf extends Component {
  private String name;
  public Leaf(String parm) {
    this.name = parm;
    System.out.println(parm.trim()+" constructed.");
  }
  public void display() {
    System.out.println(this.getName());
  }
  public String getName() {
    return name;
  }
}
```

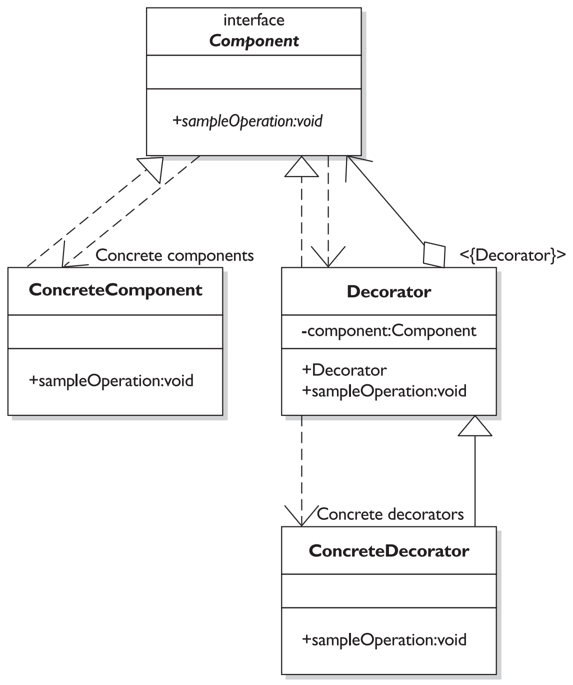


FIGURE 5-9

UML for the Decorator pattern

Decorator

An alternative to subclassing to extend functionality, the Decorator pattern's intent is to attach flexible additional responsibilities to an object dynamically. The Decorator pattern uses composition instead of inheritance to extend the functionality of an object at runtime.

The Decorator pattern is also known as *Wrapper*. The UML is shown in Figure 5-9.

Benefits Following is a list of benefits of using the Decorator pattern:

- It provides greater flexibility than static inheritance.
- It avoids the need to place feature-laden classes higher up the hierarchy.
- It simplifies coding by allowing you to develop a series of functionality-targeted classes, instead of coding all of the behavior into the object.
- It enhances the extensibility of the object, because changes are made by coding new classes.

Applicable Scenarios The following scenarios are most appropriate for the Decorator pattern:

- You want to transparently and dynamically add responsibilities to objects without affecting other objects.
- You want to add responsibilities to an object that you may want to change in the future.
- Extending functionality by subclassing is no longer practical.

Java EE Technology Feature and Java SE API Association The Java EE technology feature associated with the Decorator pattern is *javax.ejb.EJBObject*.

The Java SE API associated with the Decorator pattern is *java.io.BufferedReader*.

Example Code The following example Java code demonstrates the Decorator pattern:

```
package javaee.architect.Decorator;
public class DecoratorPattern {
  public static void main(String[] args) {
    System.out.println("Decorator Pattern Demonstration.");
    System.out.println("--------------------------------");
    // Create object decorated with A
    System.out.println("Creating component decorated with A.");
    ComponentIF decorated1 = new ConcreteDecoratorA();
    // Call action on object decorated with A
    System.out.println("Calling action() on component decorated with A.");
    decorated1.action();
    // Create object decorated with B
    System.out.println("Creating component decorated with B.");
    ComponentIF decorated2 = new ConcreteDecoratorB();
    // Call action on object decorated with B
    System.out.println("Calling action() on component decorated with B.");
    decorated2.action();
    System.out.println();
  }
}

package javaee.architect.Decorator;
public interface ComponentIF {
  public void action();
}
```

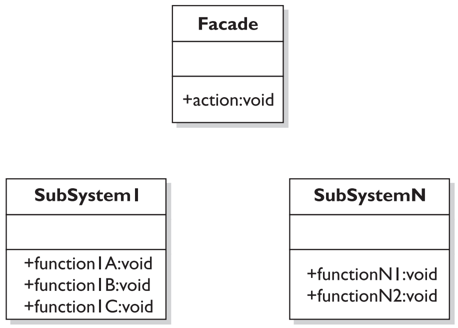
```
package javaee.architect.Decorator;
public class ConcreteComponent implements ComponentIF {
  public void action() {
    System.out.println("ConcreteComponent.action() called.");
  }
}

package javaee.architect.Decorator;
public class ConcreteDecoratorA extends Decorator {
  String addedVariable;
  public void action() {
    super.action();
    System.out.println("ConcreteDecoratorA.action() called.");
    addedVariable = "extra";
    System.out.println("ConcreteDecoratorA.addedVariable="+addedVariable);
  }
}

package javaee.architect.Decorator;
public class ConcreteDecoratorB extends Decorator {
  public void action() {
    super.action();
    System.out.println("ConcreteDecoratorB.action() called.");
    addedMethod();
  }
  private void addedMethod() {
    System.out.println("ConcreteDecoratorB.addedMethod() called.");
  }
}

package javaee.architect.Decorator;
public class Decorator implements ComponentIF {
  ComponentIF component = new ConcreteComponent();
  public void action() {
    component.action();
  }
}
```

Facade

The Facade pattern's intent is to provide a unified and simplified interface to a set of interfaces in a subsystem. The Facade pattern describes a higher-level interface that makes the subsystem(s) easier to use. Practically, every Abstract Factory is a type of Facade. Figure 5-10 shows the UML.

FIGURE 5-10

UML for the Facade pattern



Benefits Following is a list of benefits of using the Facade pattern:

- It provides a simpler interface to a complex subsystem without reducing the options provided by the subsystem.
- It shields clients from the complexity of the subsystem components.
- It promotes looser coupling between the subsystem and its clients.
- It reduces the coupling between subsystems provided that every subsystem uses its own Facade pattern and other parts of the system use the Facade pattern to communicate with the subsystem.

Applicable Scenarios The following scenarios are most appropriate for the Facade pattern:

- You need to provide a simple interface to a complex subsystem.
- Several dependencies exist between clients and the implementation classes of an abstraction.
- Layering the subsystems is necessary or desired.

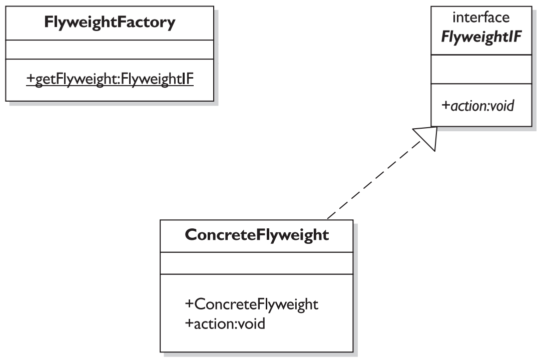
Java SE API Association The Java SE API associated with the Facade pattern is *java.net.URL*.

Example Code The following example Java code demonstrates the Facade pattern:

```
package javaee.architect.Facade;
public class FacadePattern {
  public static void main(String[] args) {
    System.out.println("Facade Pattern Demonstration.");
    System.out.println("-----------------------------");
    // Construct and call Facade
    System.out.println("Constructing facade.");
    Facade facade = new Facade();
    System.out.println("Calling facade.processOrder().");
    facade.processOrder();
    System.out.println();
  }
}

package javaee.architect.Facade;
public class Facade {
  public void processOrder() {
    // Call methods on sub-systems to complete the process
    SubSystem1 subsys1 = new SubSystem1();
    subsys1.getCustomer();
    subsys1.getSecurity();
    subsys1.priceTransaction();
    SubSystemN subsysN = new SubSystemN();
    subsysN.checkBalances();
    subsysN.completeOrder();
  }
}

package javaee.architect.Facade;
public class SubSystem1 {
  public void getCustomer() {
    // Place functionality here...
    System.out.println("SubSystem1.getCustomer() called.");}
  public void getSecurity() {
    // Place functionality here...
    System.out.println("SubSystem1.getSecurity() called.");}
  public void priceTransaction() {
    // Place functionality here...
    System.out.println("SubSystem1.priceTransaction() called.");}
}

package javaee.architect.Facade;
public class SubSystemN {
  public void checkBalances() {
    // Place functionality here...
    System.out.println("SubSystemN.checkBalances() called.");}
  public void completeOrder() {
    // Place functionality here...
    System.out.println("SubSystemN.completeOrder() called.");}
}
```

FIGURE 5-11

UML for the Flyweight pattern



Flyweight

The Flyweight pattern's intent is to utilize sharing to support large numbers of fine-grained objects in an efficient manner. Figure 5-11 shows the UML.

Benefits Following are benefits of using the Flyweight pattern:

- It reduces the number of objects to deal with.
- It reduces the amount of memory and storage devices required if the objects are persisted.

Applicable Scenarios The following scenarios are most appropriate for the Flyweight pattern:

- An application uses a considerable number of objects.
- The storage costs are high because of the quantity of objects.
- The application does not depend on object identity.

Java SE API Association The Java SE API associated with the Flyweight pattern is *java.lang.String*.

Example Code The following example Java code demonstrates the Flyweight pattern:

```
package javaee.architect.Flyweight;
public class FlyweightPattern {
  public static void main(String[] args) {
    System.out.println("Flyweight Pattern Demonstration.");
    System.out.println("--------------------------------");
    // Create states
    State stateF = new State(false);
    State stateT = new State(true);
    // Get reference to (and in doing so create) flyweight
    FlyweightIF myfwkey1 = FlyweightFactory.getFlyweight("myfwkey");
    // Get new reference to the same flyweight
    FlyweightIF myfwkey2 = FlyweightFactory.getFlyweight("myfwkey");
    // Call action on both references
    System.out.println("Call flyweight action with state=false");
    myfwkey1.action(stateF);
    System.out.println("Call flyweight action with state=true");
    myfwkey2.action(stateT);
    System.out.println();
  }
}

package javaee.architect.Flyweight;
public class ConcreteFlyweight implements FlyweightIF {
  // Add state to the concrete flyweight.
  private boolean state;
  public ConcreteFlyweight(State parm) {
    this.state = parm.getState();
  }
  public void action(State parm) {
    // Display internal state and state passed by client.
    System.out.println("ConcreteFlyweight.action("
      +parm.getState()+") called.");
    this.state = parm.getState();
    System.out.println("ConcreteFlyweight.state = "
      + this.state);
  }
}

package javaee.architect.Flyweight;
import java.util.*;
public class FlyweightFactory {
  private static Map map = new HashMap();
  public static FlyweightIF getFlyweight(String parm) {
    // Return the Flyweight if it exists,
    // or create it if it doesn’t.
    FlyweightIF flyweight = null;
    try {
      if (map.containsKey(parm)) {
        // Return existing flyweight
        flyweight = (FlyweightIF) map.get(parm);
      } else {
        // Create flyweight with a 'true' state
        flyweight = new ConcreteFlyweight(new State(true));
        map.put(parm, flyweight);
        System.out.println("Created flyweight "+parm+" with state=true");
        System.out.println("");
      }
    } catch (ClassCastException cce) {
      System.out.println(cce.getMessage());
    }
    return flyweight;
  }
}

package javaee.architect.Flyweight;
public interface FlyweightIF {
      // method to receive and act on extrinsic state.
  public void action(State parm);
}

package javaee.architect.Flyweight;
public class State {
  private boolean state;
  public State(boolean parm) {this.state = parm;}
  public boolean getState() {return state;}
}
```

Proxy

The Proxy pattern's intent is to provide a surrogate or placeholder for another object to control access to it. The most common implementations are remote and virtual proxy.

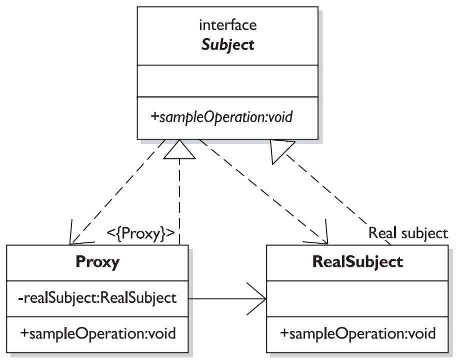
The Proxy pattern is also known as *Surrogate*. Figure 5-12 shows the UML.

Benefits Following is a list of benefits of using the Proxy pattern:

- The remote proxy can shield the fact that the implementation resides in another address space.
- The virtual proxy can perform optimizations—for example, by creating objects on demand.

FIGURE 5-12

UML for the Proxy pattern



Applicable Scenario The Proxy pattern is appropriate when a more versatile or sophisticated reference to an object, rather than a simple pointer, is needed.

Java EE Technology Feature The Java EE technology feature associated with the Proxy pattern is *javax.ejb.EJBObject* (EJB remote reference) in a structural sense. Actually the "stub" object in the client's address space provides the proxy.

Example Code The following Java code demonstrates the Proxy pattern:

