Design Patterns in Java
The patterns discussed in this section are some of the most common, basic and important design patterns one can find in the areas of object-oriented design and programming. Some of these fundamental design patterns, such as the Interface, Abstract Parent, Private Methods, etc., are used extensively during the discussion of the other patterns in this book.

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Pattern Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Interface</td>
<td>Can be used to design a set of service provider classes that offer the same service so that a client object can use different classes of service provider objects in a seamless manner without having to alter the client implementation.</td>
</tr>
<tr>
<td>4</td>
<td>Abstract Parent Class</td>
<td>Useful for designing a framework for the consistent implementation of the functionality common to a set of related classes.</td>
</tr>
<tr>
<td>5</td>
<td>Private Methods</td>
<td>Provide a way of designing a class behavior so that external objects are not permitted to access the behavior that is meant only for the internal use.</td>
</tr>
<tr>
<td>6</td>
<td>Accessor Methods</td>
<td>Provide a way of accessing an object’s state using specific methods. This approach discourages different client objects from directly accessing the attributes of an object, resulting in a more maintainable class structure.</td>
</tr>
<tr>
<td>7</td>
<td>Constant Data Manager</td>
<td>Useful for designing an easy to maintain, centralized repository for the constant data in an application.</td>
</tr>
<tr>
<td>8</td>
<td>Immutable Object</td>
<td>Used to ensure that the state of an object cannot be changed. May be used to ensure that the concurrent access to a data object by several client objects does not result in race conditions.</td>
</tr>
</tbody>
</table>
The Java programming language has built-in support for some of the fundamental design patterns in the form of language features. The other fundamental patterns can very easily be implemented using the Java language constructs.
This pattern was previously described in Grand98.

**DESCRIPTION**

In general, the functionality of an object-oriented system is encapsulated in the form of a set of objects. These objects provide different services either on their own or by interacting with other objects. In other words, a given object may rely upon the services offered by a different object to provide the service it is designed for. An object that requests a service from another object is referred as a client object. Some other objects in the system may seek the services offered by the client object.

From Figure 3.1, the client object assumes that the service provider objects corresponding to a specific service request are always of the same class type and interacts directly with the service provider object. This type of direct interaction ties the client with a specific class type for a given service request. This approach works fine when there is only one class of objects offering a given service, but may not be adequate when there is more than one class of objects that provide the same service required by the client (Figure 3.2). Because the client expects the service provider to be always of the same class type, it will not be able to make use of the different classes of service provider objects in a seamless manner. It requires changes to the design and implementation of the client and greatly reduces the reusability of the client by other objects.

In such cases, the Interface pattern can be used to better design different service provider classes that offer the same service to enable the client object to use different classes of service provider objects with little or no need for altering

![Figure 3.1 Client–Service Provider Interaction](image-url)
the client code. Applying the Interface pattern, the common services offered by different service provider classes can be abstracted out and declared as a separate interface. Each of the service provider classes can be designed as implementers of this common interface.

With this arrangement, the client can safely assume the service provider object to be of the interface type. From the class hierarchy in Figure 3.3, objects of different service provider classes can be treated as objects of the interface type. This enables the client to use different types of service provider objects in a seamless manner without requiring any changes. The client does not need to be altered even when a new service provider is designed as part of the class hierarchy in Figure 3.3.

**EXAMPLE**

Let us build an application to calculate and display the salaries of different employees of an organization with the categorization of designations as listed in Table 3.1.
Let us assume that the application needs to consider only those employees whose designations are part of Category-A. The salary calculation functionality for all employees of Category-A can be designed in the form of the `CategoryA` class as follows:

```java
public class CategoryA {
    double baseSalary;
    double OT;
    public CategoryA(double base, double overTime) {
        baseSalary = base;
        OT = overTime;
    }
    public double getSalary() {
        return (baseSalary + OT);
    }
}
```

The class representation of an employee, in its simplest form, can be designed as in the following listing with two attributes: the employee name and the category of designation.

```java
public class Employee {
    CategoryA salaryCalculator;
    String name;
    public Employee(String s, CategoryA c) {
        name = s;
        salaryCalculator = c;
    }
    public void display() {
        System.out.println("Name="+name);
        System.out.println("salary= "+
                          salaryCalculator.getSalary());
    }
}
```

<table>
<thead>
<tr>
<th>Designations</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programmer, Designer and Consultant</td>
<td>Category-A</td>
</tr>
<tr>
<td>Sales Rep, Sales Manager, Account Rep</td>
<td>Category-B</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>C-Level Executives</td>
<td>Category-n</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 3.1 Different Categories of Designations
A client object can configure an Employee object with values for the name and the category type attributes at the time of invoking its constructor. Subsequently the client object can invoke the display method to display the details of the employee name and salary. Because we are dealing only with employees who belong to Category-A, instances of the Employee class always expect the category type and hence the salary calculator to be always of the CategoryA type. As part of its implementation of the display method, the Employee class uses the salary calculation service provided by the CategoryA class.

The main application object MainApp that needs to display the salary details of employees performs the following tasks:

- Creates an instance of the CategoryA class by passing appropriate details required for the salary calculation.
- Creates an Employee object and configures it with the CategoryA object created above.
- Invokes the display method on the Employee object.
- The Employee object makes use of the services of the CategoryA object in calculating the salary of the employee it represents. In this aspect, the Employee object acts as a client to the CategoryA object.

public class MainApp {
    public static void main(String [] args) {
        CategoryA c = new CategoryA(10000, 200);
        Employee e = new Employee("Jennifer,"c);
        e.display();
    }
}

This design works fine as long as the need is to calculate the salary for Category-A employees only and there is only one class of objects that provides this service. But the fact that the Employee object expects the salary calculation service provider object to be always of the CategoryA class type affects the maintainability and results in an application design that is restrictive in terms of its adaptability.

Let us assume that the application also needs to calculate the salary of employees who are part of Category-B, such as sales representatives and account representatives, and the corresponding salary calculation service is provided by objects of a different class CategoryB.

public class CategoryB {
    double salesAmt;
    double baseSalary;
    final static double commission = 0.02;
    public CategoryB(double sa, double base) {
        baseSalary = base;
The main application object MainApp will be able to create an instance of the CategoryB class but will not be able to configure the Employee object with this instance. This is because the Employee object expects the salary calculator to be always of the CategoryA type. As a result, the main application will not be able to reuse the existing Employee class to represent different types of employees (Figure 3.4). The existing Employee class implementation needs to undergo necessary modifications to accept additional salary calculator service provider types. These limitations can be addressed by using the Interface pattern resulting in a much more flexible application design.

Applying the Interface pattern, the following three changes can be made to the application design.

1. The common salary calculating service provided by different objects can be abstracted out to a separate SalaryCalculator interface.

   ```java
   public interface SalaryCalculator {
       public double getSalary();
   }
   ```

2. Each of the CategoryA and the CategoryB classes can be designed as implementers of the SalaryCalculator interface (Figure 3.5).

   ```java
   public class CategoryA implements SalaryCalculator {
       double baseSalary;
       double OT;
   }
   ```
```java
public CategoryA(double base, double overTime) {
    baseSalary = base;
    OT = overTime;
}

public double getSalary() {
    return (baseSalary + OT);
}

public class CategoryB implements SalaryCalculator {
    double salesAmt;
    double baseSalary;
    final static double commission = 0.02;

    public CategoryB(double sa, double base) {
        baseSalary = base;
        salesAmt = sa;
    }

    public double getSalary() {
        return (baseSalary + (commission * salesAmt));
    }
}
```

3. The `Employee` class implementation needs to be changed to accept a salary calculator service provider of type `SalaryCalculator`.

```java
public class Employee {
    SalaryCalculator empType;
    String name;
```
public Employee(String s, SalaryCalculator c) {
    name = s;
    empType = c;
}
public void display() {
    System.out.println("Name= " + name);
    System.out.println("salary= " + empType.getSalary());
}

With these changes in place, the main application object MainApp can now create objects of different types of salary calculator classes and use them to configure different Employee objects. Because the Employee class, in the revised design, accepts objects of the SalaryCalculator type, it can be configured with an instance of any SalaryCalculator implementer class (or its subclass). Figure 3.6 shows the application object association.

```
public class MainApp {
    public static void main(String [] args) {
        SalaryCalculator c = new CategoryA(10000, 200);
        Employee e = new Employee("Jennifer",c);
        e.display();
        c = new CategoryB(20000, 800);
        e = new Employee("Shania",c);
        e.display();
    }
}
```

![Diagram](image_url)

Figure 3.6  Example Application/Class Association
PRACTICE QUESTIONS

1. Design a Search interface that declares methods for searching an item in a list. Design and implement two implementers — BinarySearch and LinearSearch — to conduct a binary and linear search of the list, respectively.

2. Design an AddressValidator interface that declares methods for validating different parts of a given address. Design and implement two implementer classes — USAddress and CAAddress — to validate a given U.S. and Canadian address, respectively.
ABSTRACT PARENT CLASS

This pattern was previously described in Grand98.

DESCRIPTION

The Abstract Parent Class pattern is useful for designing a framework for the consistent implementation of functionality common to a set of related classes.

An abstract method is a method that is declared, but contains no implementation. An abstract class is a class with one or more abstract methods. Abstract methods, with more than one possible implementation, represent variable parts of the behavior of an abstract class. An abstract class may contain implementations for other methods, which represent the invariable parts of the class functionality.

Different subclasses may be designed when the functionality outlined by abstract methods in an abstract class needs to be implemented differently. An abstract class, as is, may not be directly instantiated. When a class is designed as a subclass of an abstract class, it must implement all of the abstract methods declared in the parent abstract class. Otherwise the subclass itself becomes an abstract class. Only nonabstract subclasses of an abstract class can be instantiated. The requirement that every concrete subclass of an abstract class must implement all of its abstract methods ensures that the variable part of the functionality will be implemented in a consistent manner in terms of the method signatures. The set of methods implemented by the abstract parent class is automatically inherited by all subclasses. This eliminates the need for redundant implementations of these methods by each subclass. Figure 4.1 shows an abstract class with two concrete subclasses.

In the Java programming language there is no support for multiple inheritance. That means a class can inherit only from one single class. Hence inheritance should be used only when it is absolutely necessary. Whenever possible, methods denoting the common behavior should be declared in the form of a Java interface to be implemented by different implementer classes. But interfaces suffer from the limitation that they cannot provide method implementations. This means that every implementer of an interface must explicitly implement all methods declared in an interface, even when some of these methods represent the invariable part of the functionality and have exactly the same implementation in all of the
implementer classes. This leads to redundant code. The following example demonstrates how the Abstract Parent Class pattern can be used in such cases without requiring redundant method implementations.

**EXAMPLE**

In a typical organization, it is very common to have employees with different designations. This can be represented in form of a class hierarchy with a base *Employee* class and a set of subclasses each corresponding to employees with a specific designation.

Let us consider the following operations as part of designing the representation of an employee.

1. Save employee data
2. Display employee data
3. Access employee attributes such as name and ID
4. Calculate compensation

While Operation 1 through Operation 3 remain the same for all employees, the compensation calculation will be different for employees with different designations. Such an operation, which can be performed in different ways, is a good candidate to be declared as an abstract method. This forces different concrete subclasses of the *Employee* class to provide a custom implementation for the salary calculation operation.

From the base *Employee* class implementation in Listing 4.1, it can be seen that the base *Employee* class provides implementation for the `save`, `getID`, `getName` and `toString` methods while it declares the `computeCompensation` method as an abstract method.

Let us define two concrete subclasses — *Consultant* and *SalesRep* — of the *Employee* class (Listing 4.2) representing employees who are consultants and sales representatives, respectively. Each of these subclasses must implement the `computeCompensation` method. Otherwise these subclasses need to be
declared as abstract and it becomes impossible to instantiate them. Figure 4.2 shows the class hierarchy with Consultant and SalesRep concrete subclasses of the Employee class.

Abstract Parent Class versus Interface

As an alternate design strategy, we could design the employee representation as a Java interface, instead of designing it as an abstract class, with both the Consultant and the SalesRep classes as its implementers. Figure 4.3 shows the resulting class hierarchy.

But doing so would require both the implementers to implement the save, getID, getName, toString and the computeCompensation methods. Because the implementation of the save, getID, getName and toString
Listing 4.2  Concrete Employee Subclasses

```java
public class Consultant extends Employee {
    public String computeCompensation() {
        return "consultant salary is base + " + " allowance + OT - tax deductions";
    }
    public Consultant(String empName, String empID) {
        super(empName, empID);
    }
}

public class SalesRep extends Employee {
    //variable part behavior
    public String computeCompensation() {
        return "sales Rep Salary is Base + commission + " + " allowance - tax deductions";
    }
    public SalesRep(String empName, String empID) {
        super(empName, empID);
    }
}
```

Figure 4.2  Employee Class Hierarchy
methods remains the same for all implementers, this leads to redundant code in the application. The implementation of these invariable methods cannot be made part of the Employee interface. This is because a Java interface cannot provide implementation for a method. An interface is used for the declaration purpose only. By designing the Employee class as an abstract class, the need for a redundant implementation can be eliminated.

**PRACTICE QUESTIONS**

1. Consider the details of different bank account types as follows:
   a. All bank accounts allow
      i. Deposits
      ii. Balance enquiries
   b. Savings accounts
      i. Allow no checking
      ii. Do not charge service fee
      iii. Give interest
   c. Checking accounts
      i. Allow checking
      ii. Charge service fee
      iii. Do not give interest

   Design a class hierarchy with Account as an abstract class with the class representations for both the savings account and the checking account as two concrete subclasses of it.
2. Both the right-angled triangle and the equilateral triangle are triangles with specific differences. Design a class hierarchy with **Triangle** as an abstract class with the class representations for both the right-angled triangle and the equilateral triangle as two concrete subclasses of it.
PRIVATE METHODS

DESCRIPTION
Typically a class is designed to offer a well-defined and related set of services to its clients. These services are offered in the form of its methods, which constitute the overall behavior of that object. In case of a well-designed class, each method is designed to perform a single, defined task. Some of these methods may use the functionality offered by other methods or even other objects to perform the task they are designed for. Not all methods of a class are always meant to be used by external client objects. Those methods that offer defined services to different client objects make up an object's public protocol and are to be declared as public methods. Some of the other methods may exist to be used internally by other methods or inner classes of the same object. The Private Methods pattern recommends designing such methods as private methods.

In Java, a method signature starts with an access specifier (private/protected/public). Access specifiers indicate the scope and visibility of a method/variable. A method is declared as private by using the “private” keyword as part of its signature. e.g.,

```java
private int hasValidChars()
{
    //...
}
```

External client objects cannot directly access private methods. This in turn hides the behavior contained in these methods from client objects.

EXAMPLE
Let us design an OrderManager class as in Figure 5.1 that can be used by different client objects to create orders.

```java
public class OrderManager {
    private int orderID = 0;
    //Meant to be used internally
```
private int getNextID() {
    ++orderID;
    return orderID;
}
// public method to be used by client objects
public void saveOrder(String item, int qty) {
    int ID = getNextID();
    System.out.println("Order ID=\" + ID + \"; Item=\" + item + \
    \"; Qty=\" + qty + \" is saved. ");
}

From the OrderManager implementation it can be observed that the saveOrder
method is declared as public as it is meant to be used by client objects, whereas
the getNextID method is used internally by the saveOrder method and is not
meant to be used by client objects directly. Hence the getNextID method is
designed as a private method. This automatically prevents client objects from
accessing the getNextID method directly.

PRACTICE QUESTIONS

1. Design a CreditCard class, which offers the functionality to validate
   credit card numbers. Design the card validation method to internally use
   a private method to check if the card number has valid characters.
2. The OrderManager class built during the example discussion does not
define a constructor. Add a private constructor to the OrderManager
class. What changes must be made to the OrderManager class so that
client objects can create OrderManager instances?
ACCESSOR METHODS

DESCRIPTION

The Accessor Methods pattern is one of the most commonly used patterns in the area of object-oriented programming. In fact, this pattern has been used in most of the examples discussed in this book for different patterns. In general, the values of different instance variables of an object, at a given point of time, constitute its state. The state of an object can be grouped into two categories — public and private. The public state of an object is available to different client objects to access, whereas the private state of an object is meant to be used internally by the object itself and not to be accessed by other objects.