```
package javaee.architect.Proxy;
public class ProxyPattern {
  public static void main(String[] args) {
    System.out.println("Proxy Pattern Demonstration.");
    System.out.println("---------------------------");
    // Create service proxy (instantiates service too)
    System.out.println("Creating proxy to service.");
    ServiceIF proxy = new Proxy();
    // Call action method on service via proxy
    System.out.println("Calling action method on proxy.");
    proxy.action();
    System.out.println();
  }
}

package javaee.architect.Proxy;
public class Proxy implements ServiceIF {
  // Proxy to be the service
  private Service service = new Service();
  public void action() {
    service.action();
  }
}

package javaee.architect.Proxy;
public class Service implements ServiceIF {
  // Service to be proxied
  public Service() {
    System.out.println("Service constructed.");
  }
  public void action() {
    System.out.println("Service.action() called.");
  }
}

package javaee.architect.Proxy;
public interface ServiceIF {
  // Interface for Service and Proxy
  public void action();
}
```

GoF Behavioral Design Patterns

Behavioral patterns are concerned with the interaction and responsibility of objects. They help make complex behavior manageable by specifying the responsibilities of objects and the ways they communicate with each other.

The following Behavioral patterns are described by GoF:

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

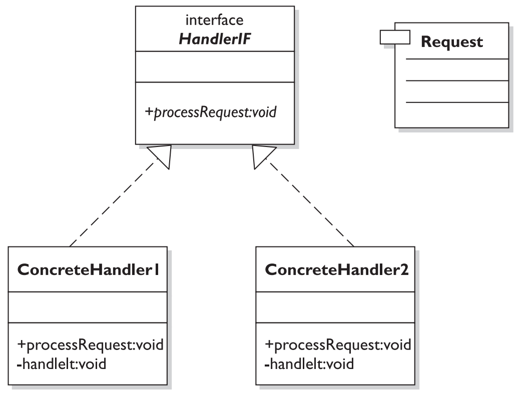


FIGURE 5-13

UML for the Chain of Responsibility pattern

Chain of Responsibility

The Chain of Responsibility pattern's intent is to avoid coupling the sender of a request to its receiver by giving multiple objects a chance to handle the request.

The request is passed along the chain of receiving objects until an object processes it. Figure 5-13 shows the UML.

Benefits Following are the benefits of using the Chain of Responsibility pattern:

- It reduces coupling.
- It adds flexibility when assigning responsibilities to objects.
- It allows a set of classes to act as one; events produced in one class can be sent to other handler classes within the composition.

Applicable Scenarios The following scenarios are most appropriate for the Chain of Responsibility pattern:

- More than one object can handle a request, and the handler is unknown.
- A request is to be issued to one of several objects, and the receiver is not specified explicitly.
- The set of objects able to handle the request is to be specified dynamically.

Java EE Technology Feature The Java EE technology feature associated with the Chain of Responsibility pattern is *RequestDispatcher* in the servlet/JSP API.

Example Code The following example Java code demonstrates the Chain of Responsibility pattern:

```
package javaee.architect.ChainOfResponsibility;
public class ChainOfResponsibilityPattern {
  public static void main(String[] args) {
    System.out.println("Chain Of Responsibility Pattern Demonstration.");
    System.out.println("----------------------------------------------");
    try {
      // Create Equity Order request.
      System.out.println("Creating Equity Order request.");
      Request equityOrderRequest = new Request(Request.EQUITY_ORDER);
      // Create Bond Order request.
      System.out.println("Creating Bond Order request.");
      Request bondOrderRequest = new Request(Request.BOND_ORDER);
      // Create a request handler.
      System.out.println("Creating 1st handler.");
      HandlerIF handler = new ConcreteHandler1();
      // Process the Equity Order.
      System.out.println("Calling 1st handler with Equity Order.");
      handler.processRequest(equityOrderRequest);
      // Process the Bond Order.
      System.out.println("Calling 1st handler with Bond Order");
      handler.processRequest(bondOrderRequest);
    } catch (Exception e) {System.out.println(e.getMessage());}
    System.out.println();
  }
}

package javaee.architect.ChainOfResponsibility;
public class ConcreteHandler1 implements HandlerIF {
  public void processRequest(Request parm) {
    // Start the processing chain here...
    switch (parm.getType()) {
      case Request.EQUITY_ORDER: // This object processes equity orders
        handleIt(parm);          // so call the function to handle it.
        break;
      case Request.BOND_ORDER:   // Another object processes bond orders so
        System.out.println("Creating 2nd handler."); // pass request along.
        new ConcreteHandler2().processRequest(parm);
        break;
    }
  }
  private void handleIt(Request parm) {
    System.out.println("ConcreteHandler1 has handled the processing.");
  }
}
```

```
package javaee.architect.ChainOfResponsibility;
public class ConcreteHandler2 implements HandlerIF {
  public void processRequest(Request parm) {
    // You could add on to the processing chain here...
    handleIt(parm);
  }
  private void handleIt(Request parm) {
    System.out.println("ConcreteHandler2 has handled the processing.");
  }
}

package javaee.architect.ChainOfResponsibility;
public interface HandlerIF {
  public void processRequest(Request request);
}

package javaee.architect.ChainOfResponsibility;
public class Request {
  // The universe of known requests that can be handled.
  public final static int EQUITY_ORDER = 100;
  public final static int BOND_ORDER   = 200;
  // This objects type of request.
  private int type;
  public Request(int parm) throws Exception {
    // Validate the request type with the known universe.
    if ((parm == EQUITY_ORDER) || (parm == BOND_ORDER))
      // Store this request type.
      this.type = parm;
    else
      throw new Exception("Unknown Request type "+parm+".");
  }
  public int getType() {
    return type;
  }
}
```

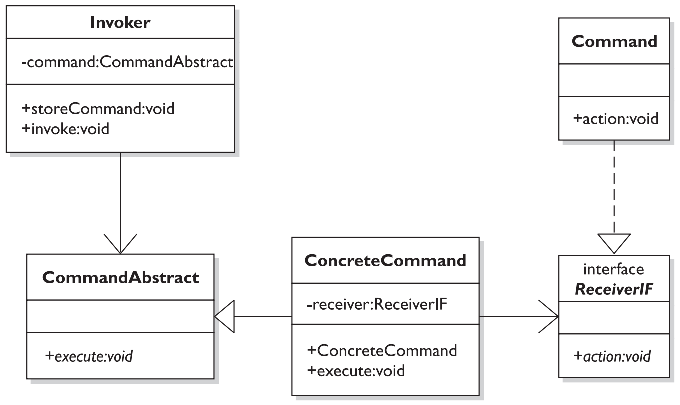
Command

The Command pattern's intent is to encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support rollback types of operations.

The Command pattern is also known as *Action or Transaction*. The UML is shown in Figure 5-14.

FIGURE 5-14

UML for the Command pattern



Benefits Following is a list of benefits of using the Command pattern:

- It separates the object that invokes the operation from the object that actually performs the operation.
- It simplifies adding new commands, because existing classes remain unchanged.

Applicable Scenarios The following scenarios are most appropriate for the Command pattern:

- You need to parameterize objects according to an action to perform.
- You create, queue, and execute requests at different times.
- You need to support rollback, logging, or transaction functionality.

Java EE Technology Feature These are the Java EE technology features associated with the Command pattern:

- MessageBeans invoke business logic based on content of messages dispatched to them.
- Servlets/JSPs are invoked corresponding to the type of HTTP request that is received by the web container.

Example Code The following example Java code demonstrates the Command pattern:

```
package javaee.architect.Command;
public class CommandPattern {
  public static void main(String[] args) {
    System.out.println("Command Pattern Demonstration.");
    System.out.println("------------------------------");
    // Create receiver objects.
    System.out.println("Creating receivers.");
    ReceiverIF order = new Order();
    ReceiverIF trade = new Trade();
    // Create commands passing in receiver objects.
    System.out.println("Creating commands.");
    CommandAbstract cmdOrder = new ConcreteCommand(order);
    CommandAbstract cmdTrade = new ConcreteCommand(trade);
    // Create invokers.
    System.out.println("Creating invokers.");
    Invoker invOrder = new Invoker();
    Invoker invTrade = new Invoker();
    // Storing commands in invokers respectively.
    System.out.println("Storing commands in invokers.");
    invOrder.storeCommand(cmdOrder);
    invTrade.storeCommand(cmdTrade);
    // Call invoke on the invoker to execute the command.
    System.out.println("Invoking the invokers.");
    invOrder.invoke();
    invTrade.invoke();
    System.out.println();
  }
}

package javaee.architect.Command;
abstract class CommandAbstract {
  public abstract void execute();
}

package javaee.architect.Command;
public class ConcreteCommand extends CommandAbstract {
  // The binding between action and receiver
  private ReceiverIF receiver;
  public ConcreteCommand(ReceiverIF receive) {
    this.receiver = receive;
  }
  public void execute() {
      receiver.action();
  }
}
```

```
package javaee.architect.Command;
public class Invoker {
  private CommandAbstract command;
  public void storeCommand(CommandAbstract cmd) {
    this.command = cmd;
  }
  public void invoke() {
    command.execute();
  }
}

package javaee.architect.Command;
public class Order implements ReceiverIF {
  public void action() {
    System.out.println("Order.action() called.");
  }
}

package javaee.architect.Command;
public interface ReceiverIF {
  public void action();
}

package javaee.architect.Command;
public class Trade implements ReceiverIF {
  public void action() {
    System.out.println("Trade.action() called.");
  }
}
```

Interpreter

The Interpreter pattern's intent is to define a representation of the grammar of a given language, along with an interpreter that uses this representation to interpret sentences in the language. The UML is shown in Figure 5-15.

Benefits Following is a list of benefits of using the Interpreter pattern:

- It is easier to change and extend the grammar.
- Implementing the grammar is straightforward.

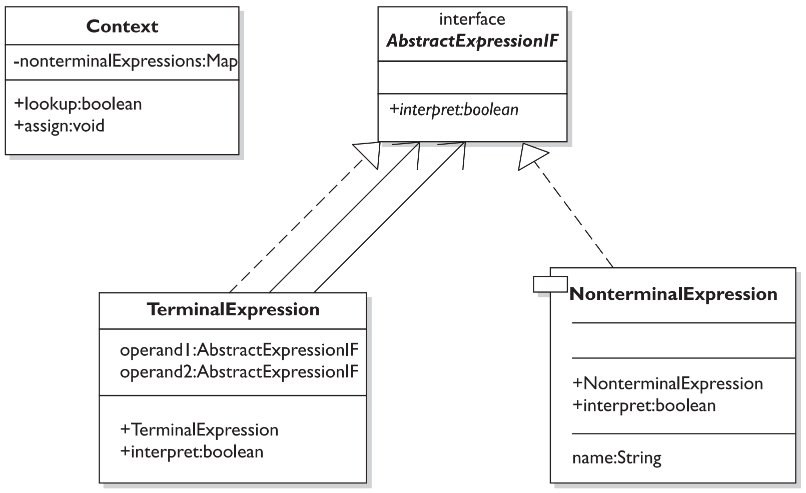


FIGURE 5-15

UML for the Interpreter pattern

Applicable Scenarios The following scenarios are most appropriate for the Interpreter pattern:

- The grammar of the language is not complicated.
- Efficiency is not a priority.

Example Code The following example Java code demonstrates the Interpreter pattern:

```
package javaee.architect.Interpreter;
import java.util.ArrayList;
import java.util.ListIterator;
import java.util.StringTokenizer;
public class InterpreterPattern {
  public static void main(String[] args) {
    System.out.println("Interpreter Pattern Demonstration.");
    System.out.println("----------------------------------");
    BookInterpreterContext bookInterpreterContext = new BookInterpreterContext();
    bookInterpreterContext.addTitle("Pickwick Papers");
    bookInterpreterContext.addTitle("Great Expectations");
    bookInterpreterContext.addTitle("Wuthering Heights");
    bookInterpreterContext.addTitle("Crossfile");
    bookInterpreterContext.addAuthor("William Shakespeare");
    bookInterpreterContext.addAuthor("Emily Bronte");
    bookInterpreterContext.addAuthor("James Marathon");
    bookInterpreterContext.addTitleAndAuthor(
      new TitleAndAuthor("Pickwick Papers", "William Shakespeare"));
    bookInterpreterContext.addTitleAndAuthor(
      new TitleAndAuthor("Great Expectations", "William Shakespeare"));
    bookInterpreterContext.addTitleAndAuthor(
      new TitleAndAuthor("Wuthering Heights", "Emily Bronte"));
    bookInterpreterContext.addTitleAndAuthor(
      new TitleAndAuthor("Crossfire", "James Marathon"));
    BookInterpreterClient bookInterpreterClient
      = new BookInterpreterClient(bookInterpreterContext);
    System.out.println("show author ->"
      + bookInterpreterClient.interpret("show author"));
    System.out.println("show title ->"
      + bookInterpreterClient.interpret("show title"));
    System.out.println("show author for title <Crossfire> ->"
      + bookInterpreterClient.interpret("show author for title <Crossfire>"));
    System.out.println("show title for author <William Shakespeare> ->"
      + bookInterpreterClient.interpret(
        "show title for author <William Shakespeare>"));
    System.out.println();
  }
}
class BookInterpreterClient {
  BookInterpreterContext bookInterpreterContext;
  public BookInterpreterClient(BookInterpreterContext parm) {
    bookInterpreterContext = parm;
  }
  // language syntax:
  // show title
  // show author
  // show title for author <author-name>
  // show author for title <title-name>
  public String interpret(String expression) {
    StringTokenizer expressionTokens = new StringTokenizer(expression);
    String currentToken;
    char mainQuery = ' ';
    char subQuery = ' ';
    String searchString = null;
    boolean forUsed = false;
    boolean searchStarted = false;
    boolean searchEnded = false;
    StringBuffer result = new StringBuffer();
    while (expressionTokens.hasMoreTokens()) {
      currentToken = expressionTokens.nextToken();
      if (currentToken.equals("show")) {
        continue;//show in all queries, not really used
      } else if (currentToken.equals("title")) {
        if (mainQuery == ' ') {
          mainQuery = 'T';
        } else  {
          if ((subQuery == ' ') && (forUsed)) {
            subQuery = 'T';
          }
        }
      } else if (currentToken.equals("author")) {
        if (mainQuery == ' ') {
          mainQuery = 'A';
        }  else {
          if ((subQuery == ' ') && (forUsed)) {
            subQuery = 'A';
          }
        }
      } else if (currentToken.equals("for")) {
        forUsed = true;
      } else if ((searchString == null) && (subQuery != ' ')
          && (currentToken.startsWith("<"))) {
        searchString = currentToken;
        searchStarted = true;
        if (currentToken.endsWith(">")) {
          searchEnded = true;
        }
      } else if ((searchStarted) && (!searchEnded)) {
        searchString = searchString + " " + currentToken;
        if (currentToken.endsWith(">")) {
          searchEnded = true;
        }
      }
    }
    if (searchString != null) {
      searchString
        = searchString.substring(1,(searchString.length() - 1));//remove <>
    }
    BookAbstractExpression abstractExpression;
    switch (mainQuery) {
      case 'A' : {
       switch (subQuery) {
         case 'T' : {
           abstractExpression = new BookAuthorTitleExpression(searchString);
           break;
         } default : {
           abstractExpression = new BookAuthorExpression();
           break;
         }
       }
       break;
      } case 'T' : {
       switch (subQuery) {
         case 'A' : {
           abstractExpression = new BookTitleAuthorExpression(searchString);
           break;
         } default : {
           abstractExpression = new BookTitleExpression();
           break;
         }
       }
       break;
      } default : return result.toString();
    }
    result.append(abstractExpression.interpret(bookInterpreterContext));
    return result.toString();
  }
}
class BookInterpreterContext {
  private ArrayList titles = new ArrayList();
  private ArrayList authors = new ArrayList();
  private ArrayList titlesAndAuthors = new ArrayList();
  public void addTitle(String title) {titles.add(title);}
  public void addAuthor(String author) {authors.add(author);}
  public void addTitleAndAuthor(TitleAndAuthor titleAndAuthor)
    {titlesAndAuthors.add(titleAndAuthor);}
  public ArrayList getAllTitles() {return titles;}
  public ArrayList getAllAuthors() {return authors;}
  public ArrayList getAuthorsForTitle(String titleIn) {
    ArrayList authorsForTitle = new ArrayList();
    TitleAndAuthor tempTitleAndAuthor;
    ListIterator titlesAndAuthorsIterator = titlesAndAuthors.listIterator();
    while (titlesAndAuthorsIterator.hasNext()) {
      tempTitleAndAuthor = (TitleAndAuthor)titlesAndAuthorsIterator.next();
      if (titleIn.equals(tempTitleAndAuthor.getTitle())) {
        authorsForTitle.add(tempTitleAndAuthor.getAuthor());
      }
    }
    return authorsForTitle;
  }
  public ArrayList getTitlesForAuthor(String authorIn) {
    ArrayList titlesForAuthor = new ArrayList();
    TitleAndAuthor tempTitleAndAuthor;
    ListIterator authorsAndTitlesIterator = titlesAndAuthors.listIterator();
    while (authorsAndTitlesIterator.hasNext()) {
      tempTitleAndAuthor = (TitleAndAuthor)authorsAndTitlesIterator.next();
      if (authorIn.equals(tempTitleAndAuthor.getAuthor())) {
        titlesForAuthor.add(tempTitleAndAuthor.getTitle());
      }
    }
    return titlesForAuthor;
  }
}
abstract class BookAbstractExpression {
  public abstract String interpret(BookInterpreterContext parm);
}
class BookAuthorExpression extends BookAbstractExpression {
  public String interpret(BookInterpreterContext parm) {
    ArrayList authors = parm.getAllAuthors();
    ListIterator authorsIterator = authors.listIterator();
    StringBuffer titleBuffer = new StringBuffer("");
    boolean first = true;
    while (authorsIterator.hasNext()) {
      if (!first) {titleBuffer.append(", ");}
      else {first = false;}
      titleBuffer.append((String)authorsIterator.next());
    }
    return titleBuffer.toString();
  }
}
class BookAuthorTitleExpression extends BookAbstractExpression {
  String title;
  public BookAuthorTitleExpression(String parm) {title = parm;}
  public String interpret(BookInterpreterContext parm) {
    ArrayList authorsAndTitles = parm.getAuthorsForTitle(title);
    ListIterator authorsAndTitlesIterator = authorsAndTitles.listIterator();
    StringBuffer authorBuffer = new StringBuffer("");
    boolean first = true;
    while (authorsAndTitlesIterator.hasNext()) {
      if (!first) {authorBuffer.append(", ");}
      else {first = false;}
      authorBuffer.append((String)authorsAndTitlesIterator.next());
    }
    return authorBuffer.toString();
  }
}
class BookTitleExpression extends BookAbstractExpression {
  public String interpret(BookInterpreterContext parm) {
    ArrayList titles = parm.getAllTitles();
    ListIterator titlesIterator = titles.listIterator();
    StringBuffer titleBuffer = new StringBuffer("");
    boolean first = true;
    while (titlesIterator.hasNext()) {
      if (!first) {titleBuffer.append(", ");}
      else {first = false;}
      titleBuffer.append((String)titlesIterator.next());
    }
    return titleBuffer.toString();
  }
}
class BookTitleAuthorExpression extends BookAbstractExpression {
  String title;
  public BookTitleAuthorExpression(String parm) {title = parm;}
  public String interpret(BookInterpreterContext parm) {
    ArrayList titlesAndAuthors = parm.getTitlesForAuthor(title);
    ListIterator titlesAndAuthorsIterator = titlesAndAuthors.listIterator();
    StringBuffer titleBuffer = new StringBuffer("");
    boolean first = true;
    while (titlesAndAuthorsIterator.hasNext()) {
      if (!first) {titleBuffer.append(", ");}
      else {first = false;}
      titleBuffer.append((String)titlesAndAuthorsIterator.next());
    }
    return titleBuffer.toString();
  }
}
class TitleAndAuthor {
  private String title;
  private String author;
  public TitleAndAuthor(String parm1, String parm2) {
    title = parm1;
    author = parm2;
  }
  public String getTitle() {return title;}
  public String getAuthor() {return author;}
}
```

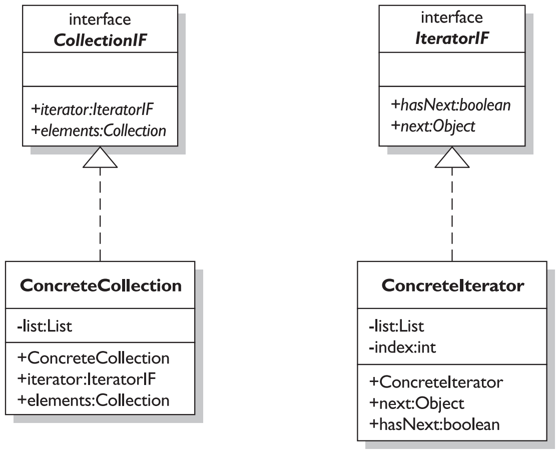
Iterator

The Iterator pattern's intent is to provide a way to access the elements of an aggregate object sequentially without exposing its underlying implementation. *java.util.Enumeration* and *java.util.Iterator* are examples of the Iterator pattern.

The Iterator pattern is also known as Cursor. The UML is shown in Figure 5-16.

FIGURE 5-16

UML for the Iterator pattern



Benefits Following is a list of benefits of using the Iterator pattern:

- It supports variations in the traversal of a collection.
- It simplifies the interface to the collection.

Applicable Scenarios The following scenarios are most appropriate for the Iterator pattern:

- Access to a collection object is required without having to expose its internal representation.
- Multiple traversals of objects need to be supported in the collection.
- A universal interface for traversing different structures needs to be provided in the collection.

Java EE Technology Feature and Java SE API Association The Java EE technology feature associated with the Command pattern is *ValueListHandler* in the J2EE Patterns Catalog.

The Java SE APIs associated with the Iterator pattern are

- *java.util.Iterator*
- *java.util.Enumeration*

Example Code The following example Java code demonstrates the Iterator pattern:

```
package javaee.architect.Iterator;
public class IteratorPattern {
  public static void main(String[] args) {
    System.out.println("Iterator Pattern Demonstration.");
    System.out.println("-------------------------------");
    System.out.println("Building string array of books.");
    String[] books = new String[8];
    books[0] = "PowerBuilder Developers Guide, 1994";
    books[1] = "Informix Developers Guide, 1995";
    books[2] = "Informix Universal Data Option, 1996";
    books[3] = "SQL Server Developers Guide, 1999";
    books[4] = "SilverStream Success I, 1999";
    books[5] = "SilverStream Success II, 2000";
    books[6] = "J2EE Unleashed, 2001";
    books[7] = "Enterprise Architect Study Guide, 2002";
    // Turn the string array into a collection.
    System.out.println("Turning string array into a collection.");
    CollectionIF collection = new ConcreteCollection(books);
    // Get an iterator for the collection.
    System.out.println("Getting an iterator for the collection..");
    IteratorIF iterator = collection.iterator();
    // Iterate through and print each object in the list.
    System.out.println("Iterate through the list.");
    int i = 0;
    while (iterator.hasNext()) {
      System.out.println((++i)+" "+iterator.next());
    }
    System.out.println();
  }
}

package javaee.architect.Iterator;
import java.util.*;
public interface CollectionIF {
  // Interface for creating a
  // collection that needs iterating.
  public IteratorIF iterator();
  public Collection elements();

}

package javaee.architect.Iterator;
import java.util.*;
public class ConcreteCollection implements CollectionIF {
  // Builds an iterable list of elements
  private List list = new ArrayList();
  public ConcreteCollection(Object[] objectList) {
    for (int i=0; i < objectList.length; i++) {
      list.add(objectList[i]);
    }
  }
  public IteratorIF iterator() {
    return new ConcreteIterator(this);
  }
  public Collection elements() {
    return Collections.unmodifiableList(list);
  }
}

package javaee.architect.Iterator;
```

```
import java.util.*;
public class ConcreteIterator implements IteratorIF {
  private List list;
  private int index;
  public ConcreteIterator(CollectionIF parm) {
    list = (List) parm.elements();
    index = 0;
  }
  public Object next() throws RuntimeException {
    try {
      return list.get(index++);
    } catch (IndexOutOfBoundsException ioobe) {
      throw new RuntimeException("No Such Element");
    }
  }
  public boolean hasNext() {
    return (index < list.size()) ? true : false;
  }
}

package javaee.architect.Iterator;
public interface IteratorIF {
  // Interface for Iterators.
  public boolean hasNext();
  public Object next();
}
```

Mediator

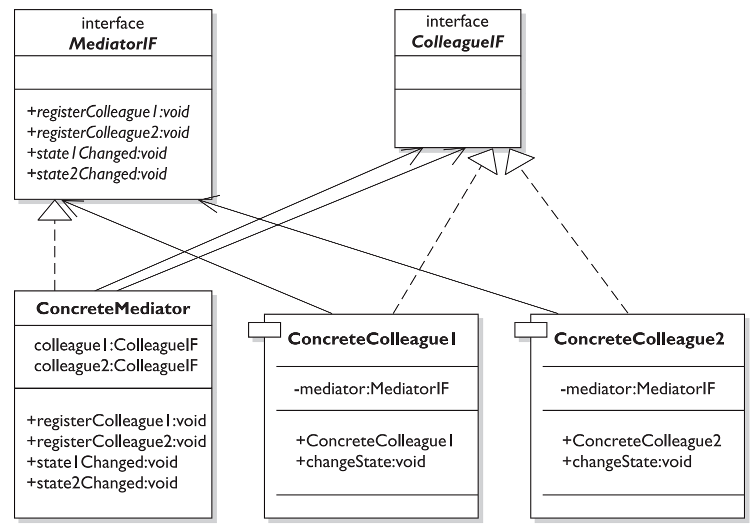
The Mediator pattern's intent is to define an object that encapsulates how a set of objects interacts. It helps to promote a looser coupling by keeping objects from referring to each other explicitly, therefore allowing any interaction to vary independently. The UML is shown in Figure 5-17.

Benefits Following is a list of benefits of using the Mediator pattern:

- It decouples colleagues.
- It simplifies object protocols.
- It centralizes control.
- The individual components become simpler and much easier to deal with because they do not need to pass messages to one another.
- The components do not need to contain logic to deal with their intercommunication and are therefore more generic.

FIGURE 5-17

UML for the Mediator pattern



Applicable Scenarios The following scenarios are most appropriate for the Mediator pattern:

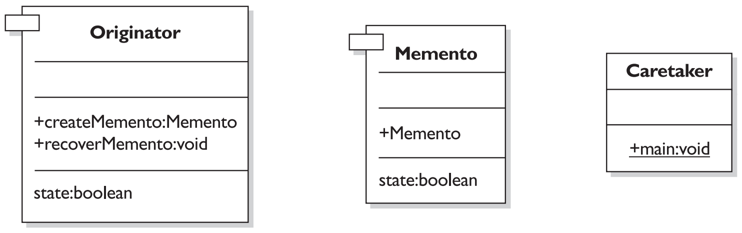
- A set of objects communicates in complex but well-defined ways.
- Custom behavior distributed between several objects is required without subclassing. It is commonly used structurally in message-based systems. The messages themselves are the means by which related objects are decoupled.

Example Code The following example Java code demonstrates the Mediator pattern:

```
package javaee.architect.Mediator;
public class MediatorPattern {
  public static void main(String[] args) {
    System.out.println("Mediator Pattern Demonstration.");
    System.out.println("-------------------------------");
    // Construct mediator and colleagues
    System.out.println("Constructing mediator and colleagues.");
    MediatorIF  mediator   = new ConcreteMediator();
    ColleagueIF colleague1 = new ConcreteColleague1(mediator);
    ColleagueIF colleague2 = new ConcreteColleague2(mediator);
    // Display colleague values.
    System.out.println("Displaying colleague states.");
    System.out.println("colleague1.toString()="+colleague1);
    System.out.println("colleague2.toString()="+colleague2);
    // Change state on colleague1 and the mediator
    // will coordinate the change with colleague2.
    System.out.println("Calling colleague1.changeState()");
    ((ConcreteColleague1) colleague1).changeState();
    // Display colleague values now.
    System.out.println("Displaying colleague states now.");
    System.out.println("colleague1.toString()="+colleague1);
    System.out.println("colleague2.toString()="+colleague2);
    // Change state on colleague2 and see what happens.
    System.out.println("Calling colleague2.changeState()");
    ((ConcreteColleague2) colleague2).changeState();
    // Display colleague values now.
    System.out.println("Displaying colleague states again.");
    System.out.println("colleague1.toString()="+colleague1);
    System.out.println("colleague2.toString()="+colleague2);
    System.out.println();
  }
}

package javaee.architect.Mediator;
public interface ColleagueIF { }

package javaee.architect.Mediator;
public class ConcreteColleague1 implements ColleagueIF {
  private MediatorIF mediator;
  // This colleague uses a boolean for it's state.
  private boolean state;
  public ConcreteColleague1(MediatorIF parm) {
    this.mediator = parm;
    this.mediator.registerColleague1(this);
  }
  public void setState(boolean parm) {
    this.state = parm;
  }
  public void changeState() {
    state = state ? false : true;
    mediator.state1Changed();
  }
  public String toString() {
    return new Boolean(state).toString();
  }
}

package javaee.architect.Mediator;
public class ConcreteColleague2 implements ColleagueIF {
  private MediatorIF mediator;
  // This colleague uses a string for its state.
  private String state = "false";
  public ConcreteColleague2(MediatorIF parm) {
    this.mediator = parm;
    this.mediator.registerColleague2(this);
  }
  public void setState(String parm) {
    this.state = parm;
  }
  public void changeState() {
    state = state.equals("false") ? "true" : "false";
    mediator.state2Changed();
  }
  public String toString() {
    return state;
  }
}

package javaee.architect.Mediator;
public class ConcreteMediator implements MediatorIF {
  ColleagueIF colleague1;
  ColleagueIF colleague2;
  public void registerColleague1(ColleagueIF parm) {
    this.colleague1 = (ConcreteColleague1) parm;
  }
  public void registerColleague2(ColleagueIF parm) {
    this.colleague2 = (ConcreteColleague2) parm;
  }
  public void state1Changed() {
    String s = (colleague2.toString().equals("true")) ? "false" : "true";
    ((ConcreteColleague2) colleague2).setState(s);
  }
  public void state2Changed() {
    boolean b = (colleague1.toString().equals("true")) ? false : true;
    ((ConcreteColleague1) colleague1).setState(b);
  }
}

package javaee.architect.Mediator;
public interface MediatorIF {
  //Interface for communicating with colleagues
  public void registerColleague1(ColleagueIF parm);
  public void registerColleague2(ColleagueIF parm);
  public void state1Changed();
  public void state2Changed();
}
```

FIGURE 5-18

UML for the Memento pattern



Memento

The Memento pattern's intent is to capture and internalize an object's internal state so that objects can be restored to this state later. It must do this without violating encapsulation.

The Memento pattern is also known as *Token*. The UML is shown in Figure 5-18.

Benefits Following is a list of benefits of using the Memento pattern:

- It preserves encapsulation boundaries.
- It simplifies the originator.

Applicable Scenarios The following scenarios are most appropriate for the Memento pattern:

- A snapshot containing enough information regarding the state of an object can be saved so that it can be restored to the complete state using the snapshot information later.
- Using a direct interface to obtain the state would impose implementation details that would break the rules of encapsulation for the object.

Java EE Technology Feature The Java EE technology feature associated with the Memento pattern is EntityBeans using Bean-Managed Persistence (BMP).

Example Code The following example Java code demonstrates the Memento pattern:

```
package javaee.architect.Memento;
public class MementoPattern {
  public static void main(String[] args) {
    System.out.println("Memento Pattern Demonstration.");
    System.out.println("-------------------------------");
    // Run the caretaker
    Caretaker.run();
    System.out.println();
  }
}

package javaee.architect.Memento;
public class Caretaker {
  public static void run() {
    // Create originator and set initial values.
    System.out.println("Creating originator and setting initial values.");
    Originator originator = new Originator();
    originator.setState(true);
    originator.setName("The Originator");
    // Create memento.
    System.out.println("Creating memento.");
    Memento memento = originator.createMemento();
    System.out.println(originator);
    // Change originator values.
    System.out.println("Changing originator values.");
    originator.setState(false);
    originator.setName("To be undone.");
    System.out.println(originator);
    // Recover state from memento.
    System.out.println("Recovering originator values from memento.");
    originator.recoverFromMemento(memento);
    System.out.println(originator);
  }
}

package javaee.architect.Memento;
public class Memento {
  private boolean state;
  private String  name;
  Memento(boolean parm1, String parm2) {
    this.state = parm1;
    this.name  = parm2;
  }
  boolean getState() {return this.state;}
  String getName()   {return this.name;}
}

package javaee.architect.Memento;
public class Originator {
  private boolean state;
  private String  name;
  private String  other;
  // Create memento, save critical data in it.
  public Memento createMemento() {
    return new Memento(state, name);
  }
  // Recover critical data from memento.
  public void recoverFromMemento(Memento memento) {
    this.state = memento.getState();
    this.name  = memento.getName();
  }
  public void setState(boolean parm) {
    this.state = parm;
  }
  public void setName(String parm) {
    this.name = parm;
  }
  public String toString() {
    return "Originator.toString() state="+state+", name="+name;
  }
}
```

Observer

The Observer pattern's intent is to define a one-to-many dependency so that when one object changes state, all its dependents are notified and updated automatically. Java provides support for implementing the Observer pattern via the *java.util.Observer* interface and the *java.util.Observable* class.

The Observer pattern is also known as *Dependents or Publish-Subscribe*. The UML is shown in Figure 5-19.

Benefits Following is a list of benefits of using the Observer pattern:

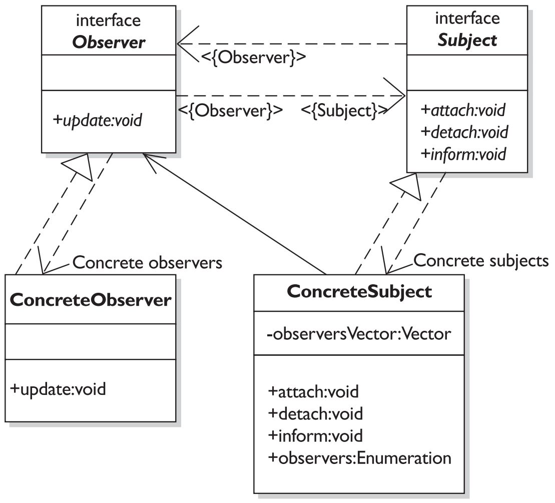
- It abstracts the coupling between the subject and the observer.
- It provides support for broadcast-type communication.

Applicable Scenarios The following scenarios are most appropriate for the Observer pattern:

- A change to an object requires changing other objects, and the number of objects that need to be changed is unknown.
- An object needs to notify other objects without making any assumptions about the identity of those objects.

FIGURE 5-19

UML for the Observer pattern



Java EE Technology Feature and Java SE API Association The Java EE technology feature associated with the Observer pattern is the JMS (Java Message Server) Publish/Subscribe Model.

The Java SE APIs associated with the Observer pattern are

- *java.lang.Observable*
- *java.lang.Observer*

Example Code The following example Java code demonstrates the Observer pattern:

```
package javaee.architect.Observer;
public class ObserverPattern {
  public static void main(String[] args) {
    System.out.println("Observer Pattern Demonstration.");
    System.out.println("--------------------------------");
    // Constructing observers.
    System.out.println("Constructing observer1 and observer2.");
    ObserverIF observer1 = new ConcreteObserver();
    ObserverIF observer2 = new ConcreteObserver();
    // Constructing observable (subject).
    System.out.println("Constructing observerable (subject).");
    ConcreteSubject subject = new ConcreteSubject();
    // Add observer object references to the subject.
    System.out.println("Registering observers with subject.");
    subject.addObserver(observer1);
    subject.addObserver(observer2);
    System.out.println("Doing something in the subject over time...");
    System.out.println();
    System.out.println("            Observable   Observer1    Observer2");
    System.out.println("Iteration    changed?    notified?    notified?");
    // Use loop to simulate time.
    for(int i=0;i < 10;i++) {
      System.out.print(i+":             ");
      subject.doSomething();
      System.out.println();
    }
    System.out.println();
    System.out.println("Removing observer1 from the subject...repeating...");
    System.out.println();
    subject.removeObserver(observer1);
    // Another loop to simulate time.
    for(int i=0;i < 10;i++) {
      System.out.print(i+":             ");
      subject.doSomething();
      System.out.println();
    }
  }
}

package javaee.architect.Observer;
public class ConcreteObserver implements ObserverIF {
  private ConcreteSubject subject; // Reference to subject
  public void update() {
    if (subject == null) { subject = new ConcreteSubject(); }
    System.out.print("         Yes!");
  }
}

package javaee.architect.Observer;
import java.util.*;
public class ConcreteSubject implements SubjectIF {
  List observers = new ArrayList();
  public void addObserver(ObserverIF parm) {observers.add(parm);}
  public void removeObserver(ObserverIF parm)
{observers.remove(observers.indexOf(parm));}
  public void notifyObservers() {
    for (Iterator i = observers.iterator(); i.hasNext();) {
      ((ObserverIF) i.next()).update();
    }
  }
  public void doSomething() {
    double d = Math.random();
    if (d<0.25 || d>0.75) {
      System.out.print("Yes");
      notifyObservers();
    } else {
      System.out.print("No");
    }
  }
}

package javaee.architect.Observer;
public interface ObserverIF {
  public void update();
}

package javaee.architect.Observer;
public interface SubjectIF {
  public void addObserver(ObserverIF parm);
  public void removeObserver(ObserverIF parm);
  public void notifyObservers();
}
```

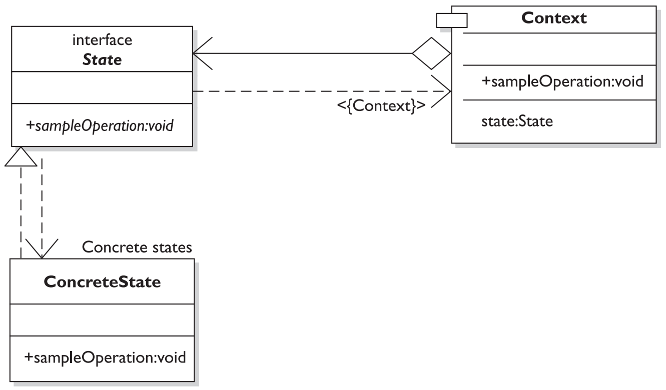
State

The State pattern's intent is to allow an object to alter its behavior when its internal state changes, appearing as though the object itself has changed its class. Another view of the intent of the State pattern is to encapsulate the states of an object as discrete objects, with each object belonging to a separate subclass of an abstract state class.

The State pattern is also known as *Objects for States* and acts in a similar way to the Receiver in the Command pattern. The UML is shown in Figure 5-20.

FIGURE 5-20

UML for the State pattern



Benefits Following is a list of benefits of using the State pattern:

- It keeps state-specific behavior local and partitions behavior for different states.
- It makes any state transitions explicit.

Applicable Scenarios The following scenarios are most appropriate for the State pattern:

- The behavior of an object depends on its state and it must be able to change its behavior at runtime according on the new state.
- Operations have large, multipart conditional statements that depend on the state of the object.

Example Code The following example Java code demonstrates the State pattern:

```
package javaee.architect.State;
public class StatePattern {
  public static void main(String[] args) {
    System.out.println("State Pattern Demonstration.");
    System.out.println("----------------------------");
    // Construct context.
    System.out.println("Constructing context.");
    Context context = new Context();
    // Call request, make state handle the request.
    System.out.println("Calling context.request().");
    context.request();
    // Flip state.
    System.out.println("Calling context.changeState().");
    context.changeState();
    // call request.
    System.out.println("Calling context.request().");
    context.request();
    System.out.println();
  }
}

package javaee.architect.State;
public class ConcreteState1 implements StateIF {
  public void handle() {
    System.out.println("ConcreteState1.handle() called.");
  }
}
```

```
package javaee.architect.State;
public class ConcreteState2 implements StateIF {
  public void handle() {
    System.out.println("ConcreteState2.handle() called.");
  }
}

package javaee.architect.State;
public class Context {
  // Initial state.
  private StateIF state = new ConcreteState1();
  // Request operation.
  public void request() {
    state.handle();
  }
  // Switch states
  public void changeState() {
      if (state instanceof ConcreteState1)
      state = new ConcreteState2();
    else
      state = new ConcreteState1();
  }
}

package javaee.architect.State;
public interface StateIF {
  public void handle();
}
```

Strategy

The Strategy pattern's intent is to define a family of functionality, encapsulate each one, and make them interchangeable. The Strategy pattern lets the functionality vary independently from the clients that use it.

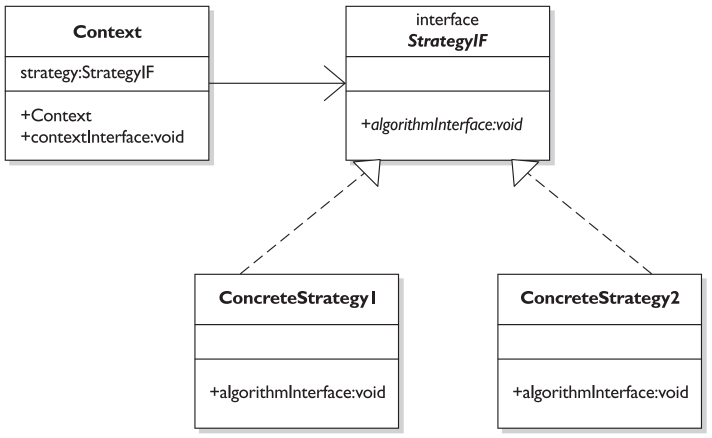
The Strategy pattern is also known as *Policy*. The UML is shown in Figure 5-21.

Benefits Following is a list of benefits of using the Strategy pattern:

- It provides a substitute to subclassing.
- It defines each behavior within its own class, eliminating the need for conditional statements.
- It makes it easier to extend and incorporate new behavior without changing the application.

FIGURE 5-21

UML for the Strategy pattern



Applicable Scenarios The following scenarios are most appropriate for the Strategy pattern:

- Multiple classes differ only in their behaviors. The servlet API is a classic example of this.
- You need different variations of an algorithm.
- An algorithm uses data that is unknown to the client.

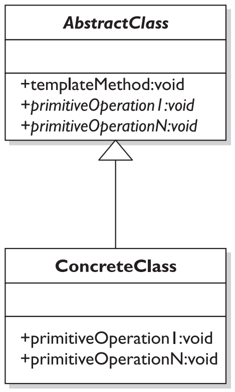
Example Code The following example Java code demonstrates the Strategy pattern:

```
package javaee.architect.Strategy;
public class StrategyPattern {
  public static void main(String[] args) {
    System.out.println("Strategy Pattern Demonstration.");
    System.out.println("-------------------------------");
    // Construct strategies.
    System.out.println("Constructing strategies.");
    StrategyIF strategy1 = new ConcreteStrategy1();
    StrategyIF strategy2 = new ConcreteStrategy2();
    // Construct contexts.
    System.out.println("Constructing contexts.");
    Context context1 = new Context(strategy1);
    Context context2 = new Context(strategy2);
    // Execute contextInterface.
    System.out.println("Constructing context interfaces.");
    context1.contextInterface("J2EE Unleashed");
    context2.contextInterface("J2EE Unleashed");
    context1.contextInterface("The Secret Commissions");
    context2.contextInterface("The Secret Commissions");
    System.out.println();
  }
}

package javaee.architect.Strategy;
public class ConcreteStrategy1 implements StrategyIF {
  // Switch text to all upper case.
  public void algorithmInterface(String parm) {
    System.out.println(parm.toUpperCase());
  }
}

package javaee.architect.Strategy;
public class ConcreteStrategy2 implements StrategyIF {
  // Switch text beginning with "the".
  public void algorithmInterface(String parm) {
    System.out.println((parm.toLowerCase().startsWith("the "))
      ? parm.substring (4)+ ", " + parm.substring(0,4)
      : parm);
  }
}

package javaee.architect.Strategy;
public class Context {
  // Reference to the strategy.
  StrategyIF strategy;
  // Register reference to strategy on construction.
  public Context(StrategyIF parm) {this.strategy = parm;}
  // Call strategy's method.
  public void contextInterface(String parm) {strategy.algorithmInterface(parm);}
}

package javaee.architect.Strategy;
public interface StrategyIF {
  public void algorithmInterface(String parm);
}
```

Template Method

The Template Method pattern's intent is to define the skeleton of a function in an operation, deferring some steps to its subclasses. The Template Method lets subclasses redefine certain steps of a function without changing the structure of the function. The HttpServlet does this in the servlet API. The UML is shown in Figure 5-22.

FIGURE 5-22


AbstractClass
+templateMethod:void
+primitiveOperation1:void
+primitiveOperationN:void
ConcreteClass
+primitiveOperation1:void
+primitiveOperationN:void

UML for the Template Method pattern

Benefit The Template Method pattern is a very common technique for reusing code.

Applicable Scenarios The following scenarios are most appropriate for the Template Method pattern:

- You want to implement the nonvarying parts of an algorithm in a single class and the varying parts of the algorithm in subclasses.
- Common behavior among subclasses should be moved to a single common class, avoiding duplication.

Example Code The following example Java code demonstrates the Template Method pattern:

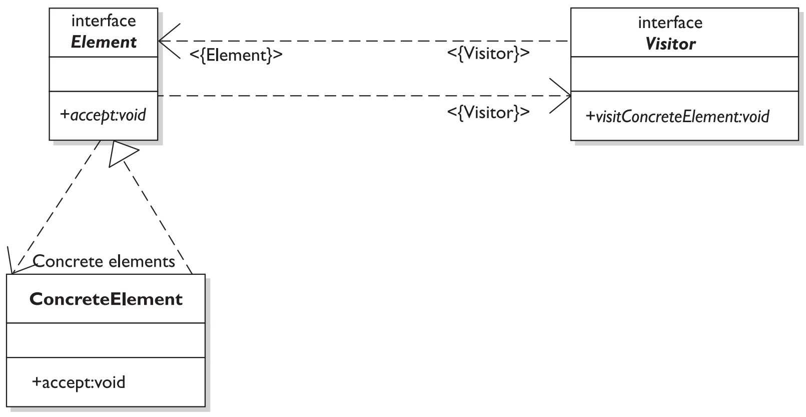
```
package javaee.architect.TemplateMethod;
public class TemplateMethodPattern {
  public static void main(String[] args) {
    System.out.println("TemplateMethod Pattern Demonstration.");
    System.out.println("-------------------------------------");
    // Construct concrete classes.
    System.out.println("Constructing concrete classes.");
    AbstractClass class1 = new ConcreteClass1();
    AbstractClass class2 = new ConcreteClass2();
    // Call template method.
    System.out.println("Calling template methods.");
    class1.templateMethod();
    class2.templateMethod();
    System.out.println();
  }
}

package javaee.architect.TemplateMethod;
public abstract class AbstractClass {
  public void templateMethod() {
    System.out.println("AbstractClass.templateMethod() called.");
    primitiveOperation1();
    primitiveOperationN();
  }
  public abstract void primitiveOperation1();
  public abstract void primitiveOperationN();
}

package javaee.architect.TemplateMethod;
public class ConcreteClass1 extends AbstractClass {
  public void primitiveOperation1() {
    System.out.println("ConcreteClass1.primitiveOperation1() called.");
  }
  public void primitiveOperationN() {
    System.out.println("ConcreteClass1.primitiveOperationN() called.");
  }
}

package javaee.architect.TemplateMethod;
public class ConcreteClass2 extends AbstractClass {
  public void primitiveOperation1() {
    System.out.println("ConcreteClass2.primitiveOperation1() called.");
  }
  public void primitiveOperationN() {
    System.out.println("ConcreteClass2.primitiveOperationN() called.");
  }
}
```

Visitor

The Visitor pattern's intent is to represent an operation to be performed on elements of an object structure. The Visitor pattern allows for the addition of a new operation without changing the classes of the elements on which it is to operate. Figure 5-23 shows the UML.

FIGURE 5-23

UML for the Visitor pattern



Benefits Following are the benefits of using the Visitor pattern:

- It simplifies the addition of new operations.
- It gathers related operations while separating unrelated ones.

Applicable Scenarios The following scenarios are most appropriate for the Visitor pattern:

- An object structure contains many objects with differing interfaces and there is a need to perform operations on these objects in a way that depends on their concrete classes.
- Many distinct and unrelated operations need to be performed on objects in a structure and there is a need to avoid cluttering the classes with these operations.
- The classes defining the object structure rarely change but you frequently need to define new operations that perform over the structure.

Example Code The following example Java code demonstrates the Visitor pattern:

```
package javaee.architect.Visitor;
public class VisitorPattern {
  public static void main(String[] args) {
    System.out.println("Visitor Pattern Demonstration.");
    System.out.println("------------------------------");
    // Construct list of elements.
    System.out.println("Constructing two elements.");
    ElementIF[] elements = new ElementIF[2];
    elements[0] = new ConcreteElementA();
    elements[1] = new ConcreteElementB();
    // Construct object structure.
    System.out.println("Constructing object structure.");
    ObjectStructure objectStructure = new ObjectStructure(elements);
    // Visit elements in object structure.
    System.out.println("Visiting elements in object structure.");
    objectStructure.visitElements();
    System.out.println();
  }
}

package javaee.architect.Visitor;
public class ConcreteElementA implements ElementIF {
  public void accept(VisitorIF parm) {
    parm.visitConcreteElementA(this);
  }
  public void operationA() {
    System.out.println("ConcreteElementA.operationA() called.");
  }
}

package javaee.architect.Visitor;
public class ConcreteElementB implements ElementIF {
  public void accept(VisitorIF parm) {
    parm.visitConcreteElementB(this);
  }
  public void operationB() {
    System.out.println("ConcreteElementB.operationB() called.");
  }
}

package javaee.architect.Visitor;
public class ConcreteVisitor implements VisitorIF {
  public void visitConcreteElementA(ConcreteElementA parm) {
    parm.operationA();
  }
  public void visitConcreteElementB(ConcreteElementB parm) {
    parm.operationB();
  }
}

package javaee.architect.Visitor;
public interface ElementIF {
  public void accept(VisitorIF parm);
}
```

```
package javaee.architect.Visitor;
import java.util.*;
public class ObjectStructure {
  private List objectStruct;
  private VisitorIF visitor;
  public ObjectStructure(ElementIF[] parm) {
    objectStruct = Arrays.asList(parm);
  }
  public void visitElements() {
    if (visitor == null) { visitor = new ConcreteVisitor(); }
    for (Iterator i = objectStruct.iterator(); i.hasNext();) {
      ((ElementIF) i.next()).accept(visitor);
    }
  }
}