Consider the class representation of a customer in Figure 6.1.

The instance variable ID is maintained separately and used internally by each Customer class instance and is not to be known by other objects. This makes the variable ID the private state of a Customer object to be used internally by the Customer object. On the other hand, variables such as name, SSN (Social Security Number) and the address make up the public state of the Customer object and are supposed to be used by client objects. In case of such an object, the Accessor Method pattern recommends:

- All instance variables being declared as private and provide public methods known as accessor methods to access the public state of an object. This prevents external client objects from accessing object instance variables directly. In addition, accessor methods hide from the client whether a property is stored as a direct attribute or as a derived one.

```
<table>
<thead>
<tr>
<th>Customer</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID:int</td>
</tr>
<tr>
<td>name:String</td>
</tr>
<tr>
<td>SSN:String</td>
</tr>
<tr>
<td>address:String</td>
</tr>
</tbody>
</table>
```

Figure 6.1  Customer Class
- Client objects can make use of accessor methods to move a Customer object from one state (source) to another state (target). In general, if the object cannot reach the target state, it should notify the caller object that the transition could not be completed. This can be accomplished by having the accessor method throw an exception.

- An object can access its private variables directly. But doing so could greatly affect the maintainability of an application, which the object is part of. When there is a change in the way a particular instance variable is to be defined, it requires changes to be made in every place of the application code where the instance variable is referenced directly. Similar to its client objects, if an object is designed to access its instance variables through accessor methods, any change to the definition of an instance variable requires a change only to its accessor methods.

**ACCESSOR METHOD NOMENCLATURE**

There is no specific requirement for an accessor method to be named following a certain naming convention. But most commonly the following naming rules are followed:

- To access a non-Boolean instance variable:
  - Define a `getXXXX()` method to read the values of an instance variable `XXXX`. E.g., define a `getFirstName()` method to read the value of an instance variable named `firstName`.
  - Define a `setXXXX(new value)` method to alter the value of an instance variable `XXXX`. E.g., define a `setFirstName(String)` method to alter the value of an instance variable named `firstName`.

- To access a Boolean instance variable:
  - Define an `isXXXX()` method to check if the value of an instance variable `XXXX` is true or false. E.g., define an `isActive()` method on a Customer object to check if the customer represented by the Customer object is active.
  - Define a `setXXXX(new value)` method to alter the value of a Boolean instance variable `XXXX`. E.g., define a `setActive(boolean)` method on a Customer object to mark the customer as active.

The following Customer class example explains the usage of accessor methods.

**EXAMPLE**

Suppose that you are designing a Customer class as part of a large application. A generic representation of a customer in its simplest form can be designed as in Figure 6.2.

Applying the Accessor Method pattern, the set of accessor methods listed in Table 6.1 can be defined corresponding to each of the instance variables (Listing 6.1).

Figure 6.3 shows the resulting class structure.
Different client objects can access the object state variables using the accessor methods listed in Table 6.1. The Customer object itself can access its state variables directly, but using the accessor methods will greatly improve the maintainability of the Customer class code. This in turn contributes to the overall application maintainability.

### DIRECT REFERENCE VERSUS ACCESSOR METHODS

Let us suppose that we need to add the following two new methods to the Customer class.

1. isValidCustomer — To check if the customer data is valid.
2. save — To save the customer data to a data file.

As can be seen from the Customer class implementation in Listing 6.2, the newly added methods access different instance variables directly. Different client
objects can use the Customer class in this form without any difficulty. But when there is a change in the definition of any of the instance variables, it requires a change to the implementation of all the methods that access these instance variables directly. For example, if the address variable need to be changed from its current definition as a string to a StringBuffer or something different, then all methods that refer to the address variable directly needs to be altered.

As an alternative approach, Customer object methods can be redesigned to access the object state through its accessor methods (Listing 6.3).

Listing 6.1 Customer Class with Accessor Methods

```java
public class Customer {
    private String firstName;
    private String lastName;
    private String address;
    private boolean active;
    public String getFirstName() {
        return firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public String getAddress() {
        return address;
    }
    public boolean isActive() {
        return active;
    }
    public void setFirstName(String newValue) {
        firstName = newValue;
    }
    public void setLastName(String newValue) {
        lastName = newValue;
    }
    public void setAddress(String newValue) {
        address = newValue;
    }
    public void isActive(boolean newValue) {
        active = newValue;
    }
}
```
In this approach, any change to the definition of any of the instance variables requires a change only to the implementation of the corresponding accessor methods. No changes are required for any other part of the class implementation and the class becomes more maintainable.

**PRACTICE QUESTIONS**

1. Design an Order class with accessor methods for its instance variables.
2. Identify the effect of using accessor methods when a class is subclassed.
Listing 6.2 Customer Class Directly Accessing Its Instance Variables

```java
public class Customer {
    ...
    ...
    public String getFirstName() {
        return firstName;
    }
    ...
    ...
    public boolean isValidCustomer() {
        if ((firstName.length() > 0) && (lastName.length() > 0) &&
            (address.length() > 0))
            return true;
        return false;
    }
    public void save() {
        String data =
            firstName + "," + lastName + "," + address + 
            "," + active;
        FileUtil futil = new FileUtil();
        futil.writeToFile("customer.txt", data, true, true);
    }
}
```
public class Customer {
  ...
  ...
  public String getFirstName() {
    return firstName;
  }
  ...
  ...
  public boolean isValidCustomer() {
    if ((getFirstName().length() > 0) &&
        (getLastName().length() > 0) &&
        (getAddress().length() > 0))
      return true;
    return false;
  }
  public void save() {
    String data =
      getFirstName() + "," + getLastName() + "," +
      getAddress() + "," + isActive();
    FileUtil futil = new FileUtil();
    futil.writeToFile("customer.txt", data, true, true);
  }
}
CONSTANT DATA MANAGER

DESCRIPTION

Objects in an application usually make use of different types of data in offering the functionality they are designed for. Such data can either be variable data or constant data. The Constant Data Manager pattern is useful for designing an efficient storage mechanism for the constant data used by different objects in an application. In general, application objects access different types of constant data items such as data file names, button labels, maximum and minimum range values, error codes and error messages, etc.

Instead of allowing the constant data to be present in different objects, the Constant Data Manager pattern recommends all such data, which is considered as constant in an application, be kept in a separate object and accessed by other objects in the application. This type of separation provides an easy to maintain, centralized repository for the constant data in an application.

EXAMPLE

Let us consider a Customer Data Management application that makes use of three types of objects — Account, Address and CreditCard — to represent different parts of the customer data (Figure 7.1). Each of these objects makes use of different items of constant data as part of offering the services it is designed for (Listing 7.1).

Instead of allowing the distribution of the constant data across different classes, it can be encapsulated in a separate ConstantDataManager (Listing 7.2) object and is accessed by each of the Account, Address and CreditCard objects.

The interaction among these classes can be depicted as in Figure 7.2.

Whenever any of the constant data items needs to be modified, only the ConstantDataManager needs to be altered without affecting other application objects. On the other side, it is easy to lose track of constants that do not get used anymore when code gets thrown out over the years but constants remain in the class.
PRACTICE QUESTIONS

1. Constant data can also be declared in a Java interface. Any class that implements such an interface can use the constants declared in it without any qualifications. Redesign the example application with the ConstantDataManager as an interface.

2. Identify how the Constant Data Manager pattern can be used to store different application-specific error messages.

Figure 7.1 Different Application Objects
3. The ConstantDataManager in Listing 7.2 contains hard-coded values for different constant items. Enhance the ConstantDataManager class to read values from a file and initialize different constant data items when it is first constructed.
Listing 7.2  ConstantDataManager Class

public class ConstantDataManager {
    public static final String ACCOUNT_DATA_FILE = "ACCOUNT.TXT";
    public static final int VALID_MIN_LNAME_LEN = 2;
    public static final String ADDRESS_DATA_FILE = "ADDRESS.TXT";
    public static final int VALID_ST_LEN = 2;
    public static final String VALID_ZIP_CHARS = "0123456789";
    public static final String DEFAULT_COUNTRY = "USA";
    public static final String CC_DATA_FILE = "CC.TXT";
    public static final String VALID_CC_CHARS = "0123456789";
    public static final String MASTER = "MASTER";
    public static final String VISA = "VISA";
    public static final String DISCOVER = "DISCOVER";
}
Figure 7.2  Different Application Objects Access the ConstantDataManager for the Constant Data
IMMUTABLE OBJECT

This pattern was previously described in Grand98.

DESCRIPTION

In general, classes in an application are designed to carry data and have behavior. Sometimes a class may be designed in such a way that its instances can be used just as carriers of related data without any specific behavior. Such classes can be called data model classes and instances of such classes are referred to as data objects. For example, consider the Employee class in Figure 8.1 and Listing 8.1.

```
<table>
<thead>
<tr>
<th>Employee</th>
</tr>
</thead>
<tbody>
<tr>
<td>firstName:String</td>
</tr>
<tr>
<td>lastName:String</td>
</tr>
<tr>
<td>SSN:String</td>
</tr>
<tr>
<td>address:String</td>
</tr>
<tr>
<td>car:Car</td>
</tr>
<tr>
<td>getFirstName():String</td>
</tr>
<tr>
<td>getLastName():String</td>
</tr>
<tr>
<td>getSSN():String</td>
</tr>
<tr>
<td>getAddress():String</td>
</tr>
<tr>
<td>getCar():Car</td>
</tr>
<tr>
<td>setFirstName(fname:String)</td>
</tr>
<tr>
<td>setLastName(lname:String)</td>
</tr>
<tr>
<td>setSSN(ssn:String)</td>
</tr>
<tr>
<td>setAddress(addr:String)</td>
</tr>
<tr>
<td>setCar(c:Car)</td>
</tr>
<tr>
<td>save():boolean</td>
</tr>
<tr>
<td>delete():boolean</td>
</tr>
<tr>
<td>isValid():boolean</td>
</tr>
<tr>
<td>update():boolean</td>
</tr>
</tbody>
</table>
```

Figure 8.1  Employee Representation
public class Employee {
    // State
    private String firstName;
    private String lastName;
    private String SSN;
    private String address;
    private Car car;
    // Constructor
    public Employee(String fn, String ln, String ssn, String addr, Car c) {
        firstName = fn;
        lastName = ln;
        SSN = ssn;
        address = addr;
        car = c;
    }
    // Behavior
    public boolean save() {
        // ...
        return true;
    }
    public boolean isValid() {
        // ...
        return true;
    }
    public boolean update() {
        // ...
        return true;
    }
}

(continued)
Instances of the Employee class above have both the data and the behavior. The corresponding data model class can be designed as in Figure 8.2 and Listing 8.2 without any behavior.

In a typical application scenario, several client objects may simultaneously access instances of such data model classes. This could lead to problems if changes
public class EmployeeModel {
    // State
    private String firstName;
    private String lastName;
    private String SSN;
    private String address;
    private Car car;

    // Constructor
    public EmployeeModel(String fn, String ln, String ssn, String addr, Car c) {
        firstName = fn;
        lastName = ln;
        SSN = ssn;
        address = addr;
        car = c;
    }

    (continued)
to the state of a data object are not coordinated properly. The Immutable Object pattern can be used to ensure that the concurrent access to a data object by several client objects does not result in any problem. The Immutable Object pattern accomplishes this without involving the overhead of synchronizing the methods to access the object data.
Applying the Immutable Object pattern, the data model class can be designed in such a way that the data carried by an instance of the data model class remains unchanged over its entire lifetime. That means the instances of the data model class become immutable.

In general, concurrent access to an object creates problems when one thread can change data while a different thread is reading the same data. The fact that the data of an immutable object cannot be modified makes it automatically thread-safe and eliminates any concurrent access related problems.

Though using the Immutable Object pattern opens up an application for all kinds of performance tuning tricks, it must be noted that designing an object as immutable is an important decision. Every now and then it turns out that objects that were once thought of as immutables are in fact mutable, which could result in difficult implementation changes.

**EXAMPLE**

As an example, let us redesign the `EmployeeModel` class to make it immutable by applying the following changes.

1. All instance variables (state) must be set in the constructor alone. No other method should be provided to modify the state of the object. The constructor is automatically thread-safe and hence does not lead to problems.
2. It may be possible to override class methods to modify the state. In order to prevent this, declare the class as `final`. Declaring a class as final does not allow the class to be extended further.
3. All instance variables should be declared `final` so that they can be set only once, inside the constructor.
4. If any of the instance variables contain a reference to an object, the corresponding getter method should return a copy of the object it refers to, but not the actual object itself.

Figure 8.3 and Listing 8.3 show the resulting immutable version of the `EmployeeModel` class.

The immutable version of the `EmployeeModel` objects can safely be used in a multithreaded environment.

**PRACTICE QUESTIONS**

1. Design an immutable class that contains the line styles and colors used in a given image.
2. Design an immutable class to carry the data related to a company such as the company address, phone, fax, company name and other details.
Figure 8.3 EmployeeModel Class: Immutable Version
Listing 8.3  EmployeeModel Class: Immutable Version

public final class EmployeeModel {
    // State
    private final String firstName;
    private final String lastName;
    private final String SSN;
    private final String address;
    private final Car car;
    // Constructor
    public EmployeeModel(String fn, String ln, String ssn,
                          String addr, Car c) {
        firstName = fn;
        lastName = ln;
        SSN = ssn;
        address = addr;
        car = c;
    }
    // Getters
    public String getFirstName() {
        return firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public String getSSN() {
        return SSN;
    }
    public Car getCar() {
        // return a copy of the car object
        return (Car) car.clone();
    }
    public String getAddress() {
        return address;
    }
}
CREATIONAL PATTERNS

Creational Patterns:

Deal with one of the most commonly performed tasks in an OO application, the creation of objects. Support a uniform, simple, and controlled mechanism to create objects. Allow the encapsulation of the details about what classes are instantiated and how these instances are created. Encourage the use of interfaces, which reduces coupling.

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Pattern Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Factory Method</td>
<td>When a client object does not know which class to instantiate, it can make use of the factory method to create an instance of an appropriate class from a class hierarchy or a family of related classes. The factory method may be designed as part of the client itself or in a separate class. The class that contains the factory method or any of its subclasses decides on which class to select and how to instantiate it.</td>
</tr>
<tr>
<td>2</td>
<td>Singleton</td>
<td>Provides a controlled object creation mechanism to ensure that only one instance of a given class exists.</td>
</tr>
<tr>
<td>3</td>
<td>Abstract Factory</td>
<td>Allows the creation of an instance of a class from a suite of related classes without having a client object to specify the actual concrete class to be instantiated.</td>
</tr>
<tr>
<td>4</td>
<td>Prototype</td>
<td>Provides a simpler way of creating an object by cloning it from an existing (prototype) object.</td>
</tr>
<tr>
<td>5</td>
<td>Builder</td>
<td>Allows the creation of a complex object by providing the information on only its type and content, keeping the details of the object creation transparent to the client. This allows the same construction process to produce different representations of the object.</td>
</tr>
</tbody>
</table>
FACTORY METHOD

DESCRIPTION

In general, all subclasses in a class hierarchy inherit the methods implemented by the parent class. A subclass may override the parent class implementation to offer a different type of functionality for the same method. When an application object is aware of the exact functionality it needs, it can directly instantiate the class from the class hierarchy that offers the required functionality.

At times, an application object may only know that it needs to access a class from within the class hierarchy, but does not know exactly which class from among the set of subclasses of the parent class is to be selected. The choice of an appropriate class may depend on factors such as:

- The state of the running application
- Application configuration settings
- Expansion of requirements or enhancements

In such cases, an application object needs to implement the class selection criteria to instantiate an appropriate class from the hierarchy to access its services (Figure 1.1).

This type of design has the following disadvantages:

Because every application object that intends to use the services offered by the class hierarchy needs to implement the class selection criteria, it results in a high degree of coupling between an application object and the service provider class hierarchy.

Whenever the class selection criteria change, every application object that uses the class hierarchy must undergo a corresponding change.

Because class selection criteria needs to take all the factors that could affect the selection process into account, the implementation of an application object could contain inelegant conditional statements.
If different classes in the class hierarchy need to be instantiated in diverse manners, the implementation of an application object can become more complex. It requires an application object to be fully aware of the existence and the functionality offered by each class in the service provider class hierarchy.

In such cases, the Factory Method pattern recommends encapsulating the functionality required, to select and instantiate an appropriate class, inside a designated method referred to as a *factory method*. Thus, a factory method can be defined as a method in a class that:

- Selects an appropriate class from a class hierarchy based on the application context and other influencing factors
- Instantiates the selected class and returns it as an instance of the parent class type

Encapsulation of the required implementation to select and instantiate an appropriate class in a separate method has the following advantages:

Application objects can make use of the factory method to get access to the appropriate class instance. This eliminates the need for an application object to deal with the varying class selection criteria. Besides the class selection criteria, the factory method also implements any special mechanisms required to instantiate the selected class. This is applicable if different classes in the hierarchy need to be instantiated in different ways. The factory method hides these details from application objects and eliminates the need for them to deal with these intricacies.
Because the factory method returns the selected class instance as an object of the parent class type, an application object does not have to be aware of the existence of the classes in the hierarchy.

One of the simplest ways of designing a factory method is to create an abstract class or an interface that just declares the factory method. Different subclasses (or implementer classes in the case of an interface) can be designed to implement the factory method in its entirety as depicted in Figure 1.2. Another strategy is to create a concrete creator class with default implementation for the factory method in it. Different subclasses of this concrete class can override the factory method to implement specialized class selection criteria.
DESCRIPTION

The Singleton pattern is an easy to understand design pattern. Sometimes, there may be a need to have one and only one instance of a given class during the lifetime of an application. This may be due to necessity or, more often, due to the fact that only a single instance of the class is sufficient. For example, we may need a single database connection object in an application. The Singleton pattern is useful in such cases because it ensures that there exists one and only one instance of a particular object ever. Further, it suggests that client objects should be able to access the single instance in a consistent manner.

WHO SHOULD BE RESPONSIBLE?

Having an instance of the class in a global variable seems like an easy way to maintain the single instance. All client objects can access this instance in a consistent manner through this global variable. But this does not prevent clients from creating other instances of the class. For this approach to be successful, all of the client objects have to be responsible for controlling the number of instances of the class. This widely distributed responsibility is not desirable because a client should be free from any class creation process details. The responsibility for making sure that there is only one instance of the class should belong to the class itself, leaving client objects free from having to handle these details.

A class that maintains its single instance nature by itself is referred to as a Singleton class.
ABSTRACT FACTORY

DESCRIPTION

During the discussion of the Factory Method pattern we saw that:

In the context of a factory method, there exists a class hierarchy composed of a set of subclasses with a common parent class.

A factory method is used when a client object knows when to create an instance of the parent class type, but does not know (or should not know) exactly which class from among the set of subclasses (and possibly the parent class) should be instantiated. Besides the class selection criteria, a factory method also hides any special mechanism required to instantiate the selected class.

The Abstract Factory pattern takes the same concept to the next level. In simple terms, an abstract factory is a class that provides an interface to produce a family of objects. In the Java programming language, it can be implemented either as an interface or as an abstract class.

In the context of an abstract factory there exist:

Suites or families of related, dependent classes.

A group of concrete factory classes that implements the interface provided by the abstract factory class. Each of these factories controls or provides access to a particular suite of related, dependent objects and implements the abstract factory interface in a manner that is specific to the family of classes it controls.

The Abstract Factory pattern is useful when a client object wants to create an instance of one of a suite of related, dependent classes without having to know which specific concrete class is to be instantiated. In the absence of an abstract factory, the required implementation to select an appropriate class (in other words, the class selection criterion) needs to be present everywhere such an instance is created. An abstract factory helps avoid this duplication by providing the necessary
interface for creating such instances. Different concrete factories implement this interface. Client objects make use of these concrete factories to create objects and, therefore, do not need to know which concrete class is actually instantiated. Figure 3.1 shows the generic class association when the Abstract Factory pattern is applied.

The abstract factory shown in the Figure 3.1 class diagram is designed as a Java interface with its implementers as concrete factories. In Java, an abstract factory can also be designed as an abstract class with its concrete subclasses as factories, where each factory is responsible for creating and providing access to the objects of a particular suite of classes.

**ABSTRACT FACTORY VERSUS FACTORY METHOD**

Abstract Factory is used to create groups of related objects while hiding the actual concrete classes. This is useful for plugging in a different group of objects to alter the behavior of the system. For each group or family, a concrete factory is implemented that manages the creation of the objects and the interdependencies and consistency requirements between them. Each concrete factory implements the interface of the abstract factory.

This situation often arises when designing a framework or a library, which needs to be kept extensible. One example is the JDBC (Java Database Connectivity)
driver system, where each driver contains classes that implement the `Connection`, `Statement` and `ResultSet` interfaces. The set of classes that the Oracle JDBC driver contains are different from the set of classes that the DB2 JDBC driver contains and they must not be mixed up. This is where the role of the factory comes in: It knows which classes belong together and how to create objects in a consistent way.

Factory Method is specifying a method for the creation of an object, thus allowing subclasses or implementing classes to define the concrete object. Abstract Factories are usually implemented using the Factory Method pattern. Another approach would be to use the Prototype pattern.
As discussed in earlier chapters, both the Factory Method and the Abstract Factory patterns allow a system to be independent of the object creation process. In other words, these patterns enable a client object to create an instance of an appropriate class by invoking a designated method without having to specify the exact concrete class to be instantiated. While addressing the same problem as the Factory Method and Abstract Factory patterns, the Prototype pattern offers a different, more flexible way of achieving the same result.

Other uses of the Prototype pattern include:

- When a client needs to create a set of objects that are alike or differ from each other only in terms of their state and it is expensive to create such objects in terms of the time and the processing involved.
- As an alternative to building numerous factories that mirror the classes to be instantiated (as in the Factory Method).

In such cases, the Prototype pattern suggests to:

- Create one object upfront and designate it as a prototype object.
- Create other objects by simply making a copy of the prototype object and making required modifications.

In the real world, we use the Prototype pattern on many occasions to reduce the time and effort spent on different tasks. The following are two such examples:

1. **New Software Program Creation** — Typically programmers tend to make a copy of an existing program with similar structure and modify it to create new programs.
2. **Cover Letters** — When applying for positions at different organizations, an applicant may not create cover letters for each organization individually from scratch. Instead, the applicant would create one cover letter in the
most appealing format, make a copy of it and personalize it for every organization.

As can be seen from the examples above, some of the objects are created from scratch, whereas other objects are created as copies of existing objects and then modified. But the system or the process that uses these objects does not differentiate between them on the basis of how they are actually created. In a similar manner, when using the Prototype pattern, a system should be independent of the creation, composition and representation details of the objects it uses.

One of the requirements of the prototype object is that it should provide a way for clients to create a copy of it. By default, all Java objects inherit the built-in `clone()` method from the topmost `java.lang.Object` class. The built-in `clone()` method creates a clone of the original object as a shallow copy.

**SHALLOW COPY VERSUS DEEP COPY**

When an object is cloned as a shallow copy:

The original top-level object and all of its primitive members are duplicated. Any lower-level objects that the top-level object contains are not duplicated. Only references to these objects are copied. This results in both the original and the cloned object referring to the same copy of the lower-level object. Figure 4.1 shows this behavior.

In contrast, when an object is cloned as a deep copy:

The original top-level object and all of its primitive members are duplicated. Any lower-level objects that the top-level object contains are also duplicated. In this case, both the original and the cloned object refer to two different lower-level objects. Figure 4.2 shows this behavior.

![Figure 4.1 Shallow Copy](image-url)
Figure 4.2 Deep Copy

**Shallow Copy Example**

The following is an example of creating a shallow copy using the built-in java.lang.Object clone() method. Let us design a Person class (Listing 4.1) as an implementer of the built-in Java java.lang.Cloneable interface with two attributes, a string variable name and a Car object car.

In general, a class must implement the Cloneable interface to indicate that a field-for-field copy of instances of that class is allowed by the Object.clone() method. When a class implements the Cloneable interface, it should override the Object.clone method with a public method. Note that when the clone method is invoked on an object that does not implement the Cloneable interface, the exception CloneNotSupportedException is thrown.

As part of its implementation of the public clone method, the Person class simply invokes the built-in clone method. The built-in clone method creates a clone of the current object as a shallow copy, which is returned to the calling client object.

**Deep Copy Example**

The same example above can be redesigned by overriding the built-in clone() method to create a deep copy of the Person object (Listing 4.3). As part of its implementation of the clone method, to create a deep copy, the Person class creates a new Person object with its attribute values the same as the original object and returns it to the client object.
In general, object construction details such as instantiating and initializing the components that make up the object are kept within the object, often as part of its constructor. This type of design closely ties the object construction process with the components that make up the object. This approach is suitable as long as the object under construction is simple and the object construction process is definite and always produces the same representation of the object.

This design may not be effective when the object being created is complex and the series of steps constituting the object creation process can be implemented in different ways producing different representations of the object. Because different implementations of the construction process are all kept within the object, the object can become bulky (construction bloat) and less modular. Subsequently, adding a new implementation or making changes to an existing implementation requires changes to the existing code.

Using the Builder pattern, the process of constructing such an object can be designed more effectively. The Builder pattern suggests moving the construction logic out of the object class to a separate class referred to as a builder class. There can be more than one such builder class each with different implementation for the series of steps to construct the object. Each such builder implementation results in a different representation of the object. This type of separation reduces the object size. In addition:

- The design turns out to be more modular with each implementation contained in a different builder object.
- Adding a new implementation (i.e., adding a new builder) becomes easier. The object construction process becomes independent of the components that make up the object. This provides more control over the object construction process.

In terms of implementation, each of the different steps in the construction process can be declared as methods of a common interface to be implemented by different concrete builders. Figure 5.1 shows the resulting builder class hierarchy.
A client object can create an instance of a concrete builder and invoke the set of methods required to construct different parts of the final object. Figure 5.2 shows the corresponding message flow.

This approach requires every client object to be aware of the construction logic. Whenever the construction logic undergoes a change, all client objects need to be modified accordingly. The Builder pattern introduces another level of separation that addresses this problem. Instead of having client objects invoke different builder methods directly, the Builder pattern suggests using a dedicated object referred to as a *Director*, which is responsible for invoking different builder
methods required for the construction of the final object. Different client objects can make use of the Director object to create the required object. Once the object is constructed, the client object can directly request from the builder the fully constructed object. To facilitate this process, a new method `getObject` can be declared in the common Builder interface to be implemented by different concrete builders.

The new design eliminates the need for a client object to deal with the methods constituting the object construction process and encapsulates the details of how the object is constructed from the client. Figure 5.3 shows the association between different classes.

The interaction between the client object, the Director and the Builder objects can be summarized as follows:

The client object creates instances of an appropriate concrete Builder implementer and the Director. The client may use a factory for creating an appropriate Builder object.

The client associates the Builder object with the Director object.

The client invokes the `build` method on the Director instance to begin the object creation process. Internally, the Director invokes different Builder methods required to construct the final object.

Once the object creation is completed, the client invokes the `getObject` method on the concrete Builder instance to get the newly created object. Figure 5.4 shows the overall message flow.

![Figure 5.3 Class Association](image-url)
Figure 5.4  Object Creation When the Builder Pattern Is Applied
Collectional patterns primarily:

- Deal with groups or collections of objects
- Deal with the details of how to compose classes and objects to form larger structures
- Concentrate on the most efficient way of designing a class so that its instances do not carry any duplicate data
- Allow the definition of operations on collections of objects

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Pattern Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Composite</td>
<td>Allows both individual objects and composite objects to be treated uniformly.</td>
</tr>
<tr>
<td>7</td>
<td>Iterator</td>
<td>Allows a client to access the contents of an aggregate object (collection of objects) in some sequential manner, without having any knowledge about the internal representation of its contents.</td>
</tr>
<tr>
<td></td>
<td>Flyweight</td>
<td>The intrinsic, invariant common information and the variable parts of a class are separated into two classes, leading to savings in terms of the memory usage and the amount of time required for the creation of a large number of its instances.</td>
</tr>
<tr>
<td></td>
<td>Visitor</td>
<td>Allows an operation to be defined across a collection of different objects without changing the classes of objects on which it operates.</td>
</tr>
</tbody>
</table>
Every component or object can be classified into one of the two categories — Individual Components or Composite Components — which are composed of individual components or other composite components. The Composite pattern is useful in designing a common interface for both individual and composite components so that client programs can view both the individual components and groups of components uniformly. In other words, the Composite design pattern allows a client object to treat both single components and collections of components in an identical manner.

This can also be explained in terms of a tree structure. The Composite pattern allows uniform reference to both Nonterminal nodes (which represent collections of components or composites) and terminal nodes (which represent individual components).

**EXAMPLE**

Let us create an application to simulate the Windows/UNIX file system. The file system consists mainly of two types of components — directories and files. Directories can be made up of other directories or files, whereas files cannot contain any other file system component. In this aspect, directories act as non-terminal nodes and files act as terminal nodes of a tree structure.

**DESIGN APPROACH I**

Let us define a common interface for both directories and files in the form of a Java interface FileSystemComponent (Figure 6.1). The FileSystemComponent interface declares methods that are common for both file components and directory components.

Let us further define two classes — FileComponent and DirComponent — as implementers of the common FileSystemComponent interface. Figure 6.2 shows the resulting class hierarchy.
The `FileSystemComponent` interface contains the method `getComponentSize(): long`.

**Figure 6.1** The Common `FileSystemComponent` Interface

The `FileSystemComponent` class hierarchy includes:
- `FileComponent`
- `DirComponent`

The `FileComponent` class represents a file in the file system and offers implementation for the following methods.

**getComponentSize()**
This method returns the size (in kilobytes) of the file represented by the `FileComponent` object.

The `DirComponent` class represents a directory in the file system. Since directories are composite entities, the `DirComponent` provides methods to deal with the components it contains. These methods are in addition to the common `getComponentSize` method declared in the `FileSystemComponent` interface.

**Figure 6.2** The `FileSystemComponent` Class Hierarchy

FileComponent
The `FileComponent` class represents a file in the file system and offers implementation for the following methods.

**getComponentSize()**
This method returns the size (in kilobytes) of the file represented by the `FileComponent` object.

DirComponent
This class represents a directory in the file system. Since directories are composite entities, the `DirComponent` provides methods to deal with the components it contains. These methods are in addition to the common `getComponentSize` method declared in the `FileSystemComponent` interface.
**addComponent(FileSystemComponent)**

This method is used by client applications to add different DirComponent and FileComponent objects to a DirComponent object.

**getComponent(int)**

The DirComponent stores the other FileSystemComponent objects inside a vector. This method is used to retrieve one such object stored at the specified location.

**getComponentSize()**

This method returns the size (in kilobytes) of the directory represented by the DirComponent object. As part of the implementation, the DirComponent object iterates over the collection of FileSystemComponent objects it contains, in a recursive manner, and sums up the sizes of all individual FileComponents. The final sum is returned as the size of the directory it represents.

A typical client would first create a set of FileSystemComponent objects (both DirComponent and FileComponent instances). It can use the addComponent method of the DirComponent to add different FileSystemComponents to a DirComponent, creating a hierarchy of file system (FileSystemComponent) objects.

When the client wants to query any of these objects for its size, it can simply invoke the getComponentSize method. The client does not have to be aware of the calculations involved or the manner in which the calculations are carried out in determining the component size. In this aspect, the client treats both the FileComponent and the DirComponent object in the same manner. No separate code is required to query FileComponent objects and DirComponent objects for their size.

Though the client treats both the FileComponent and DirComponent objects in a uniform manner in the case of the common getComponentSize method, it does need to distinguish when calling composite specific methods such as addComponent and getComponent defined exclusively in the DirComponent. Because these methods are not available with FileComponent objects, the client needs to check to make sure that the FileSystemComponent object it is working with is in fact a DirComponent object.