package javaee.architect.Visitor;
public interface VisitorIF {
  public void visitConcreteElementA(ConcreteElementA parm);
  public void visitConcreteElementB(ConcreteElementB parm);
}
```

Now that we've covered each of the Gang of Four's (GoF) Design Patterns, let's review scenarios and also identify the Design Pattern that is most appropriate as a solution.

Sun's J2EE Patterns

Part I of the certification exam requires that you know the GoF Design Patterns only, but for Parts II and III, you may find it helpful to study and then include in your solution some of the J2EE Patterns from Sun. Although we do not go into great detail on these patterns, the next few sections will at least serve as an introduction by covering the scenarios for which they are potential solutions.

exam watch

You may find it helpful to study and then include in your solutions some of the J2EE Patterns from Sun. For complete details on the J2EE Patterns, refer to the following web sites: http://java.sun.com/reference/blueprints and http://java.sun.com/blueprints/patterns/index.html.

SCENARIO & SOLUTION

Given Scenario	Appropriate Design Pattern
The system needs to be independent of how its objects are created, composed, and represented. The system needs to be configured with one of a multiple family of objects. The family of related objects is intended to be used together and this constraint needs to be enforced. You want to provide a library of objects that does not show implementations but only reveals interfaces.	Abstract Factory
The algorithm for creating a complex object needs to be independent of the components that compose the object and how they are assembled. The construction process is to allow different representations of the constructed object.	Builder
A class is not able to anticipate the class of objects it needs to create. A class wants its subclasses to specify the objects it instantiates. Classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.	Factory Method
The classes to instantiate are specified at runtime. You want to avoid building a class hierarchy of factories that parallels the hierarchy of objects. Instances of the class have one of only a few different combinations of state.	Prototype
A single instance of a class is needed, and it must be accessible to clients from a well-known access point.	Singleton
You want to utilize an existing class with an incompatible interface. You want to create a reusable class that cooperates with classes that don't necessarily have compatible interfaces. You need to use several existing subclasses but do not want to adapt their interfaces by subclassing each one.	Adapter
You want to avoid a permanent binding between the functional abstraction and its implementation. Both the functional abstraction and its implementation need to be extended using subclasses. Changes to the implementation should not impact the client (not even a recompile).	Bridge

SCENARIO & SOLUTION

Given Scenario	Appropriate Design Pattern
You want to represent a full or partial hierarchy of objects. You want clients to be able to ignore the differences between the varying objects in the hierarchy. The structure is dynamic and can have any level of complexity.	Composite
You want to transparently and dynamically add responsibilities to objects without affecting other objects. You want to add responsibilities to an object that you may want to change in the future. Extending functionality by subclassing is no longer practical.	Decorator
You want to provide a simpler interface to a more complex subsystem. Several dependencies exist between clients and the implementation classes of an abstraction. You want to layer the subsystems.	Facade
The application uses a considerable number of objects. The storage costs are high because of the quantity of objects. The application does not depend on object identity.	Flyweight
You need a more versatile or sophisticated reference to an object, rather than a simple pointer.	Proxy
More than one object can handle a request and the handler is unknown. A request is to be issued to one of several objects and the receiver is not specified explicitly. The set of objects able to handle the request is to be specified dynamically.	Chain of Responsibility
You need to parameterize objects by an action to perform. You specify, queue, and execute requests at different times. You need to support rollback, logging, or transaction functionality.	Command
The grammar of the language is not complicated and efficiency is not a priority.	Interpreter
Access to a collection object is required without having to expose its internal representation. You need to support multiple traversals of objects in the collection. You need to provide a universal interface for traversing different structures in the collection.	Iterator

SCENARIO & SOLUTION

Given Scenario	Appropriate Design Pattern
A set of objects communicates in complex but well-defined ways. Custom behavior distributed between several objects is required without subclassing.	Mediator
A snapshot containing enough information regarding the state of an object can be saved so that it can be restored to the complete state using the snapshot information later. Using a direct interface to obtain the state would impose implementation details that would break the rules of encapsulation for the object.	Memento
A change to an object requires changing other objects, and the number of objects that need to be changed is unknown. An object needs to notify other objects without making any assumptions about the identity of those objects.	Observer
The behavior of an object depends on its state and it must be able to change its behavior at runtime according on the new state. Operations have large multipart conditional statements that depend on the state of the object.	State
Multiple classes differ only in their behavior. You need different variations of an algorithm. An algorithm uses data that is unknown to the client.	Strategy
You want to implement the nonvarying parts of an algorithm in a single class and the varying parts of the algorithm in subclasses. Common behavior among subclasses should be moved to a single common class, avoiding duplication.	Template Method
An object structure contains many objects with differing interfaces and you need to perform operations on these objects in a way that depends on their concrete classes. Many distinct and unrelated operations need to be performed on objects in a structure and you need to avoid cluttering the classes with these operations. The classes defining the object structure rarely change but you frequently need to define new operations that perform over the structure.	Visitor

Similar to the GoF Design Patterns, the J2EE Patterns are broken down into the various sections that address the tiers (or layers) that make up an application:

- Presentation Tier
- Business Tier
- Integration Tier

Presentation Tier J2EE Patterns

The presentation tier encapsulates the logic required to service the clients accessing a system. Presentation tier patterns intercept a client request and then provide facilities such as single sign-on, management of the client session, and access to services in the business tier before constructing and delivering the response back to the client.

The J2EE patterns available for the presentation layer follow:

- Composite View
- Dispatcher View
- Front Controller
- Intercepting Filter
- Service To Worker
- View Helper

The next table lists scenarios along with suggestions of one or more of the presentation tier J2EE patterns to aid in the solution.

Business Tier Patterns

The business tier provides the services required by application clients and contains the business data and logic. All business processing for the application is gathered and placed into this tier. Enterprise JavaBean (EJB) components are one of the ways to implement business processing in this tier.

Here are the J2EE patterns available for the business tier:

- Business Delegate
- Composite Entity (formally Aggregate Entity)
- Service Locator
- Session Facade
- Transfer Object (formally Value Object)

SCENARIO & SOLUTION

Given Scenario	Appropriate Presentation Tier Pattern
You have an application that needs to preprocess and/or post-process a client request…	Intercepting Filter
You have an application that requires centralized control for client request handling…	Front Controller and Intercepting Filter
You need to add logging, debugging, or some other behavior to be carried out for each client request…	Front Controller and Intercepting Filter
You want to create a generic command interface for delegating processing from the controller to the helper components…	Front Controller
You want to delegate processing to a JSP or servlet and you want to implement your Model View Controller (MVC) Controller as a JSP or servlet…	Front Controller
You want to create an MVC View from multiple subviews…	Composite View
You need to implement an MVC View as a JSP or servlet…	View Helper
You would like to partition your MVC Model and MVC View…	View Helper
Your application needs to encapsulate presentation-related data formatting logic…	View Helper
You want to implement your Helper components as Custom tags or JavaBeans…	View Helper
Your application needs to combine multiple presentation patterns…	Service To Worker and Dispatcher View
You want to encapsulate MVC View management and navigation logic…	Service To Worker and Dispatcher View

- Transfer Object Assembler (formally Value Object Assembler)
- Value List Handler

The following table is a list of scenarios along with suggestions of one or more of the business tier J2EE patterns to aid in the solution.

Integration Tier J2EE Patterns

This tier is responsible for accessing external resources and systems, such as relational and nonrelational data stores and any legacy applications. A business tier object uses

SCENARIO & SOLUTION

Given Scenario	Appropriate Business Tier Pattern
You need to minimize coupling between presentation and business layers…	Business Delegate
You need to cache business services for clients…	Business Delegate
Your application needs a simpler interface to clients…	Business Delegate
Within the business tier you want to shield the client from implementation (lookup/creation/access) details of business services…	Business Delegate and Service Locator
Your application needs to separate the lookup for vendor or other technology dependencies for services…	Service Locator
You need to provide a uniform method for service lookup and creation…	Service Locator
You want to shield the complexity and dependencies for EJB and JMS component lookup…	Service Locator
You need to transfer data between application tiers…	Transfer Object
You have to reduce network traffic between clients and EJBs…	Session Facade
You want to minimize the number of remote method invocations by providing coarser-grained method access to business tier components…	Session Facade
You want to manage relationships between EJB components and hide the complexity of their interactions…	Session Facade
You need to shield components in the business tier from clients…	Session Facade and Business Delegate
You want to provide uniform access to components in the business tier…	Session Facade
You need to design complex, coarser-grained EJB entity beans…	Composite Entity
You have to identify coarse-grained objects and dependent objects for EJB entity bean design…	Composite Entity
You want to minimize or eliminate the EJB entity bean clients' dependency on the actual database schema…	Composite Entity
You have to improve manageability and minimize number of EJB entity beans…	Composite Entity

SCENARIO & SOLUTION

Given Scenario	Appropriate Business Tier Pattern
You want to minimize (or eliminate) EJB entity bean to entity bean relationships…	Composite Entity and Session Facade
You need to get the data model for the application from various business tier components…	Transfer Object Assembler (This could also be a DataAccessObject as well)
You want on-the-fly data model construction…	Transfer Object Assembler
You want to shield the data model construction complexity from clients…	Transfer Object Assembler
Your application needs to provide query and list processing facilities…	Value List Handler
You want to reduce the overhead of using EJB finder methods…	Value List Handler
You need to facilitate server-side caching of query results, with forward and backward navigation, for clients …	Value List Handler

the integration tier when it requires data or services that reside at the resource level. The components in this tier can use JDBC, Java EE connector technology, or some other proprietary software to access data at the resource level.

Here are the J2EE patterns available for the integration tier:

- Data Access Object
- Service Activator

The following table is a list of scenarios along with suggestions of one or more of the integration tier J2EE patterns to aid in the solution.

SCENARIO & SOLUTION

You want to reduce the amount of coupling between business and resource tiers (layers)…	Data Access Object
You need to centralize the access to resource tiers (layers) …	Data Access Object
You must reduce complexity for accessing resource from the business tier (layer) …	Data Access Object
You want to provide asynchronous processing for EJB components…	Service Activator
You need to send a message to an EJB…	Service Activator

exam watch

The level of detail that we've provided on Sun's core J2EE patterns is somewhat cursory. Although you will not need to have knowledge of them for Part I, you will find it useful to learn more about them and then use them in Parts II and III.

CERTIFICATION OBJECTIVE 5.03

State the Name of a Gamma et al. Design Pattern Given the UML Diagram and/or a Brief Description of the Pattern's Functionality

Study each design pattern diagram shown earlier. The following table has a brief description of each pattern's functionality:

Pattern's Functionality	Pattern Name
Provides an interface for creating families of related or dependent objects without specifying the concrete classes.	Abstract Factory
Separates construction of a complex object from its representation so that the construction process can create different representations.	Builder
Defines an interface for creating an object, letting subclasses decide which class to instantiate. Allows a class to defer the actual instantiation to subclasses.	Factory Method
Specifies the kinds of objects to create using a prototypical instance, and creates new objects by copying this prototype.	Prototype
Ensures a class has only one instance, and provides a global point of access to it.	Singleton
Converts the class's interface into another interface that the client expects. Lets classes work together that couldn't otherwise do so because of incompatible interfaces.	Adapter
Decouples abstraction from its implementation so that the two can vary independently.	Bridge

Pattern's Functionality	Pattern Name
Composes objects into tree structures to represent part-whole hierarchies. Lets clients treat individual objects and compositions of objects in a uniform manner.	Composite
Attaches added responsibilities to an object dynamically. Provides flexible alternative to subclassing to extend functionality.	Decorator
Provides a unified interface to a set of interfaces in one or more subsystems. Defines a higher-level interface that makes the subsystems easier to use.	Facade
Uses sharing to support large numbers of fine-grained objects in an efficient manner.	Flyweight
Provides a placeholder or surrogate for another object to control access to it.	Proxy
Avoids coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. The receiving objects are chained together and pass the request along the chain until it is handled.	Chain Of Responsibility
Encapsulates a request as an object, allowing the client to be parameterized with different requests, to queue or log requests, and to be able to support undo operations.	Command
Given a language, defines a representation for its grammar along with an interpreter of the grammar that uses the representation to interpret sentences in the language.	Interpreter
Provides a way to access the elements of a collection (aggregate) object sequentially without having to expose the underlying representation.	Iterator
Defines an object that encapsulates how a set of objects interacts. Promotes loose coupling by keeping objects from referring to each other directly and varying their interaction independently.	Mediator
Without violating encapsulation, captures and externalizes an object's internal state so that the object's essential state can be restored later.	Memento
Defines a one-to-many dependency among objects so that when one object changes state, all its dependents (subscribers) are notified and updated automatically.	Observer
Allows an object to alter its behavior when its internal state changes; the object will appear to change its class.	State
Defines a family of algorithms, encapsulating each one, and makes them interchangeable. Lets the algorithm vary independently from clients that use it.	Strategy
Defines the skeleton of an algorithm (function) in an operation, deferring some steps to subclasses. Lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.	Template Method
Represents an operation to be performed on the elements of an object structure. Lets you define a new operation without changing the classes of the elements on which it operates.	Visitor

The following table shows the alternate names for the Gamma et al. Design Patterns:

Pattern Name	Also Known As
Abstract Factory	Kit
Factory Method	Virtual Constructor
Adapter	Wrapper
Bridge	Handle/Body
Decorator	Wrapper
Proxy	Surrogate
Command	Action or Transaction
Iterator	Cursor
Memento	Token
Observer	Dependents or Publish-Subscribe
State	Objects for States
Strategy	Policy

CERTIFICATION OBJECTIVE 5.04