The following Design Approach II eliminates this requirement from the client.

**DESIGN APPROACH II**

The objective of this approach is to:

Provide the same advantage of allowing the client application to treat both the composite DirComponent and the individual FileComponent objects in a uniform manner while invoking the getComponentSize method
Figure 6.3  Class Association

Free the client application from having to check to make sure that the FileSystemComponent it is dealing with is an instance of the DirComponent class while invoking any of the composite-specific methods such as addComponent or getComponent.

In the new design (Figure 6.3), the composite-specific addComponent and getComponent methods are moved to the common interface FileSystemComponent. The FileSystemComponent provides the default implementation for these methods and is designed as an abstract class.

The default implementation of these methods consists of what is applicable to FileComponent objects. FileComponent objects are individual objects and do not contain other FileSystemComponent objects within. Hence, the default implementation does nothing and simply throws a custom CompositeException exception. The derived composite DirComponent class overrides these methods to provide custom implementation.

Because there is no change in the way the common getComponentSize method is designed, the client will still be able to treat both the composite DirComponent and FileComponent objects identically.

Because the common parent FileSystemComponent class now contains default implementations for the addComponent and the getComponent methods, the client application does not need to make any check before making a call to these composite-specific methods.
ITERATOR

DESCRIPTION

The Iterator pattern allows a client object to access the contents of a container in a sequential manner, without having any knowledge about the internal representation of its contents.

The term *container*, used above, can simply be defined as *a collection of data or objects*. The objects within the container could in turn be collections, making it a collection of collections. The Iterator pattern enables a client object to traverse through this collection of objects (or the container) without having the container to reveal how the data is stored internally.

To accomplish this, the Iterator pattern suggests that a *Container* object should be designed to provide a public interface in the form of an *Iterator* object for different client objects to access its contents. An *Iterator* object contains public methods to allow a client object to navigate through the list of objects within the container.

ITERATORS IN JAVA

One of the simplest iterators available in Java is the *java.sql.ResultSet* class, which is used to hold database records. This class offers a method *next()* for navigating along rows and a set of *getter* methods for column positioning.

Java also offers an interface *Enumeration* as part of the *java.util* package, which declares the methods listed in Table 7.1.

| Table 7.1  Enumeration Methods |
|-------------------------|-----------------|---------------------------------|
| Method                  | Return          | Description                     |
| hasMoreElements()       | boolean         | Checks if there are more elements in the collection |
| nextElement()           | Object          | Returns the next element in the collection |
Concrete iterators can be built as implementers of the Enumeration interface by providing implementation for its methods.

In addition, the java.util.Vector class offers a method:

    public final synchronized Enumeration elements()

that returns an enumeration of elements or objects. The returned Enumeration object works as an iterator for the Vector object. The Java Enumeration interface methods listed in Table 7.1 can be used on the returned Enumeration object to sequentially fetch elements stored in the Vector object.

Besides the Enumeration interface, Java also offers the java.util.Iterator interface. The Iterator interface declares three methods as in Table 7.2.

Similar to the Enumeration interface, concrete iterators can be built as implementers of the java.util.Iterator interface.

Though it is considered useful to employ existing Java iterator interfaces such as Iterator or Enumeration, it is not necessary to utilize one of these built-in Java interfaces to implement an iterator. One can design a custom iterator interface that is more suitable for an application need.

FILTERED ITERATORS

In the case of the java.util.Vector class, its iterator simply returns the next element in the collection. In addition to this basic behavior, an iterator may be implemented to do more than simply returning the next object in line. For instance, an iterator object can return a selected set of objects (instead of all objects) in a sequential order. This filtering can be based on some form of input from the client. These types of iterators are referred to as filtered iterators.

INTERNAL VERSUS EXTERNAL ITERATORS

An iterator can be designed either as an internal iterator or as an external iterator.

Internal iterators

- The collection itself offers methods to allow a client to visit different objects within the collection. For example, the java.util.ResultSet class contains the data and also offers methods such as next() to navigate through the item list.
- There can be only one iterator on a collection at any given time.
- The collection has to maintain or save the state of iteration.
External iterators
- The iteration functionality is separated from the collection and kept inside a different object referred to as an iterator. Usually, the collection itself returns an appropriate iterator object to the client depending on the client input. For example, the java.util.Vector class has its iterator defined in the form of a separate object of type Enumeration. This object is returned to a client object in response to the elements() method call.
- There can be multiple iterators on a given collection at any given time.
- The overhead involved in storing the state of iteration is not associated with the collection. It lies with the exclusive Iterator object.
Structural patterns primarily:

Deal with objects delegating responsibilities to other objects. This results in a layered architecture of components with low degree of coupling. Facilitate interobject communication when one object is not accessible to the other by normal means or when an object is not usable because of its incompatible interface. Provide ways to structure an aggregate object so that it is created in full and to reclaim system resources in a timely manner.
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Pattern Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>Decorator</td>
<td>Extends the functionality of an object in a manner that is transparent to its clients without using inheritance.</td>
</tr>
<tr>
<td>9</td>
<td>Adapter</td>
<td>Allows the conversion of the interface of a class to another interface that clients expect. This allows classes with incompatible interfaces to work together.</td>
</tr>
<tr>
<td>10</td>
<td>Chain of Responsibility</td>
<td>Avoids coupling a (request) sender object to a receiver object. Allows a sender object to pass its request along a chain of objects without knowing which object will actually handle the request.</td>
</tr>
<tr>
<td>11</td>
<td>Façade</td>
<td>Provides a higher-level interface to a subsystem of classes, making the subsystem easier to use.</td>
</tr>
<tr>
<td>12</td>
<td>Proxy</td>
<td>Allows a separate object to be used as a substitute to provide controlled access to an object that is not accessible by normal means.</td>
</tr>
<tr>
<td>13</td>
<td>Bridge</td>
<td>Allows the separation of an abstract interface from its implementation. This eliminates the dependency between the two, allowing them to be modified independently.</td>
</tr>
<tr>
<td></td>
<td>Virtual Proxy</td>
<td>Facilitates the mechanism for delaying the creation of an object until it is actually needed in a manner that is transparent to its client objects.</td>
</tr>
<tr>
<td></td>
<td>Counting Proxy</td>
<td>When there is a need to perform supplemental operations such as logging and counting before or after a method call on an object, recommends encapsulating the supplemental functionality into a separate object.</td>
</tr>
<tr>
<td></td>
<td>Aggregate Enforcer</td>
<td>Recommends that when an aggregate object is instantiated, all of its member variables representing the set of constituting objects must also be initialized. In other words, whenever an aggregate object is instantiated it must be constructed in full.</td>
</tr>
<tr>
<td></td>
<td>Explicit Object Release</td>
<td>Recommends that when an object goes out of scope, all of the system resources tied up with that object must be released in a timely manner.</td>
</tr>
<tr>
<td></td>
<td>Object Cache</td>
<td>Stores the results of a method call on an object in a repository. When client objects invoke the same method, instead of accessing the actual object, results are returned to the client object from the repository. This is done mainly to achieve a faster response time.</td>
</tr>
</tbody>
</table>
DECORATOR

DESCRIPTION

The Decorator Pattern is used to extend the functionality of an object dynamically without having to change the original class source or using inheritance. This is accomplished by creating an object wrapper referred to as a Decorator around the actual object.

CHARACTERISTICS OF A DECORATOR

The Decorator object is designed to have the same interface as the underlying object. This allows a client object to interact with the Decorator object in exactly the same manner as it would with the underlying actual object.

The Decorator object contains a reference to the actual object.

The Decorator object receives all requests (calls) from a client. It in turn forwards these calls to the underlying object.

The Decorator object adds some additional functionality before or after forwarding requests to the underlying object. This ensures that the additional functionality can be added to a given object externally at runtime without modifying its structure.

Typically, in object-oriented design, the functionality of a given class is extended using inheritance. Table 8.1 lists the differences between the Decorator pattern and inheritance.

EXAMPLE

Let us revisit the message logging utility we built while discussing the Factory Method and the Singleton patterns earlier. Our design mainly comprised a Logger interface and two of its implementers — FileLogger and ConsoleLogger — to log messages to a file and to the screen, respectively. In addition, we had the LoggerFactory class with a factory method in it.
Table 8.1 Decorator Pattern versus Inheritance

<table>
<thead>
<tr>
<th>Decorator Pattern</th>
<th>Inheritance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Used to extend the functionality of a particular object.</td>
<td>Used to extend the functionality of a class of objects.</td>
</tr>
<tr>
<td>Does not require subclassing.</td>
<td>Requires subclassing.</td>
</tr>
<tr>
<td>Dynamic.</td>
<td>Static.</td>
</tr>
<tr>
<td>Runtime assignment of responsibilities.</td>
<td>Compile time assignment of responsibilities.</td>
</tr>
<tr>
<td>Prevents the proliferation of subclasses leading to less complexity and confusion.</td>
<td>Could lead to numerous subclasses, exploding class hierarchy on specific occasions.</td>
</tr>
<tr>
<td>More flexible.</td>
<td>Less flexible.</td>
</tr>
<tr>
<td>Possible to have different decorator objects for a given object simultaneously. A client can choose what capabilities it wants by sending messages to an appropriate decorator.</td>
<td>Having subclasses for all possible combinations of additional capabilities, which clients expect out of a given class, could lead to a proliferation of subclasses.</td>
</tr>
<tr>
<td>Easy to add any combination of capabilities. The same capability can even be added twice.</td>
<td>Difficult.</td>
</tr>
</tbody>
</table>

The LoggerFactory is not shown in Figure 8.1. This is because it is not directly related to the current example discussion.

Let us suppose that some of the clients are now in need of logging messages in new ways beyond what is offered by the message logging utility. Let us consider the following two small features that clients would like to have:

Transform an incoming message to an HTML document.
Apply a simple encryption by transposition logic on an incoming message.

```
<<interface>>
Logger
log(msg:String)
```

```
ConsoleLogger
log(msg:String)
```

```
FileLogger
log(msg:String)
```

Figure 8.1 Logging Utility Class Hierarchy
### Table 8.2  Subclasses of **FileLogger** and **ConsoleLogger**

<table>
<thead>
<tr>
<th>Subclass</th>
<th>Parent Class</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTMLFileLogger</td>
<td>FileLogger</td>
<td>Transform an incoming message to an HTML document and store it in a log file.</td>
</tr>
<tr>
<td>HTMLConsLogger</td>
<td>ConsoleLogger</td>
<td>Transform an incoming message to an HTML document and display it on the screen.</td>
</tr>
<tr>
<td>EncFileLogger</td>
<td>FileLogger</td>
<td>Apply encryption on an incoming message and store it in a log file.</td>
</tr>
<tr>
<td>EncConsLogger</td>
<td>ConsoleLogger</td>
<td>Apply encryption on an incoming message and display it on the screen.</td>
</tr>
</tbody>
</table>

Typically, in object-oriented design, without changing the code of an existing class, new functionality can be added by applying inheritance, i.e., by subclassing an existing class and overriding its methods to add the required new functionality. Applying inheritance, we would subclass both the `FileLogger` and the `ConsoleLogger` classes to add the new functionality with the following set of new subclasses (Table 8.2).

As can be seen from the class diagram in Figure 8.2, a set of four new subclasses are added in order to add the new functionality. If we had additional `Logger` types (for example a `DBLogger` to log messages to a database), it would lead to more subclasses. With every new feature that needs to be added, there will be a multiplicative growth in the number of subclasses and soon we will have an exploding class hierarchy.

---

**Figure 8.2** The Resulting Class Hierarchy after Applying Inheritance to Add the New Functionality
The Decorator pattern comes to our rescue in situations like this. The Decorator pattern recommends having a wrapper around an object to extend its functionality by object composition rather than by inheritance.

Applying the Decorator pattern, let us define a default root decorator LoggerDecorator (Listing 8.1) for the message logging utility with the following characteristics:

The LoggerDecorator contains a reference to a Logger instance. This reference points to a Logger object it wraps.
The LoggerDecorator implements the Logger interface and provides the basic default implementation for the log method, where it simply forwards an incoming call to the Logger object it wraps. Every subclass of the LoggerDecorator is hence guaranteed to have the log method defined in it.

It is important for every logger decorator to have the log method because a decorator object must provide the same interface as the object it wraps. When clients create an instance of the decorator, they interact with the decorator in exactly the same manner as they would with the original object using the same interface.

Let us define two subclasses, HTMLLogger and EncryptLogger, of the default LoggerDecorator as shown in Figure 8.3.

**CONCRETE LOGGER DECORATORS**

**HTMLLogger**
The HTMLLogger (Listing 8.2) overrides the default implementation of the log method. Inside the log method, this decorator transforms an incoming message to an HTML document and then sends it to the Logger instance it contains for logging.
EncryptLogger

Similar to the HTMLLogger, the EncryptLogger (Listing 8.3) overrides the log method. Inside the log method, the EncryptLogger implements simple encryption logic by shifting characters to the right by one position and sends it to the Logger instance it contains for logging.

The class diagram in Figure 8.4 shows how different classes are arranged while applying the Decorator pattern.

In order to log messages using the newly designed decorators a client object (Listing 8.4) needs to:

- Create an appropriate Logger instance (FileLogger/ConsoleLogger) using the LoggerFactory factory method.
- Create an appropriate LoggerDecorator instance by passing the Logger instance created in Step 1 as an argument to its constructor.
- Invoke methods on the LoggerDecorator instance as it would on the Logger instance.

Figure 8.5 shows the message flow when a client object uses the HTMLLogger object to log messages.
Listing 8.2  HTMLLogger Class

```java
public class HTMLLogger extends LoggerDecorator {
    public HTMLLogger(Logger inp_logger) {
        super(inp_logger);
    }

    public void log(String DataLine) {
        /*
         * Added functionality
         */
        DataLine = makeHTML(DataLine);
        /*
         * Now forward the encrypted text to the FileLogger
         * for storage
         */
        logger.log(DataLine);
    }

    public String makeHTML(String DataLine) {
        /*
         * Make it into an HTML document.
         */
        DataLine = "<HTML><BODY>" + "<b>" + DataLine + "</b>" + "</BODY></HTML>";
        return DataLine;
    }
}
```

ADDING A NEW MESSAGE LOGGER

In case of the message logging utility, applying the Decorator pattern does not lead to a large number of subclasses with a fast growing class hierarchy as it would if we apply inheritance. Let us say that we have another Logger type, say a DBLogger, that logs messages to a database. In order to apply the HTML transformation or to apply the encryption before logging to the database, all that a client object needs to do is to follow the list of steps mentioned earlier. Because the DBLogger would be of the Logger type, it can be sent to any of the HTMLLogger or the EncryptLogger classes as an argument while invoking their constructors.
Listing 8.3  EncryptLogger Class

```java
public class EncryptLogger extends LoggerDecorator {
    public EncryptLogger(Logger inp_logger) {
        super(inp_logger);
    }
    public void log(String DataLine) {
        /*
         * Added functionality
         */
        DataLine = encrypt(DataLine);
        /*
         * Now forward the encrypted text to the FileLogger for storage
         */
        logger.log(DataLine);
    }
    public String encrypt(String DataLine) {
        /*
         * Apply simple encryption by Transposition...
         * Shift all characters by one position.
         */
        DataLine = DataLine.substring(DataLine.length() - 1) +
                   DataLine.substring(0, DataLine.length() - 1);
        return DataLine;
    }
}
```//end of class

**ADDING A NEW DECORATOR**

From the example it can be observed that a LoggerDecorator instance contains a reference to an object of type Logger. It forwards requests to this Logger object before or after adding the new functionality. Since the base LoggerDecorator class implements the Logger interface, an instance of LoggerDecorator or any of its subclasses can be treated as of the Logger type. Hence a LoggerDecorator can contain an instance of any of its subclasses and forward calls to it. In general, a decorator object can contain another decorator object and can forward calls to it. In this way, new decorators, and hence new functionality, can be built by wrapping an existing decorator object.
Figure 8.4  Association between Different Logger Classes and Logger Decorators

Listing 8.4  Client DecoratorClient Class

class DecoratorClient {
    public static void main(String[] args) {
        LoggerFactory factory = new LoggerFactory();
        Logger logger = factory.getLogger();
        HTMLLogger hLogger = new HTMLLogger(logger);
        //the decorator object provides the same interface.
        hLogger.log("A Message to Log");
        EncryptLogger eLogger = new EncryptLogger(logger);
        eLogger.log("A Message to Log");
    }
}  //End of class
Figure 8.5  Message Flow When a Client Uses the HTMLLogger (Decorator) to Log a Message

PRACTICE QUESTIONS

1. Create a FileReader utility class with a method to read lines from a file.
2. The EncryptLogger in the example application encrypts a given text by shifting characters to the right by one position. Create a Decorator DecryptFileReader for the FileReader to add the decryption functionality, after reading data from a file.
3. Enhance DecoratorClient class to do the following:
   - Write a message to a file using the EncryptLogger.
   - Read using the DecryptFileReader decorator to display the message in an unencrypted form.
ADAPTER

DESCRIPTION

In general, clients of a class access the services offered by the class through its interface. Sometimes, an existing class may provide the functionality required by a client, but its interface may not be what the client expects. This could happen due to various reasons such as the existing interface may be too detailed, or it may lack in detail, or the terminology used by the interface may be different from what the client is looking for.

In such cases, the existing interface needs to be converted into another interface, which the client expects, preserving the reusability of the existing class. Without such conversion, the client will not be able to use the functionality offered by the class. This can be accomplished by using the Adapter pattern. The Adapter pattern suggests defining a wrapper class around the object with the incompatible interface. This wrapper object is referred as an adapter and the object it wraps is referred to as an adaptee. The adapter provides the required interface expected by the client. The implementation of the adapter interface converts client requests into calls to the adaptee class interface. In other words, when a client calls an adapter method, internally the adapter class calls a method of the adaptee class, which the client has no knowledge of. This gives the client indirect access to the adaptee class. Thus, an adapter can be used to make classes work together that could not otherwise because of incompatible interfaces.

The term interface used in the discussion above:

- Does not refer to the concept of an interface in Java programming language, though a class’s interface may be declared using a Java interface.
- Does not refer to the user interface of a typical GUI application consisting of windows and GUI controls.
- Does refer to the programming interface that a class exposes, which is meant to be used by other classes. As an example, when a class is designed as an abstract class or a Java interface, the set of methods declared in it makes up the class’s interface.
CLASS ADAPTERS VERSUS OBJECT ADAPTERS

Adapters can be classified broadly into two categories — class adapters and object adapters — based on the way a given adapter is designed.

Class Adapter

A class adapter is designed by subclassing the adaptee class. In addition, a class adapter implements the interface expected by the client object. When a client object invokes a class adapter method, the adapter internally calls an adaptee method that it inherited.

Object Adapter

An object adapter contains a reference to an adaptee object. Similar to a class adapter, an object adapter also implements the interface, which the client expects. When a client object calls an object adapter method, the object adapter invokes an appropriate method on the adaptee instance whose reference it contains. Table 9.1 lists the differences between class and object adapters in detail.
<table>
<thead>
<tr>
<th>Table 9.1  Class Adapters versus Object Adapters</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class Adapters</strong></td>
</tr>
<tr>
<td>Based on the concept of inheritance.</td>
</tr>
<tr>
<td>Can be used to adapt the interface of the adaptee only. Cannot adapt the interfaces of its subclasses, as the adapter is statically linked with the adaptee when it is created.</td>
</tr>
<tr>
<td>Because the adapter is designed as a subclass of the adaptee, it is possible to override some of the adaptee's behavior. <strong>Note:</strong> In Java, a subclass cannot override a method that is declared as final in its parent class.</td>
</tr>
<tr>
<td>The client will have some knowledge of the adaptee's interface as the full public interface of the adaptee is visible to the client.</td>
</tr>
<tr>
<td>In Java applications: Suitable when the expected interface is available in the form of a Java interface and not as an abstract or concrete class. This is because the Java programming language allows only single inheritance. Since a class adapter is designed as a subclass of the adaptee class, it will not be able to subclass the interface class (representing the expected interface) also, if the expected interface is available in the form of an abstract or concrete class.</td>
</tr>
<tr>
<td>In Java applications: Can adapt methods with protected access specifier.</td>
</tr>
</tbody>
</table>
The Chain of Responsibility pattern (CoR) recommends a low degree of coupling between an object that sends out a request and the set of potential request handler objects.

When there is more than one object that can handle or fulfill a client request, the CoR pattern recommends giving each of these objects a chance to process the request in some sequential order. Applying the CoR pattern in such a case, each of these potential handler objects can be arranged in the form of a chain, with each object having a pointer to the next object in the chain. The first object in the chain receives the request and decides either to handle the request or to pass it on to the next object in the chain. The request flows through all objects in the chain one after the other until the request is handled by one of the handlers in the chain or the request reaches the end of the chain without getting processed.

As an example, if A ⋈ B ⋈ C are objects capable of handling the request, in this order, then A should handle the request or pass on to B without determining whether B can fulfill the request. Upon receiving the request, B should either handle it or pass on to C. When C receives the request, it should either handle the request or the request falls off the chain without getting processed. In other words, a request submitted to the chain of handlers may not be fulfilled even after reaching the end of the chain.

The following are some of the important characteristics of the CoR pattern:
- The set of potential request handler objects and the order in which these objects form the chain can be decided dynamically at runtime by the client depending on the current state of the application.
- A client can have different sets of handler objects for different types of requests depending on its current state. Also, a given handler object may need to pass on an incoming request to different other handler objects depending on the request type and the state of the client application. For these communications to be simple, all potential handler objects should provide a consistent interface. In Java this can be accomplished by having
different handlers implement a common interface or be subclasses of a common abstract parent class.
The client object that initiates the request or any of the potential handler objects that forward the request do not have to know about the capabilities of the object receiving the request. This means that neither the client object nor any of the handler objects in the chain need to know which object will actually fulfill the request.
Request handling is not guaranteed. This means that the request may reach the end of the chain without being fulfilled. The following example presents a scenario where a purchase request submitted to a chain of handlers is not approved even after reaching the end of the chain.
The Façade pattern deals with a subsystem of classes. A subsystem is a set of classes that work in conjunction with each other for the purpose of providing a set of related features (functionality). For example, an Account class, Address class and CreditCard class working together, as part of a subsystem, provide features of an online customer.

In real-world applications, a subsystem could consist of a large number of classes. Clients of a subsystem may need to interact with a number of subsystem classes for their needs. This kind of direct interaction of clients with subsystem classes leads to a high degree of coupling between the client objects and the subsystem (Figure 11.1). Whenever a subsystem class undergoes a change, such as a change in its interface, all of its dependent client classes may get affected.

The Façade pattern is useful in such situations. The Façade pattern provides a higher level, simplified interface for a subsystem resulting in reduced complexity and dependency. This in turn makes the subsystem usage easier and more manageable.

A façade is a class that provides this simplified interface for a subsystem to be used by clients. With a Façade object in place, clients interact with the Façade object instead of interacting directly with subsystem classes. The Façade object takes up the responsibility of interacting with the subsystem classes. In effect, clients interface with the façade to deal with the subsystem. Thus the Façade pattern promotes a weak coupling between a subsystem and its clients (Figure 11.2).

From Figure 11.2, we can see that the Façade object decouples and shields clients from subsystem objects. When a subsystem class undergoes a change, clients do not get affected as before.

Even though clients use the simplified interface provided by the façade, when needed, a client will be able to access subsystem components directly through the lower level interfaces of the subsystem as if the Façade object does not exist. In this case, they will still have the same dependency/coupling issue as earlier.
Figure 11.1  Client Interaction with Subsystem Classes before Applying the Façade Pattern

Figure 11.2  Client Interaction with Subsystem Classes after Applying the Façade Pattern
DESCRIPTION

Let us consider the following code sample:

```java
//Client
class Customer{
    public void someMethod(){
        //Create the Service Provider Instance
        FileUtil futilObj=new FileUtil();
        //Access the Service
        futilObj.writeToFile("Some Data");
    }
}
```

As part of its implementation, the `Customer` class creates an instance of the `FileUtil` class and directly accesses its services. In other words, for a client object, the way of accessing a `FileUtil` object is fairly straightforward. From the implementation it seems to be the most commonly used way for a client object to access a service provider object. In contrast, sometimes a client object may not be able to access a service provider object (also referred to as a target object) by normal means. This could happen for a variety of reasons depending on:

- **The location of the target object** — The target object may be present in a different address space in the same or a different computer.
- **The state of existence of the target object** — The target object may not exist until it is actually needed to render a service or the object may be in a compressed form.
- **Special Behavior** — The target object may offer or deny services based on the access privileges of its client objects. Some service provider objects may need special consideration when used in a multithreaded environment.
In such cases, instead of having client objects to deal with the special requirements for accessing the target object, the Proxy pattern suggests using a separate object referred to as a proxy to provide a means for different client objects to access the target object in a normal, straightforward manner.

The Proxy object offers the same interface as the target object. The Proxy object interacts with the target object on behalf of a client object and takes care of the specific details of communicating with the target object. As a result, client objects are no longer needed to deal with the special requirements for accessing the services of the target object. A client can call the Proxy object through its interface and the Proxy object in turn forwards those calls to the target object. Client objects need not even know that they are dealing with Proxy for the original object. The Proxy object hides the fact that a client object is dealing with an object that is either remote, unknown whether instantiated or not, or needs special authentication. In other words, a Proxy object serves as a transparent bridge between the client and an inaccessible remote object or an object whose instantiation may have been deferred.

Proxy objects are used in different scenarios leading to different types of proxies. Let us take a quick look at some of the proxies and their purpose.

**Note:** Table 12.1 lists different types of Proxy objects. In this chapter, only the remote proxy is discussed in detail. Some of the other proxy types are discussed as separate patterns later in this book.

**PROXY VERSUS OTHER PATTERNS**

From the discussion of different Proxy objects, it can be observed that there are two main characteristics of a Proxy object:

- It is an intermediary between a client object and the target object.
- It receives calls from a client object and forwards them to the target object.

In this context, it looks very similar to some of the other patterns discussed earlier in this book. Let us see in detail the similarities and differences between the Proxy pattern and some of the other similar patterns.

**Proxy versus Decorator**

- **Proxy**
  - The client object cannot access the target object directly.
  - A proxy object provides access control to the target object (in the case of the protection proxy).
  - A proxy object does not add any additional functionality.

- **Decorator**
  - The client object does have the ability to access the target object directly, if needed.
  - A Decorator object does not control access to the target object.
  - A Decorator adds additional functionality to an object.
## Table 12.1 List of Different Proxy Types

<table>
<thead>
<tr>
<th>Proxy Type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remote Proxy</td>
<td>To provide access to an object located in a different address space.</td>
</tr>
<tr>
<td>Virtual Proxy</td>
<td>To provide the required functionality to allow the on-demand creation of a memory intensive object (until required).</td>
</tr>
<tr>
<td>Cache Proxy/Server Proxy</td>
<td>To provide the functionality required to store the results of most frequently used target operations. The proxy object stores these results in some kind of a repository. When a client object requests the same operation, the proxy returns the operation results from the storage area without actually accessing the target object.</td>
</tr>
<tr>
<td>Firewall Proxy</td>
<td>The primary use of a firewall proxy is to protect target objects from bad clients. A firewall proxy can also be used to provide the functionality required to prevent clients from accessing harmful targets.</td>
</tr>
<tr>
<td>Protection Proxy</td>
<td>To provide the functionality required for allowing different clients to access the target object at different levels. Subsequently, those permissions are used to restrict access to specific parts of the proxy (in turn of the target object). A client object is not allowed to access a particular method if it does not have a specific right to execute the method.</td>
</tr>
<tr>
<td>Synchronization Proxy</td>
<td>To provide the required functionality to allow safe concurrent accesses to a target object by different client objects.</td>
</tr>
<tr>
<td>Smart Reference Proxy</td>
<td>To provide the functionality to prevent the accidental disposal/deletion of the target object when there are clients currently with references to it. To accomplish this, the proxy keeps a count of the number of references to the target object. The proxy deletes the target object if and when there are no references to it.</td>
</tr>
<tr>
<td>Counting Proxy</td>
<td>To provide some kind of audit mechanism before executing a method on the target object.</td>
</tr>
</tbody>
</table>

### Proxy versus Façade

- A Proxy object represents a single object.
- The client object cannot access the target object directly.
- A Proxy object provides access control to the single target object.
Façade
- A Façade object represents a subsystem of objects.
- The client object does have the ability to access the subsystem objects directly, if needed.
- A Façade object provides a simplified higher level interface to a subsystem of components.

Proxy versus Chain of Responsibility

Proxy
- A Proxy object represents a single object.
- Client requests are first received by the Proxy object, but are never processed directly by the Proxy object.
- Client requests are always forwarded to the target object.
- Response to the request is guaranteed, provided the communication between the client and the server locations is working.

Chain of Responsibility
- Chain can contain many objects.
- The object that receives the client request first could process the request.
- Client requests are forwarded to the next object in the chain only if the current receiver cannot process the request.
- Response to the request is not guaranteed. It means that the request may end up reaching the end of the chain and still might not be processed.

In Java, the concept of Remote Method Invocation (RMI) makes extensive use of the Remote Proxy pattern. Let us take a quick look at the concept of RMI and different components that facilitate the RMI communication process.

RMI: A QUICK OVERVIEW

RMI enables a client object to access remote objects and invoke methods on them as if they are local objects (Figure 12.1).

RMI Components

The following different components working together provide the stated RMI functionality:

Remote Interface — A remote object must implement a remote interface (one that extends java.rmi.Remote). A remote interface declares the

![Figure 12.1 Client's View of Its Communication with a Remote Object Using RMI](image-url)
methods in the remote object that can be accessed by its clients. In other words, the remote interface can be seen as the client's view of the remote object.

Requirements:
- Extend the `java.rmi.Remote` interface.
- All methods in the remote interface must be declared to throw `java.rmi.RemoteException` exception.

Remote Object — A remote object is responsible for implementing the methods declared in the associated remote interface.

Requirements:
- Must provide implementation for a remote interface.
- Must extend `java.rmi.server.UnicastRemoteObject`.
- Must have a constructor with no arguments.
- Must be associated with a server. The server creates an instance of the remote object by invoking its zero argument constructor.

RMI Registry — RMI registry provides the storage area for holding different remote objects.
- A remote object needs to be stored in the RMI registry along with a name reference to it for a client object to be able to access it.
- Only one object can be stored with a given name reference.

Client — Client is an application object attempting to use the remote object.
- Must be aware of the interface implemented by the remote object.
- Can search for a remote object using a name reference in the RMI Registry. Once the remote object reference is found, it can invoke methods on this object reference.

RMIC: Java RMI Stub Compiler — Once a remote object is compiled successfully, RMIC, the Java RMI stub compiler can be used to generate stub and skeleton class files for the remote object. Stub and skeleton classes are generated from the compiled remote object class. These stub and skeleton classes make it possible for a client object to access the remote object in a seamless manner.

The following section describes how the actual communication takes place between a client and a remote object.

RMI Communication Mechanism

In general, a client object cannot directly access a remote object by normal means. In order to make it possible for a client object to access the services of a remote object as if it is a local object, the RMIC-generated stub of the remote object class and the remote interface need to be copied to the client computer.

The stub acts as a (Remote) proxy for the remote object and is responsible for forwarding method invocations on the remote object to the server where the actual remote object implementation resides. Whenever a client references the remote object, the reference is, in fact, made to a local stub. That means, when a client makes a method call on the remote object, it is first received by the local stub instance. The stub forwards this call to the remote server. On the server the RMIC generated skeleton of the remote object receives this call.
The skeleton is a server side object and it does not need to be copied to the client computer. The skeleton is responsible for dispatching calls to the actual remote object implementation. Once the remote object executes the method, results are sent back to the client in the reverse direction.