Identify Benefits of a Specified Gamma et al. Design Pattern

Here is a list of the benefits for each of the Gamma et al. (GoF) Design Patterns:

GoF Design Pattern	Benefits
Abstract Factory	Isolates client from concrete (implementation) classes. Makes the exchanging of object families easier. Promotes consistency among objects.
Builder	Permits you to vary an object's internal representation. Isolates the code for construction and representation. Provides finer control over the construction process.

GoF Design Pattern	Benefits
Factory Method	Removes the need to bind application-specific classes into the code. The code interacts solely with the resultant interface and so will work with any classes that implement that interface. Because creating objects inside a class is more flexible than creating an object directly, it enables the subclass to provide an extended version of an object.
Prototype	Allows adding or removing objects at runtime. Specifies new objects by varying its values or structure. Reduces the need for subclassing. Allows dynamic configuring of an application with classes.
Singleton	Controls access to a single instance of the class. Reduces name space usage. Permits refinement of operations and representation. Permits a variable number of instances. Is more flexible than class methods (operations).
Adapter	Allows two or more previously incompatible objects to interact. Allows reusability of existing functionality.
Bridge	Enables the separation of implementation from the interface. Improves extensibility. Allows the hiding of implementation details from the client.
Composite	Defines class hierarchies consisting of primitive and complex objects. Makes it easier to add new kinds of components. Provides the flexibility of structure with a manageable interface.
Decorator	Provides greater flexibility than static inheritance. Avoids the need to place feature-laden classes higher-up the hierarchy. Simplifies coding by allowing you to develop a series of functionality-targeted classes, instead of coding all of the behavior into the object. Enhances the extensibility of the object, because changes are made by coding new classes.
Facade	Provides a simpler interface to a complex subsystem without reducing the options provided by the subsystem. Shields clients from the complexity of the subsystem components. Promotes looser coupling between the subsystem and its clients. Reduces the coupling between subsystems provided that every subsystem uses its own Facade pattern and other parts of the system use the Facade pattern to communicate with the subsystem.
Flyweight	Reduces the number of objects to deal with. Reduces memory and storage devices if the objects are persisted.
Proxy	Remote proxy shields the fact that the implementation resides in another address space. Virtual proxy performs optimizations—e.g., by creating objects on demand.

GoF Design Pattern	Benefits
Chain of Responsibility	Reduces coupling. Adds flexibility when assigning responsibilities to objects. Allows a set of classes to act as one; events produced in one class can be sent to other handler classes within the composition.
Command	Separates the object that invokes the operation from the object that performs the operation. Simplifies adding new commands, because existing classes remain unchanged.
Interpreter	Makes it easier to change and extend the grammar. Makes implementing the grammar straightforward.
Iterator	Supports variations in the traversal of a collection. Simplifies the interface to the collection.
Mediator	Decouples colleagues. Simplifies object protocols. Centralizes control. Individual components become simpler and much easier to deal with because they do not need to pass messages to one another. Components do not need to contain logic to deal with their intercommunication and are therefore more generic.
Memento	Preserves encapsulation boundaries. Simplifies the originator.
Observer	Abstracts the coupling between the subject and the observer. Provides support for broadcast-type communication.
State	Keeps state-specific behavior local and partitions behavior for different states. Makes any state transitions explicit.
Strategy	Provides a substitute to subclassing. Defines each behavior within its own class, eliminating the need for conditional statements. Makes it easier to extend and incorporate new behavior without changing the application.
Template Method	Lets code be reused.

CERTIFICATION OBJECTIVE 5.05

Identify the Gamma et al. Design Pattern Associated with a Specified Java EE Technology Feature

Here is a list of Java EE technology features and the associated Gamma et al. design patterns that are used to implement them:

Java EE Technology Feature	Associated GoF Design Pattern
EJB Factory (*javax.ejb.EJBHome*, *javax.ejb.EJBLocalHome*) JMS Connection Factory (*javax.jms.QueueConnectionFactory*, *javax.jms.TopicConnectionFactory*)	Factory Method
EJB remote reference (*javax.ejb.EJBObject*)	Proxy
JMS Publish/Subscribe Model	Observer

CERTIFICATION SUMMARY

By studying this chapter, you now have an understanding of the GoF design patterns and some introductory material on J2EE patterns. You should also understand which are the most appropriate patterns to use for given scenarios.

✓ TWO-MINUTE DRILL

Here are some of the key points from each certification objective in Chapter 5.

Identify the Benefits of Using Design Patterns

- ❑ Help designers to focus on solutions quicker if they recognize patterns that have been successful in the past.
- ❑ Give new ideas to designers who have studied patterns.
- ❑ Provide a common language for design discussions.
- ❑ Provide a solution to a real-world problem.
- ❑ Capture knowledge and document the best practices for a domain.
- ❑ Document decisions and the rationale that lead to the solution.
- ❑ Reuse the experience of predecessors.
- ❑ Communicate the insight already gained previously.
- ❑ Describe the circumstances (when and where), the influences (who and what), and the resolution (how and why it balances the influences).

Identify the Most Appropriate Design Pattern for a Given Scenario

- ❑ The Abstract Factory is most appropriate when the system needs to be independent of how its objects are created, composed, and represented.
- ❑ The Adapter is most appropriate when you want to utilize an existing class with an incompatible interface.
- ❑ The Bridge is most appropriate when you want to avoid a permanent binding between the functional abstraction and its implementation.
- ❑ The Builder is most appropriate when the algorithm for creating a complex object needs to independent of the components that compose the object and how they are assembled.
- ❑ The Chain of Responsibility is most appropriate when more than one object can handle a request and the handler is unknown.
- ❑ The Command is most appropriate when you need to parameterize objects by an action to perform.

- ❑ The Composite is most appropriate when you want to represent a full or partial hierarchy of objects.
- ❑ The Decorator is most appropriate when you want to transparently and dynamically add responsibilities to objects without affecting other objects.
- ❑ The Facade is most appropriate when you want to provide a simpler interface to a more complex subsystem.
- ❑ The Factory Method is most appropriate when a class is not able to anticipate the class of objects it needs to create.
- ❑ The Flyweight is most appropriate when the application uses a considerable number of objects.
- ❑ The Interpreter is most appropriate when the grammar of the language is not complicated and efficiency is not a priority.
- ❑ The Iterator is most appropriate when access to a collection object is required without having to expose its internal representation.
- ❑ The Mediator is most appropriate when a set of objects communicates in complex but well-defined ways.
- ❑ The Memento is most appropriate when a snapshot containing enough information regarding the state of an object can be saved so that it can be restored to the complete state using the snapshot information later.
- ❑ The Observer is most appropriate when a change to an object requires changing other objects, and the number of objects that need to be changed is unknown.
- ❑ The Prototype is most appropriate when the classes to instantiate are to be specified at runtime.
- ❑ The Proxy is most appropriate when you need a more versatile or sophisticated reference to an object, rather than a simple pointer.
- ❑ The Singleton is most appropriate when a single instance of a class is needed, and it must be accessible to clients from a well-known access point.
- ❑ The State is most appropriate when the behavior of an object depends on its state and it must be able to change its behavior at runtime according to the new state.
- ❑ The Strategy is most appropriate when multiple classes differ only in their behavior.

- ❑ The Template Method is most appropriate when you want to implement the nonvarying parts of an algorithm in a single class and the varying parts of the algorithm in subclasses.
- ❑ The Visitor is most appropriate when an object structure contains many objects with differing interfaces and you need to perform operations on these objects in a way that depends on their concrete classes.

State the Name of a Gamma et al. Design Pattern Given the UML Diagram and/or a Brief Description of the Pattern's Functionality

Review the GoF (Gamma et al.) diagrams and associated descriptions that appear earlier in the chapter:

Identify Benefits of a Specified Gamma et al. Design Pattern

Here are the benefits for each of the Gamma et al. design patterns:

Identify the Gamma et al. Design Pattern Associated with a Specified Java EE Technology Feature

Here is a list of Java EE technology features and the associated design patterns that are used to implement them:

- ❑ The EJB Factory (javax.ejb.EJBHome, javax.ejb.EJBLocalHome) and JMS Connection Factory (javax.jms.QueueConnectionFactory, javax.jms.TopicConnectionFactory) use the Factory Method pattern.
- ❑ The EJB remote reference (javax.ejb.EJBObject) uses the Proxy pattern.
- ❑ The JMS Publish/Subscribe Model uses the Observer pattern.

SELF TEST

The following questions will help you measure your understanding of the material presented in this chapter. Read all the choices carefully because there may be more than one correct answer. Choose all correct answers for each question.

Identify the Benefits of Using Design Patterns

1. Which of the following is not a benefit of using Design Patterns?

A. They provide a common language for design discussions.

B. They provide solutions to “real-world” problems.

C. They communicate the insight already gained previously.

D. They provide solutions to totally novel problems.

Identify the Most Appropriate Design Pattern for a Given Scenario

2. The Factory Method design pattern is useful when a client must create objects having different

A. Subclasses

B. Ancestors

C. Sizes

D. Similarities

3. What design pattern limits the number of instances a class can create?

A. Command

B. Limiter

C. Strategy

D. Singleton

4. Iterators are useful when dealing with which of the following types of classes?

A. Dynamic

B. Collection

C. Singleton

D. Small

State the Name of a Gamma et al. Design Pattern Given the UML Diagram and/or a Brief Description of the Pattern's Functionality

5. What is the Abstract Factory pattern also known as?

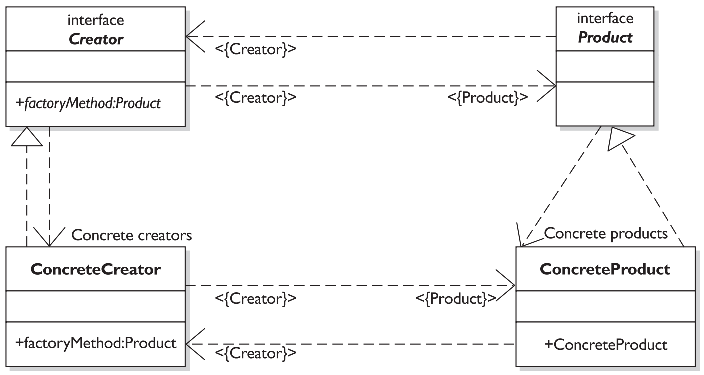
A. Kit

B. Wrapper

C. Cursor

D. Virtual Constructor

6. Which pattern is shown in the diagram?



A. Abstract Factory

B. Factory Method

C. Command

D. Chain of Responsibility

7. What pattern is also known as Virtual Constructor?

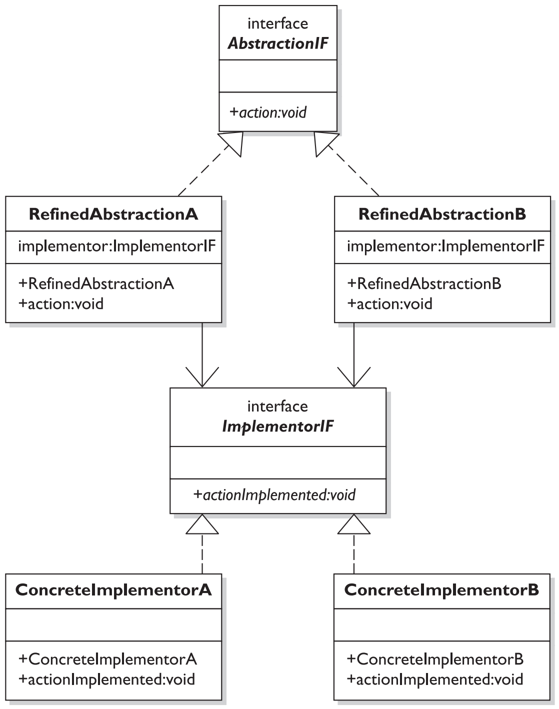
A. Abstract Factory

B. Memento

C. Wrapper

D. Factory Method

8. Which pattern is shown in the diagram?

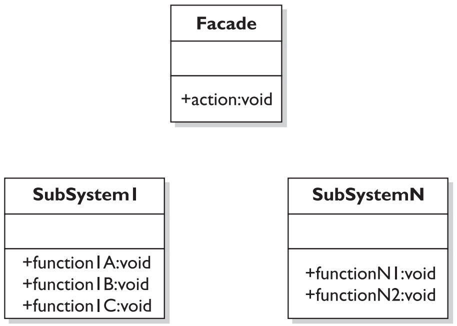


A. Proxy
B. Decorator
C. Bridge
D. Observer

9. What is the Adapter pattern also known as?

A. Surrogate
B. Wrapper
C. Token
D. Proxy

10. Which pattern is shown in the diagram?

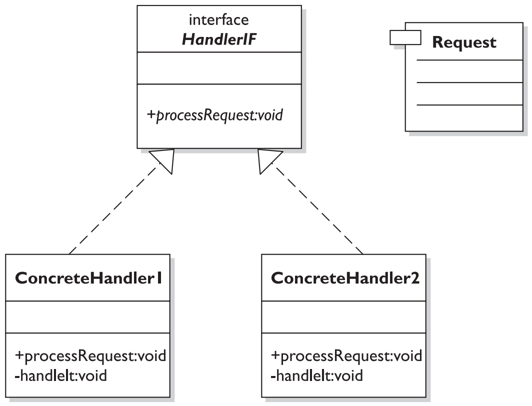


A. Proxy
B. Facade
C. Adapter
D. Bridge

11. What pattern is also known as Handle/Body?

A. Proxy
B. Adapter
C. Abstract Factory
D. Bridge

12. Which pattern is shown in the diagram?

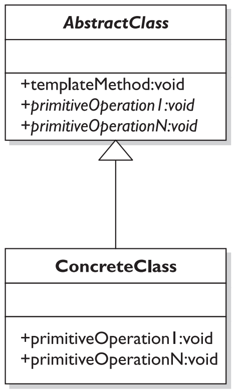


A. Chain of Responsibility
B. Command
C. Memento
D. Factory Method

13. What is the Decorator pattern also known as?

A. Wrapper
B. Adapter
C. Composite
D. Strategy

14. Which pattern is shown in the diagram?


AbstractClass
+templateMethod:void
+primitiveOperation1:void
+primitiveOperationN:void
ConcreteClass
+primitiveOperation1:void
+primitiveOperationN:void

A. Template Method
B. Command
C. Singleton
D. State

15. What pattern is also known as Surrogate?

A. Observer
B. Bridge
C. Proxy
D. Decorator

16. What is the Command pattern also known as?

A. Action
B. Transaction

C. Wrapper
D. Surrogate

17. The Command design pattern ____________ a request in an object.

A. Separates
B. Encapsulates
C. Processes
D. Decouples

Identify Benefits of a Specified Gamma et al. Design Pattern

18. Which of the following elements are parts of the Gang of Four (GoF) Design Pattern format?

A. Problem
B. Solution
C. Consequences
D. Intent

Identify the Gamma et al. Design Pattern Associated with a Specified Java EE Technology Feature

19. The Decorator pattern appears in which of the following Java packages?

A. *java.io*
B. *java.awt*
C. *java.lang*
D. *java.util*

20. Which Java package contains classes that implement the Iterator design pattern?

A. *java.enumeration*
B. *java.util*
C. *java.math*
D. *java.text*

21. What two methods are defined by the Enumeration interface?

A. `hasMoreElements()`
B. `getElement()`
C. `nextElement()`
D. `nextelement()`

SELF TEST ANSWERS

Identify the Benefits of Using Design Patterns

1. ☑ **D** is correct. Design patterns do not address totally novel problems, so this cannot be a benefit gained.
☒ **A, B,** and **C** are incorrect. These are benefits gained by using Design Patterns.

Identify the Most Appropriate Design Pattern for a Given Scenario

2. ☑ **A** is correct. The Factory Method design pattern is useful when a client must create objects having different subclasses.
☒ **B, C,** and **D** are incorrect. The Factory Method design pattern is not useful with these situations.

3. ☑ **D** is correct. The Singleton pattern limits the number of instances a class can create.
☒ **A, B,** and **C** are incorrect. These do not limit the number of instances a class can create.

4. ☑ **B** is correct. Iterators are useful when dealing with Collection classes.
☒ **A, C,** and **D** are incorrect. These are not appropriate for the Iterator pattern.

State the Name of a Gamma et al. Design Pattern Given the UML Diagram and/or a Brief Description of the Pattern's Functionality

5. ☑ **A** is correct. The Abstract Factory pattern is also known as Kit.
☒ **B, C,** and **D** are incorrect. These are not valid aliases for Abstract Factory.

6. ☑ **B** is correct. The diagram depicts the Factory Method pattern.
☒ **A, C,** and **D** are incorrect. These are not depicted in the diagram.

7. ☑ **D** is correct. The Factory Method pattern is also known as the Virtual Constructor.
☒ **A, B,** and **C** are incorrect. These are not valid aliases for Virtual Constructor.

8. ☑ **C** is correct. The diagram depicts the Bridge pattern.
☒ **A, B,** and **D** are incorrect. These are not depicted in the diagram.

9. ☑ **B** is correct. The Adapter pattern is also known as the Wrapper.
☒ **A, C,** and **D** are incorrect. These are not valid aliases for Adapter.

10. ☑ **B** is correct. The diagram depicts the Facade pattern.
☒ **A, C,** and **D** are incorrect. These are not depicted in the diagram.

11. ☑ **D** is correct. The Bridge pattern is also known as Handle/Body.
☒ **A, B,** and **C** are incorrect. These are not valid aliases for Handle/Body.

12. ☑ **A** is correct. The diagram depicts the Chain of Responsibility pattern.
☒ **B, C,** and **D** are incorrect. These are not depicted in the diagram.

13. ☑ **A** is correct. The Decorator pattern is also known as the Wrapper.
☒ **B, C,** and **D** are incorrect. These are not valid aliases for Decorator.

14. ☑ **A** is correct. The diagram depicts the Template Method pattern.
☒ **B, C,** and **D** are incorrect. These are not depicted in the diagram.

15. ☑ **C** is correct. The proxy pattern is also known as Surrogate.
☒ **A, B,** and **D** are incorrect. These are not valid aliases for Surrogate.

16. ☑ **A** and **B** are correct. The Command pattern is also known as Action or Transaction.
☒ **C** and **D** are incorrect. These are not valid aliases for Command.

17. ☑ **B** is correct. The Command design pattern encapsulates a request in an object.
☒ **A, C,** and **D** are incorrect. These are not valid descriptions of the Command pattern.

Identify Benefits of a Specified Gamma et al. Design Pattern

18. ☑ **C** and **D** are correct. Consequences and Intent are valid elements in the (GoF) Design Pattern format.
☒ **A** and **B** are incorrect. These are not valid elements in the (GoF) Design Pattern format.

Identify the Gamma et al. Design Pattern Associated with a Specified Java EE Technology Feature

19. ☑ **A** and **B** are correct. The Decorator pattern appears in the *java.io* and *java.awt* packages.
☒ **C** and **D** are incorrect. These do not contain the Decorator pattern.

20. ☑ **B** is correct. The *java.util* package contains classes that implement the Iterator design pattern.
☒ **A, C,** and **D** are incorrect. These do not implement the Iterator design pattern.

21. ☑ **A** and **C** are correct. The Enumeration interface contains `hasMoreElements()` and `nextElement()` methods.
☒ **B** and **D** are incorrect. These are not valid methods in the Enumeration interface.