Figure 12.2 shows the actual RMI communication process.

For more information on the Java RMI technology, I recommend reading the RMI tutorial at java.sun.com.

**RMI AND PROXY PATTERN**

It can be seen from the RMI communication discussion that the stub class, acting as a remote proxy for the remote object, makes it possible for a client to treat a remote object as if it is available locally. Thus, any application that uses RMI contains an implicit implementation of the Proxy pattern.
The Bridge pattern promotes the separation of an abstraction's interface from its implementation. In general, the term *abstraction* refers to the process of identifying the set of attributes and behavior of an object that is specific to a particular usage. This specific view of an object can be designed as a separate object omitting irrelevant attributes and behavior. The resulting object itself can be referred to as an *abstraction*. Note that a given object can have more than one associated abstraction, each with a distinct usage.

A given abstraction may have one or more implementations for its methods (behavior). In terms of implementation, an abstraction can be designed as an interface with one or more concrete implementers (Figure 13.1).

![Figure 13.1 Abstraction as an Interface with a Set of Concrete Implementers](image-url)
In the class hierarchy shown in Figure 13.1, the Abstraction interface declares a set of methods that represent the result of abstracting common features from different objects. Both Implementer_1 and Implementer_2 represent the set of Abstraction implementers. This approach suffers from the following two limitations:

1. When there is a need to subclass the hierarchy for some other reason, it could lead to an exponential number of subclasses and soon we will have an exploding class hierarchy.
2. Both the abstraction interface and its implementation are closely tied together and hence they cannot be independently varied without affecting each other.

Using the Bridge pattern, a more efficient and manageable design of an abstraction can be achieved. The design of an abstraction using the Bridge pattern separates its interfaces from implementations. Applying the Bridge pattern, both the interfaces and the implementations of an abstraction can be put into separate class hierarchies as in Figure 13.2.

From the class diagram in Figure 13.2, it can be seen that the Abstraction maintains an object reference of the Implementer type. A client application can

---

**Figure 13.2 Interface and Implementations in Two Separate Class Hierarchies**
choose a desired abstraction type from the Abstraction class hierarchy. The abstraction object can then be configured with an instance of an appropriate implementer from the Implementer class hierarchy. This ability to combine abstractions and implementations dynamically can be very useful in terms of extending the functionality without subclassing. When a client object invokes a method on the Abstraction object, it forwards the call to the Implementer object it contains. The Abstraction object may offer some amount of processing before forwarding the call to the Implementer object.

This type of class arrangement completely decouples the interface and the implementation of an abstraction and allows the classes in the interface and the implementation hierarchy to vary without affecting each other.

**BRIDGE PATTERN VERSUS ADAPTER PATTERN**

**Similarities:**
Both the Adapter pattern and the Bridge pattern are similar in that they both work towards concealing the details of the underlying implementation from the client.

**Differences:**
The Adapter pattern aims at making classes work together that could not otherwise because of incompatible interfaces. An Adapter is meant to change the interface of an existing object. As we have seen during our discussion on the Adapter pattern, an Adapter requires an (existing) adaptee class, indicating that the Adapter pattern is more suitable for needs after the initial system design.

The Bridge pattern is more of a design time pattern. It is used when the designer has control over the classes in the system. It is applied before a system has been implemented to allow both abstraction interfaces and its implementations to be varied independently without affecting each other. In the context of the Bridge pattern, the issue of incompatible interfaces does not exist. Client objects always use the interface exposed by the abstraction interface classes. Thus both the Bridge pattern and the Adapter pattern are used to solve different design issues.
IV

BEHAVIORAL PATTERNS

Behavioral Patterns mainly:

- Deal with the details of assigning responsibilities between different objects
- Describe the communication mechanism between objects
- Define the mechanism for choosing different algorithms by different objects at runtime
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Pattern Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>Command</td>
<td>Allows a request to be encapsulated into an object giving control over request queuing, sequencing and undoing.</td>
</tr>
<tr>
<td>15</td>
<td>Mediator</td>
<td>Encapsulates the direct object-to-object communication details among a set of objects in a separate (mediator) object. This eliminates the need for these objects to interact with each other directly.</td>
</tr>
<tr>
<td>16</td>
<td>Memento</td>
<td>Allows the state of an object to be captured and stored. The object can be put back to this (previous) state, when needed.</td>
</tr>
<tr>
<td>17</td>
<td>Observer</td>
<td>Promotes a publisher–subscriber communication model when there is a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified so they can update their state.</td>
</tr>
<tr>
<td>18</td>
<td>Interpreter</td>
<td>Useful when the objective is to provide a client program or a user the ability to specify operations in a simple language. Helps in interpreting operations specified using a language, using its grammar. More suitable for languages with simple grammar.</td>
</tr>
<tr>
<td>19</td>
<td>State</td>
<td>Allows the state-specific behavior of an object to be encapsulated in the form of a set of state objects. With each state-specific behavior mapped onto a specific state object, the object can change its behavior by configuring itself with an appropriate state object.</td>
</tr>
<tr>
<td>20</td>
<td>Strategy</td>
<td>Allows each of a family of related algorithms to be encapsulated into a set of different subclasses (strategy objects) of a common superclass. For an object to use an algorithm, the object needs to be configured with the corresponding strategy object. With this arrangement, algorithm implementation can vary without affecting its clients.</td>
</tr>
<tr>
<td>21</td>
<td>Null Object</td>
<td>Provides a way of encapsulating the (usually do nothing) behavior of a given object type into a separate null object. This object can be used to provide the default behavior when no object of the specific type is available.</td>
</tr>
</tbody>
</table>

(continued)
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Pattern Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Template Method</strong></td>
<td>When there is an algorithm that could be implemented in multiple ways, the template pattern enables keeping the outline of the algorithm in a separate method (Template Method) inside a class (Template Class), leaving out the specific implementations of this algorithm to different subclasses. In other words, the Template Method pattern is used to keep the invariant part of the functionality in one place and allow the subclasses to provide the implementation of the variant part.</td>
</tr>
<tr>
<td></td>
<td><strong>Object Authenticator</strong></td>
<td>Useful when access to an application object is restricted and requires a client object to furnish proper authentication credentials. Uses a separate object with the responsibility of verifying the access privileges of different client objects instead of keeping this responsibility on the application object.</td>
</tr>
<tr>
<td></td>
<td><strong>Common Attribute Registry</strong></td>
<td>Provides a way of designing a repository to store the common transient state of an application.</td>
</tr>
</tbody>
</table>
In general, an object-oriented application consists of a set of interacting objects each offering limited, focused functionality. In response to user interaction, the application carries out some kind of processing. For this purpose, the application makes use of the services of different objects for the processing requirement. In terms of implementation, the application may depend on a designated object that invokes methods on these objects by passing the required data as arguments (Figure 14.1). This designated object can be referred to as an *invoker* as it invokes operations on different objects. The invoker may be treated as part of the client application. The set of objects that actually contain the implementation to offer the services required for the request processing can be referred to as *Receiver* objects.

![Diagram of object interaction](image)

**Figure 14.1 Object Interaction: Before Applying the Command Pattern**

In this design, the application that forwards the request and the set of Receiver objects that offer the services required to process the request are closely tied to each other in that they interact with each other directly. This could result in a set of conditional `if` statements in the implementation of the invoker.

```java
... 
if (RequestType=TypeA){
   //do something
}
if (RequestType=TypeB){
   //do something
}
... 
```
When a new type of feature is to be added to the application, the existing code needs to be modified and it violates the basic object-oriented open-closed principle.

```java
... 
if (RequestType==TypeA)
    //do something
...
... 
if (RequestType==NewType)
    //do something
...
```

The open-closed principle states that a software module should be:

- **Open for extension** — It should be possible to alter the behavior of a module or add new features to the module functionality.
- **Closed for modification** — Such a module should not allow its code to be modified.

In a nutshell, the open-closed principle helps in designing software modules whose functionality can be extended without having to modify the existing code.

Using the Command pattern, the invoker that issues a request on behalf of the client and the set of service-rendering Receiver objects can be decoupled. The Command pattern suggests creating an abstraction for the processing to be carried out or the action to be taken in response to client requests.

This abstraction can be designed to declare a common interface to be implemented by different concrete implementers referred to as Command objects. Each Command object represents a different type of client request and the corresponding processing. In Figure 14.2, the Command interface represents the abstraction. It declares an execute method, which is implemented by two of its implementer (command) classes — ConcreteCommand_1 and ConcreteCommand_2.

A given Command object is responsible for offering the functionality required to process the request it represents, but it does not contain the actual implementation of the functionality. Command objects make use of Receiver objects in offering this functionality (Figure 14.3).

When the client application needs to offer a service in response to user (or other application) interaction:

1. It creates the necessary Receiver objects.
2. It creates an appropriate Command object and configures it with the Receiver objects created in Step 1.
3. It creates an instance of the invoker and configures it with the Command object created in Step 2.
4. The invoker invokes the execute() method on the Command object.
5. As part of its implementation of the execute method, a typical Command object invokes necessary methods on the Receiver objects it contains to provide the required service to its caller.
In the new design:

The client/invoker does not directly interact with Receiver objects and therefore, they are completely decoupled from each other.

When the application needs to offer a new feature, a new Command object can be added. This does not require any changes to the code of the invoker. Hence the new design conforms to the open-closed principle. Because the request is designed in the form of an object, it opens up a whole new set of possibilities such as:

- Storing a Command object to persistent media:
  - To be executed later.
  - To apply reverse processing to support the undo feature.
- Grouping together different Command objects to be executed as a single unit.
Example

Let us build an application to manage items in a library item database. Typical library items include books, CDs, videos and DVDs. These items are grouped into categories and a given item can belong to one or more categories. For example, a new movie video may belong to both the Video category and the New Releases category.

Let us define two classes — Item and Category — (Listing 14.3) representing a typical library item and a category of items, respectively (Figure 14.6).

From the design and the implementation of the Item and the Category classes, it can be seen that a Category object maintains a list of its current member items. Similarly, an Item object maintains the list of all categories which it is part of. For simplicity, let us suppose that the library item management application deals only with adding and deleting items. Applying the Command pattern, the action to be taken to process add item and delete item requests can be designed as implementers of a common Command Interface interface. The Command Interface provides an abstraction for the processing to be carried out in response to a typical library item management request such as add or delete item. The Command Interface implementers — AddCommand and DeleteCommand — in Figure 14.7 represent the add and the delete item request, respectively.

Let us further define an invoker ItemManager class.

```java
public class ItemManager {
    CommandInterface command;
    public void setCommand(CommandInterface c) {
        command = c;
    }
    public void process() {
        command.execute();
    }
}
```

The ItemManager:

- Contains a Command object within
- Invokes the Command object’s execute method as part of its process method implementation
- Provides a setCommand method to allow client objects to configure it with a Command object

The client CommandTest uses the invoker ItemManager to get its add item and delete item requests processed.

Application Flow

To add or delete an item, the client CommandTest (Listing 14.4):
1. Creates the necessary Item and Category objects. These objects act as receivers.
2. Creates an appropriate Command object that corresponds to its current request. The set of Receiver objects created in Step 1 is passed to the Command object at the time of its creation.
3. Creates an instance of the ItemManager and configures it with the Command object created in Step 2.
4. Invokes the process() method of the ItemManager. The ItemManager invokes the execute method on the Command object. The Command object in turn invokes necessary Receiver object methods. Different Item and Category Receiver objects perform the actual request processing. To keep the example simple, no database access logic is implemented. Both Item and Category objects are implemented to simply display a message.

When the client program is run, the following output is displayed:

Item 'A Beautiful Mind' has been added to the 'CD' Category
Item 'Duet' has been added to the 'CD' Category
Item 'Duet' has been added to the 'New Releases' Category
Item 'Duet' has been deleted from the 'New Releases' Category

The class diagram in Figure 14.8 depicts the overall class association. The sequence diagram in Figure 14.9 shows the message flow when the client CommandTest uses a Command object to add an item.
public class Item {
    private HashMap categories;
    private String desc;
    public Item(String s) {
        desc = s;
        categories = new HashMap();
    }
    public String getDesc() {
        return desc;
    }
    public void add(Category cat) {
        categories.put(cat.getDesc(), cat);
    }
    public void delete(Category cat) {
        categories.remove(cat.getDesc());
    }
}

public class Category {
    private HashMap items;
    private String desc;
    public Category(String s) {
        desc = s;
        items = new HashMap();
    }
    public String getDesc() {
        return desc;
    }
    public void add(Item i) {
        items.put(i.getDesc(), i);
        System.out.println("Item "+ i.getDesc() + "' has been added to the "+ getDesc() + "' Category ");
    }
    public void delete(Item i) {
        items.remove(i.getDesc());
        System.out.println("Item "+ i.getDesc() + "' has been deleted from the "+ getDesc() + "' Category ");
    }
}
Figure 14.6 Item-Category Association

```
Item
<table>
<thead>
<tr>
<th>categories: HashMap</th>
</tr>
</thead>
<tbody>
<tr>
<td>desc: String</td>
</tr>
<tr>
<td>getDesc(): String</td>
</tr>
<tr>
<td>add(cat: Category)</td>
</tr>
<tr>
<td>delete(cat: Category)</td>
</tr>
</tbody>
</table>

Category
<table>
<thead>
<tr>
<th>items: HashMap</th>
</tr>
</thead>
<tbody>
<tr>
<td>desc: String</td>
</tr>
<tr>
<td>getDesc(): String</td>
</tr>
<tr>
<td>add(i: Item)</td>
</tr>
<tr>
<td>delete(i: Item)</td>
</tr>
</tbody>
</table>
```

Figure 14.7 Command Object Hierarchy

```
CommandInterface
| execute() |

AddCommand
| execute() |

DeleteCommand
| execute() |
```
public class CommandTest {
    public static void main(String[] args) {
        //Add an item to the CD category
        //create Receiver objects
        Item CD = new Item("A Beautiful Mind");
        Category catCD = new Category("CD");
        //create the command object
        CommandInterface command = new AddCommand(CD, catCD);
        //create the invoker
        ItemManager manager = new ItemManager();
        //configure the invoker
        //with the command object
        manager.setCommand(command);
        manager.process();
        //Add an item to the CD category
        CD = new Item("Duet");
        catCD = new Category("CD");
        command = new AddCommand(CD, catCD);
        manager.setCommand(command);
        manager.process();
        //Add an item to the New Releases category
        CD = new Item("Duet");
        catCD = new Category("New Releases");
        command = new AddCommand(CD, catCD);
        manager.setCommand(command);
        manager.process();
        //Delete an item from the New Releases category
        command = new DeleteCommand(CD, catCD);
        manager.setCommand(command);
        manager.process();
    }
}
Figure 14.8  Class Association

Figure 14.9  Message Flow When an Item Is Added to a Category
DESCRIPTION

In general, object-oriented applications consist of a set of objects that interact with each other for the purpose of providing a service. This interaction can be direct (point-to-point) as long as the number of objects referring to each other directly is very low. Figure 15.1 depicts this type of direct interaction where ObjectA and ObjectB refer to each other directly.

As the number of objects increases, this type of direct interaction can lead to a complex maze of references among objects (Figure 15.2), which affects the maintainability of the application. Also, having an object directly referring to other objects greatly reduces the scope for reusing these objects because of higher coupling.

Figure 15.1  Point-to-Point Communication in the Case of Two Objects

Figure 15.2  Point-to-Point Communication: Increased Number of Objects
In such cases, the Mediator pattern can be used to design a controlled, coordinated communication model for a group of objects, eliminating the need for objects to refer to each other directly (Figure 15.3).

The Mediator pattern suggests abstracting all object interaction details into a separate class, referred to as a Mediator, with knowledge about the interacting group of objects. Every object in the group is still responsible for offering the service it is designed for, but objects do not interact with each other directly for this purpose. The interaction between any two different objects is routed through the Mediator class. All objects send their messages to the mediator. The mediator then sends messages to the appropriate objects as per the application’s requirements. The resulting design has the following major advantages:

With all the object interaction behavior moved into a separate (mediator) object, it becomes easier to alter the behavior of object interrelationships, by replacing the mediator with one of its subclasses with extended or altered functionality.

Moving interobject dependencies out of individual objects results in enhanced object reusability.

Because objects do not need to refer to each other directly, objects can be unit tested more easily.

The resulting low degree of coupling allows individual classes to be modified without affecting other classes.

**MEDIATOR VERSUS FAÇADE**

In some aspects the Mediator pattern looks similar to the Façade pattern discussed earlier. Table 15.1 lists the similarities and differences between the two.

During the discussion of the Command pattern, we built two example applications. Let us revisit these applications and see how the direct object-to-object interaction can be avoided by applying the Mediator pattern.
Table 15.1 Mediator versus Façade

<table>
<thead>
<tr>
<th>Mediator</th>
<th>Façade</th>
</tr>
</thead>
<tbody>
<tr>
<td>A Mediator is used to abstract the necessary functionality of a group of objects with the aim of simplifying the object interaction.</td>
<td>A Façade is used to abstract the required functionality of a subsystem of components, with the aim of providing a simplified, higher level interface.</td>
</tr>
<tr>
<td>All objects interact with each other through the Mediator. The group of objects knows the existence of the Mediator.</td>
<td>Clients use the Façade to interact with subsystem components. The existence of the Façade is not known to the subsystem components.</td>
</tr>
<tr>
<td>Because the Mediator and all the objects that are registered with it can communicate with each other, the communication is bidirectional.</td>
<td>Clients can send messages (through the Façade) to the subsystem but not vice versa, making the communication unidirectional.</td>
</tr>
<tr>
<td>A Mediator can be assumed to stay in the middle of a group of interacting objects. Using a Mediator allows the implementation of any of the interacting objects to be changed without any impact on the other objects that interact with it only through the Mediator.</td>
<td>A Façade lies in between a client object and the subsystem. Using a Façade allows the implementation of the subsystem to be changed completely without any impact on its clients, provided the clients are not given direct access to the subsystem's classes.</td>
</tr>
<tr>
<td>By subclassing the Mediator, the behavior of the object interrelationships can be extended.</td>
<td>By subclassing the Façade, the implementation of the higher level interface can be changed.</td>
</tr>
</tbody>
</table>
MEMENTO

DESCRIPTION

The state of an object can be defined as the values of its properties or attributes at any given point of time. The Memento pattern is useful for designing a mechanism to capture and store the state of an object so that subsequently, when needed, the object can be put back to this (previous) state. This is more like an undo operation. The Memento pattern can be used to accomplish this without exposing the object’s internal structure. The object whose state needs to be captured is referred to as the originator. When a client wants to save the state of the originator, it requests the current state from the originator. The originator stores all those attributes that are required for restoring its state in a separate object referred to as a Memento and returns it to the client. Thus a Memento can be viewed as an object that contains the internal state of another object, at a given point of time. A Memento object must hide the originator variable values from all objects except the originator. In other words, it should protect its internal state against access by objects other than the originator. Towards this end, a Memento should be designed to provide restricted access to other objects while the originator is allowed to access its internal state.

When the client wants to restore the originator back to its previous state, it simply passes the memento back to the originator. The originator uses the state information contained in the memento and puts itself back to the state stored in the Memento object.
The Observer pattern is useful for designing a consistent communication model between a set of dependent objects and an object that they are dependent on. This allows the dependent objects to have their state synchronized with the object that they are dependent on. The set of dependent objects are referred to as observers and the object that they are dependent on is referred to as the subject. In order to accomplish this, the Observer pattern suggests a publisher-subscriber model leading to a clear boundary between the set of Observer objects and the Subject object.

A typical observer is an object with interest or dependency in the state of the subject. A subject can have more than one such observer. Each of these observers needs to know when the subject undergoes a change in its state.

The subject cannot maintain a static list of such observers as the list of observers for a given subject could change dynamically. Hence any object with interest in the state of the subject needs to explicitly register itself as an observer with the subject. Whenever the subject undergoes a change in its state, it notifies all of its registered observers. Upon receiving notification from the subject, each of the observers queries the subject to synchronize its state with that of the subject's. Thus a subject behaves as a publisher by publishing messages to all of its subscribing observers.

In other words, the scenario contains a one-to-many relationship between a subject and the set of its observers. Whenever the subject instance undergoes a state change, all of its dependent observers are notified and they can update themselves. Each of the observer objects has to register itself with the subject to get notified when there is a change in the subject's state. An observer can register or subscribe with multiple subjects. Whenever an observer does not wish to be notified any further, it unregisters itself with the subject.

For this mechanism to work:

The subject should provide an interface for registering and unregistering for change notifications.
One of the following two must be true:

- *In the pull model* — The subject should provide an interface that enables observers to query the subject for the required state information to update their state.
- *In the push model* — The subject should send the state information that the observers may be interested in.

Observers should provide an interface for receiving notifications from the subject.

The class diagram in Figure 17.1 describes the structure of different classes and their association, catering to the above list of requirements. From this class diagram it can be seen that:

All subjects are expected to provide implementation for an interface similar to the `Observable` interface.
All observers are expected to have an interface similar to the `Observer` interface.

Several variations can be thought of while applying the Observer pattern, leading to different types of subject-observers such as observers that are interested only in specific types of changes in the subject.

**ADDING NEW OBSERVERS**

After applying the Observer pattern, different observers can be added dynamically without requiring any changes to the `Subject` class. Similarly, observers remain unaffected when the state change logic of the subject changes.
This pattern was previously described in GoF95.

**DESCRIPTION**

In general, languages are made up of a set of grammar rules. Different sentences can be constructed by following these grammar rules. Sometimes an application may need to process repeated occurrences of similar requests that are a combination of a set of grammar rules. These requests are distinct but are similar in the sense that they are all composed using the same set of rules. A simple example of this sort would be the set of different arithmetic expressions submitted to a calculator program. Though each such expression is different, they are all constructed using the basic rules that make up the grammar for the language of arithmetic expressions.

In such cases, instead of treating every distinct combination of rules as a separate case, it may be beneficial for the application to have the ability to interpret a generic combination of rules. The Interpreter pattern can be used to design this ability in an application so that other applications and users can specify operations using a simple language defined by a set of grammar rules.

Applying the Interpreter pattern:

A class hierarchy can be designed to represent the set of grammar rules with every class in the hierarchy representing a separate grammar rule. An **Interpreter** module can be designed to interpret the sentences constructed using the class hierarchy designed above and carry out the necessary operations.

Because a different class represents every grammar rule, the number of classes increases with the number of grammar rules. A language with extensive, complex grammar rules requires a large number of classes. The Interpreter pattern works best when the grammar is simple. Having a simple grammar avoids the need to have many classes corresponding to the complex set of rules involved, which are hard to manage and maintain.
EXAMPLE

Let us build a calculator application that evaluates a given arithmetic expression. For simplicity, let us consider only add, multiply and subtract operations. Instead of designing a custom algorithm for evaluating each arithmetic expression, the application could benefit from interpreting a generic arithmetic expression. The Interpreter pattern can be used to design the ability to understand a generic arithmetic expression and evaluate it.

The Interpreter pattern can be applied in two stages:

1. Define a representation for the set of rules that make up the grammar for arithmetic expressions.
2. Design an interpreter that makes use of the classes that represent different arithmetic grammar rules to understand and evaluate a given arithmetic expression.

The set of rules in Table 18.1 constitutes the grammar for arithmetic expressions.

Table 18.1 Grammar Rules for Arithmetic Expressions

<table>
<thead>
<tr>
<th>Arithmetic Expressions – Grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArithmeticExpression::= ConstantExpression</td>
</tr>
<tr>
<td>MultiplyExpression</td>
</tr>
<tr>
<td>ConstantExpression::= Integer/Double Value</td>
</tr>
<tr>
<td>AddExpression::= ArithmeticExpression ‘+’ ArithmeticExpression</td>
</tr>
<tr>
<td>MultiplyExpression::= ArithmeticExpression ‘*’ ArithmeticExpression</td>
</tr>
<tr>
<td>SubtractExpression::= ArithmeticExpression ‘-’ ArithmeticExpression</td>
</tr>
</tbody>
</table>

From Table 18.1, it can be observed that arithmetic expressions are of two types — individual (e.g., ConstantExpression) or composite (e.g., AddExpression). These expressions can be arranged in the form of a tree structure, with composite expressions as nonterminal nodes and individual expressions as terminal nodes of the tree.

Let us define a class hierarchy as Figure 18.1 to represent the set of arithmetic grammar rules.

Each of the classes representing different rules implements the common Expression interface and provides implementation for the evaluate method (Listing 18.1 through Listing 18.5).

The Context is a common information repository that stores the values of different variables (Listing 18.6). For simplicity, values are hard-coded for variables in this example.

While each of the NonTerminalExpression classes performs the arithmetic operation it represents, the TerminalExpression class simply looks up the value of the variable it represents from the Context.
The application design can evaluate any expression. But for simplicity, the main Calculator (Listing 18.7) object uses a hard-coded arithmetic expression \((a + b) \times (c - d)\) as the expression to be interpreted and evaluated.
Listing 18.2  NonTerminalExpression Class

```java
public abstract class NonTerminalExpression
    implements Expression {
private Expression leftNode;
private Expression rightNode;
public NonTerminalExpression(Expression l, Expression r) {
    setLeftNode(l);
    setRightNode(r);
}
public void setLeftNode(Expression node) {
    leftNode = node;
}
public void setRightNode(Expression node) {
    rightNode = node;
}
public Expression getLeftNode() {
    return leftNode;
}
public Expression getRightNode() {
    return rightNode;
}
}//NonTerminalExpression
```

Listing 18.3  AddExpression Class

```java
class AddExpression extends NonTerminalExpression {
public int evaluate(Context c) {
    return getLeftNode().evaluate(c) +
             getRightNode().evaluate(c);
}
public AddExpression(Expression l, Expression r) {
    super(l, r);
}
}//AddExpression
```

The Calculator object carries out the interpretation and evaluation of the input expression in three stages:
Listing 18.4  SubtractExpression Class

```java
class SubtractExpression extends NonTerminalExpression {
    public int evaluate(Context c) {
        return getLeftNode().evaluate(c) -
            getRightNode().evaluate(c);
    }
    public SubtractExpression(Expression l, Expression r) {
        super(l, r);
    }
} // SubtractExpression
```

Listing 18.5  MultiplyExpression Class

```java
class MultiplyExpression extends NonTerminalExpression {
    public int evaluate(Context c) {
        return getLeftNode().evaluate(c) *
            getRightNode().evaluate(c);
    }
    public MultiplyExpression(Expression l, Expression r) {
        super(l, r);
    }
} // MultiplyExpression
```

1. Infix-to-postfix conversion — The input infix expression is first translated into an equivalent postfix expression.
2. Construction of the tree structure — The postfix expression is then scanned to build a tree structure.
3. Postorder traversal of the tree — The tree is then postorder traversed for evaluating the expression.

```java
public class Calculator {
    ...
    ...
    public int evaluate() {
        // infix to Postfix
        String pfExpr = infixToPostFix(expression);
    }
```
class Context {
    private HashMap varList = new HashMap();
    public void assign(String var, int value) {
        varList.put(var, new Integer(value));
    }
    public int getValue(String var) {
        Integer objInt = (Integer) varList.get(var);
        return objInt.intValue();
    }
    public Context() {
        initialize();
    }
    //Values are hardcoded to keep the example simple
    private void initialize() {
        assign("a", 20);
        assign("b", 40);
        assign("c", 30);
        assign("d", 10);
    }
}

//build the Binary Tree
Expression rootNode = buildTree(pfExpr);
//Evaluate the tree
return rootNode.evaluate(ctx);
}

Infix-to-Postfix Conversion (Listing 18.8)

An expression in the standard form is an infix expression.

Example: (a + b) * (c – d)

An infix expression is more easily understood by humans but is not suitable for evaluating expressions by computers. The usage of precedence rules and parentheses in the case of complex expressions makes it difficult for computer evaluation of
public class Calculator {
    private String expression;
    private HashMap operators;
    private Context ctx;

    public static void main(String[] args) {
        Calculator calc = new Calculator();
        //instantiate the context
        Context ctx = new Context();
        //set the expression to evaluate
        calc.setExpression("(a+b)*(c-d)");
        //configure the calculator with the
        //Context
        calc.setContext(ctx);
        //Display the result
        System.out.println(" Variable Values: " +
          "a=" + ctx.getValue("a") +
          ", b=" + ctx.getValue("b") +
          ", c=" + ctx.getValue("c") +
          ", d=" + ctx.getValue("d");
        System.out.println(" Expression = (a+b)*(c-d)");
        System.out.println(" Result = " + calc.evaluate());
    }

    public Calculator() {
        operators = new HashMap();
        operators.put("+", "1");
        operators.put("-", "1");
        operators.put("/", "2");
        operators.put("*", "2");
        operators.put("", "0");
    }

    ...  
    ...

    //End of class

these expressions. A postfix expression does not contain parentheses, does not
involve precedence rules and is more suitable for evaluation by computers.

The postfix equivalent of the example expression above is ab+cd–*.

A detailed description of the process of converting an infix expression to its
postfix form is provided in the Additional Notes section.
public class Calculator {
    
    private String infixToPostFix(String str) {
        Stack s = new Stack();
        String pfExpr = "";
        String tempStr = "";
        String expr = str.trim();
        for (int i = 0; i < str.length(); i++) {
            String currChar = str.substring(i, i + 1);
            if ((isOperator(currChar) == false) && (!currChar.equals("(")) && (!currChar.equals(")")) {
                pfExpr = pfExpr + currChar;
            }
            if (currChar.equals("(")) {
                s.push(currChar);
            }
            //for ')' pop all stack contents until '('
            if (currChar.equals("") && s.isEmpty() == false) {
                tempStr = (String) s.pop();
                while (!tempStr.equals("(")) {
                    pfExpr = pfExpr + tempStr;
                    tempStr = (String) s.pop();
                }
                tempStr = "";
            }
            //if the current character is an operator
            if (isOperator(currChar)) {
                if (tempStr.equals("") == false) {
                    tempStr = (String) s.pop();
                    String strVal1 =
                }
            }
        }
    }
(continued)
(String) operators.get(tempStr);
int vall = new Integer(strVall).intValue();
String strVall2 =
(String) operators.get(currChar);
int val2 = new Integer(strVall2).intValue();
while ((vall >= val2)) {
pfExpr = pfExpr + tempStr;
vall = -100;
if (s.isEmpty() == false) {
    tempStr = (String) s.pop();
    strVall = (String) operators.get( 
        tempStr);
    vall = new Integer(strVall).intValue();
}
}
if ((vall < val2) && (vall != -100))
s.push(tempStr);
}
s.push(currChar);
} //if
} //for
while (s.isEmpty() == false) {
tempStr = (String) s.pop();
pfExpr = pfExpr + tempStr;
}
return pfExpr;
}
...
...
} //End of class

Construction of the Tree Structure (Listing 18.9)

The postfix equivalent of the input infix expression is scanned from left to right and a tree structure is built using the following algorithm:

1. Initialize an empty stack.
2. Scan the postfix string from left to right.
public class Calculator {
    
    public void setContext(Context c) {
        ctx = c;
    }
    public void setExpression(String expr) {
        expression = expr;
    }
    
    private Expression buildTree(String expr) {
        Stack s = new Stack();
        for (int i = 0; i < expr.length(); i++) {
            String currChar = expr.substring(i, i + 1);
            if (isOperator(currChar) == false) {
                Expression e = new TerminalExpression(currChar);
                s.push(e);
            } else {
                Expression r = (Expression) s.pop();
                Expression l = (Expression) s.pop();
                Expression n =
                    getNonTerminalExpression(currChar, l, r);
                s.push(n);
            }
        } //for
        return (Expression) s.pop();
    }
    
} //End of class

3. If the scanned character is an operand:
   a. Create an instance of the TerminalExpression class by passing the
      scanned character as an argument.
   b. Push the TerminalExpression object to the stack.
4. If the scanned character is an operator:
   a. Pop two top elements from the stack.
   b. Create an instance of an appropriate NonTerminalExpression sub-
      class by passing the two stack elements retrieved above as arguments.
5. Repeat Step 3 and Step 4 for all characters in the postfix string.
6. The only remaining element in the stack is the root of the tree structure.

The example postfix expression ab+cd–* results in the following tree structure
as in Figure 18.2.

![Figure 18.2 Example Expression: Tree Structure](image)

**Postorder Traversal of the Tree**

The Calculator traverses the tree structure and evaluates different Expression
objects in its postorder traversal path. There are four major tree traversal tech-
niques. These techniques are discussed as part of the Additional Notes section.
Because the binary tree in the current example is a representation of a postfix
expression, the postorder traversal technique is followed for the expression
evaluation. The Calculator object makes use of a helper Context object to
share information with different Expression objects constituting the tree struc-
ture. In general, a Context object is used as a global repository of information.
In the current example, the Calculator object stores the values of different
variables in the Context, which are used by each of different Expression
objects in evaluating the part of the expression it represents.

The postorder traversal of the tree structure in Figure 18.2 results in the
evaluation of the leftmost subtree in a recursive manner, followed by the rightmost
subtree, then the NonTerminalExpression node representing an operator.

**ADDITIONAL NOTES**

**Infix-to-Postfix Conversion**

**Infix Expression**

An expression in the standard form is an infix expression.

Example: a * b + c/d

Sometimes, an infix expression is also referred to as an in-order expression.
**Postfix Expression**

The postfix (postorder) form equivalent of the above example expression is \(ab*cd/+\).

**Conversion Algorithm**

See Table 18.2 for the conversion algorithm.

**Table 18.2 Conversion Algorithm**

1. Define operator precedence rules — In general arithmetic, the descending order of precedence is as shown in the rules below:

<table>
<thead>
<tr>
<th>Precedence Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>*, /</td>
</tr>
<tr>
<td>Same precedence</td>
</tr>
<tr>
<td>+, –</td>
</tr>
<tr>
<td>Same precedence</td>
</tr>
<tr>
<td>Expressions are evaluated from left to right.</td>
</tr>
</tbody>
</table>

2. Initialize an empty stack.
3. Initialize an empty postfix expression.
4. Scan the infix string from left to right.
5. If the scanned character is an operand, add it to the postfix string.
6. If the scanned character is a left parenthesis, push it to the stack.
7. If the scanned character is a right parenthesis:
   a. Pop elements from the stack and add to the postfix string until the stack element is a left parenthesis.
   b. Discard both the left and the right parenthesis characters.
8. If the scanned character is an operator:
   a. If the stack is empty, push the character to the stack.
   b. If the stack is not empty:
      i. If the element on top of the stack is an operator:
         A. Compare the precedence of the character with the precedence of the element on top of the stack.
         B. If top element has higher or equal precedence over the scanned character, pop the stack element and add it to the Postfix string. Repeat this step as long as the stack is not empty and the element on top of the stack has equal or higher precedence over the scanned character.
         C. Push the scanned character to stack.
      ii. If the element on top of the stack is a left parenthesis, push the scanned character to the stack.
9. Repeat Steps 5 through 8 above until all the characters are scanned.
10. After all characters are scanned, continue to pop elements from the stack and add to the postfix string until the stack is empty.
11. Return the postfix string.
**Example**

As an example, consider the infix expression \((A + B) \times (C – D)\). Let us apply the algorithm described above to convert this expression into its postfix form.

Initially the stack is empty and the postfix string has no characters. Table 18.3 shows the contents of the stack and the resulting postfix expression as each character in the input infix expression is processed.

<table>
<thead>
<tr>
<th>Infix Expression Character</th>
<th>Observation and Action to Be Taken</th>
<th>Stack</th>
<th>Postfix String</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>Push to the stack.</td>
<td>(</td>
<td>(</td>
</tr>
<tr>
<td>A</td>
<td>Operand. Add to the postfix string.</td>
<td>( A</td>
<td>A</td>
</tr>
<tr>
<td>+</td>
<td>Operator. The element on top of the stack is a left parenthesis and hence push + to the stack.</td>
<td>(+</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>Operand. Add to the postfix string.</td>
<td>(+ AB</td>
<td>AB</td>
</tr>
<tr>
<td>)</td>
<td>Right parenthesis. Pop elements from the stack until a left parenthesis is found. Add these stack elements to the postfix string. Discard both left and right parentheses.</td>
<td>AB+</td>
<td>AB+</td>
</tr>
<tr>
<td>*</td>
<td>Operator. The element on top of the stack is +. The precedence of + is less than the precedence of *. Push the operator to the stack.</td>
<td>* AB+</td>
<td>AB+</td>
</tr>
<tr>
<td>(</td>
<td>Push to the stack.</td>
<td>*(</td>
<td>AB+</td>
</tr>
<tr>
<td>C</td>
<td>Operand. Add to the postfix string.</td>
<td>*( AB+C</td>
<td>AB + C</td>
</tr>
<tr>
<td>–</td>
<td>Operator. The element on top of the stack is a left parenthesis and hence push + to the stack.</td>
<td>*(– AB+C</td>
<td>AB + C</td>
</tr>
<tr>
<td>D</td>
<td>Operand. Add to the Postfix string.</td>
<td>*(– AB+CD</td>
<td>AB + CD–</td>
</tr>
<tr>
<td>)</td>
<td>Right parenthesis. Pop elements from the stack until a left parenthesis is found. Add these stack elements to the postfix string. Discard both left and right parentheses.</td>
<td>*( AB+CD–</td>
<td></td>
</tr>
</tbody>
</table>

All characters in the infix expression are scanned

Add all remaining stack elements to the postfix string.

<table>
<thead>
<tr>
<th>Stack</th>
<th>Postfix String</th>
</tr>
</thead>
<tbody>
<tr>
<td>AB + CD–</td>
<td>AB + CD–*</td>
</tr>
</tbody>
</table>

**Binary Tree Traversal Techniques**

There are four different tree traversal techniques — Preorder, In-Order, Postorder and Level-Order. Let us discuss each of these techniques by using the following binary tree in Figure 18.3 as an example.
**Figure 18.3  Example Sorted Tree Structure**

**Preorder (Node-Left-Right)**
Start with the root node and follow the algorithm as follows:

- Visit the node first.
- Traverse the left subtree in preorder.
- Traverse the right subtree in preorder.

A preorder traversal of the above sorted tree structure to print the contents of the nodes constituting the tree results in the following display:

KDAGFSMPU

**In-Order (Left-Node-Right)**
Start with the root node and follow the algorithm as follows:

- Traverse the left subtree in in-order.
- Visit the node.
- Traverse the right subtree in in-order.

An in-order traversal of the above sorted tree structure to print the contents of the nodes constituting the tree results in the following display:

ADFGKMSPU

**Postorder (Left-Right-Node)**
Start with the root node and follow the algorithm as follows:

- Traverse the left subtree in in-order.
- Traverse the right subtree in in-order.
- Visit the node.

A postorder traversal of the above sorted tree structure to print the contents of the nodes constituting the tree results in the following display:
**Level-Order**

Start with the root node level and follow the algorithm as follows:

- Traverse different levels of the tree structure from top to bottom.
- Visit nodes from left to right within each level.

A level-order traversal of the above sorted tree structure to print the contents of the nodes constituting the tree results in the following display:

```
KDSAGMUFPP
```
STATE

DESCRIPTION

The state of an object can be defined as its exact condition at any given point of time, depending on the values of its properties or attributes. The set of methods implemented by a class constitutes the behavior of its instances. Whenever there is a change in the values of its attributes, we say that the state of an object has changed.

A simple example of this would be the case of a user selecting a specific font style or color in an HTML editor. When a user selects a different font style or color, the properties of the editor object change. This can be considered as a change in its internal state.

The State pattern is useful in designing an efficient structure for a class, a typical instance of which can exist in many different states and exhibit different behavior depending on the state it is in. In other words, in the case of an object of such a class, some or all of its behavior is completely influenced by its current state. In the State design pattern terminology, such a class is referred to as a Context class. A Context object can alter its behavior when there is a change in its internal state and is also referred as a Stateful object.

STATEFUL OBJECT: AN EXAMPLE

Most of the HTML editors available today offer different views of an HTML page at the time of creation. Let us consider one such editor that offers three views of a given Web page as follows:

1. Design view — In this view, a user is allowed to visually create a Web page without having to know about the internal HTML commands.
2. HTML view — This view offers a user the basic structure of the Web page in terms of the HTML tags and lets a user customize the Web page with additional HTML code.
3. Quick page view — This view provides a preview of the Web page being created.

When a user selects one of these views (change in the state of the Editor object), the behavior of the Editor object changes in terms of the way the current Web page is displayed.

The State pattern suggests moving the state-specific behavior out of the Context class into a set of separate classes referred to as State classes. Each of the many different states that a Context object can exist in can be mapped into a separate State class. The implementation of a State class contains the context behavior that is specific to a given state, not the overall behavior of the context itself.
The context acts as a client to the set of State objects in the sense that it makes use of different State objects to offer the necessary state-specific behavior to an application object that uses the context in a seamless manner.

In the absence of such a design, each method of the context would contain complex, inelegant conditional statements to implement the overall context behavior in it. For example,

```java
public Context{
    ...
    ...
    someMethod(){
        if (state_1){
            //do something
        } else if (state_2){
            //do something else
        }
        ...
    ...
    }
    ...
    ...
}
```

By encapsulating the state-specific behavior in separate classes, the context implementation becomes simpler to read: free of too many conditional statements such as if-else or switch-case constructs. When a Context object is first created, it initializes itself with its initial State object. This State object becomes the current State object for the context. By replacing the current State object with a new State object, the context transitions to a new state. The client application using the context is not responsible for specifying the current State object for the context, but instead, each of the State classes representing specific states are expected to provide the necessary implementation to transition the context into other states.

When an application object makes a call to a Context method (behavior), it forwards the method call to its current State object.

```java
public Context{
    ...
    ...
    someMethod(){
        objCurrentState.someMethod();
    }
    ...
    ...
}
```
The Strategy pattern is useful when there is a set of related algorithms and a client object needs to be able to dynamically pick and choose an algorithm from this set that suits its current need.

The Strategy pattern suggests keeping the implementation of each of the algorithms in a separate class. Each such algorithm encapsulated in a separate class is referred to as a strategy. An object that uses a Strategy object is often referred to as a context object.

With different Strategy objects in place, changing the behavior of a Context object is simply a matter of changing its Strategy object to the one that implements the required algorithm.

To enable a Context object to access different Strategy objects in a seamless manner, all Strategy objects must be designed to offer the same interface. In the Java programming language, this can be accomplished by designing each Strategy object either as an implementer of a common interface or as a subclass of a common abstract class that declares the required common interface.

Once the group of related algorithms is encapsulated in a set of Strategy classes in a class hierarchy, a client can choose from among these algorithms by selecting and instantiating an appropriate Strategy class. To alter the behavior of the context, a client object needs to configure the context with the selected strategy instance. This type of arrangement completely separates the implementation of an algorithm from the context that uses it. As a result, when an existing algorithm implementation is changed or a new algorithm is added to the group, both the context and the client object (that uses the context) remain unaffected.

Implementing different algorithms in the form of a method using conditional statements violates the basic object-oriented, open-closed principle. Designing each algorithm as a different class is a more elegant approach than designing all different algorithms as part of a method in the form of a conditional statement. Because each algorithm is contained in a separate class, it becomes simpler and easier to add, change or remove an algorithm.
Another approach would be to subclass the context itself and implement different algorithms in different subclasses of the context. This type of design binds the behavior to a context subclass and the behavior executed by a context subclass becomes static. With this design, to change the behavior of the context, a client object needs to create an instance of a different subclass of the context and replace the current Context object with it.

Having different algorithms encapsulated in different Strategy classes decouples the context behavior from the Context object itself. With different Strategy objects available, a client object can use the same Context object and change its behavior by configuring it with different Strategy objects. This is a more flexible approach than subclassing.

Also, sometimes subclassing can lead to a bloated class hierarchy. We have seen an example of this during the discussion of the Decorator pattern. Designing algorithms as different Strategy classes keeps the class growth linear.

**STRATEGY VERSUS STATE**

From the discussion above, the Strategy pattern looks very similar to the State pattern discussed earlier. One of the differences between the two patterns is that the Strategy pattern deals with a set of related algorithms, which are more similar in what they do as opposed to different state-specific behavior encapsulated in different State objects in the State pattern.

Table 20.1 provides a detailed list of similarities and differences between the State and the Strategy patterns.
## Table 20.1 State versus Strategy

<table>
<thead>
<tr>
<th>State Pattern</th>
<th>Strategy Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Different types of possible behavior of an object are implemented in the form of a group of separate objects (State objects).</td>
<td>Similar to the State pattern, specific behaviors are modeled in the form of separate classes (Strategy objects).</td>
</tr>
<tr>
<td>The behavior contained in each State object is specific to a given state of the associated object.</td>
<td>The behavior contained in each Strategy object is a different algorithm (from a set of related algorithms) to provide a given functionality.</td>
</tr>
<tr>
<td>An object that uses a State object to change its behavior is referred to as a Context object. A Context object needs to change its current State object to change its behavior.</td>
<td>An object that uses a Strategy object to alter its behavior is referred to as a Context object. Similar to the State pattern, for a Context object to behave differently, it needs to be configured with a different Strategy object.</td>
</tr>
<tr>
<td>Often, when an instance of the context is first created, it is associated with one of the default State objects.</td>
<td>Similarly, a context is associated with a default Strategy object that implements the default algorithm.</td>
</tr>
<tr>
<td>A given State object itself can put the context into a new state. This makes a new State object as the current State object of the context, changing the behavior of the Context object.</td>
<td>A client application using the context needs to explicitly assign a strategy to the context. A Strategy object cannot cause the context to be configured with a different Strategy object.</td>
</tr>
<tr>
<td>The choice of a State object is dependent on the state of the Context object.</td>
<td>The choice of a Strategy object is based on the application need. Not on the state of the Context object.</td>
</tr>
<tr>
<td>A given Context object undergoes state changes. The order of transition among states is well defined. These are the characteristics of an application where the State pattern could be applied. <strong>Example:</strong> A bank account behaves differently depending on the state it is in when a transaction to withdraw money is attempted. When the minimum balance is maintained — no transaction fee is charged. When the minimum balance is not maintained — transaction fee is charged. When the account is overdrawn — the transaction is not allowed.</td>
<td>A given Context object does not undergo state changes. <strong>Example:</strong> An application that needs to encrypt and save the input data to a file. Different encryption algorithms can be used to encrypt the data. These algorithms can be designed as Strategy objects. The client application can choose a strategy that implements the required algorithm.</td>
</tr>
</tbody>
</table>
The term *null* is used in most computer programming languages to refer to a nonexisting object. The Null Object pattern is applicable when a client expects to use different subclasses of a class hierarchy to execute different behavior and refers these subclasses as objects of the parent class type. At times, it may be possible that a subclass instance may not be available when the client expects one. In such cases, what a client object receives is a nonexisting object or null. When a null is returned, the client cannot invoke methods as it would if a real object is returned. Hence the client needs to check to make sure that the object is not null before invoking any of its methods. In the case of a null, the client can either provide some default behavior or do nothing.

Applying the Null Object pattern in such cases eliminates the need for a client to check if an object is null every time the object is used.

The Null Object pattern recommends encapsulating the default (or usually the do nothing) behavior into a separate class referred to as a *Null Object*. This class can be designed as one of the subclasses in the class hierarchy. Thus the Null Object provides the same set of methods as other subclasses do, but with the default (or do nothing) implementation for its methods. With the Null Object in place, when no subclass with real implementation is available, the Null Object is made available to the client. This type of arrangement eliminates the possibility of a client receiving a nonexisting object and hence the client does not need to check if the object it received is null (or nonexisting). Because the Null Object offers the same interface as other subclass objects, the client can treat them all in a uniform manner.

The following example shows how the Null Object pattern can be used to address a special case requirement of the message logging utility we built as an example of the Factory Method pattern.