2014

Design Patterns: Participant Guide

JP Morgan Chase & Co.
Technology Training

Contents

Module 1: Collection of Design Patterns 4
Lesson 1: Abstract Factory Design Pattern 5
Topic 1: Abstract Factory Definition 6
Topic 2: Case Study 10
Topic 3: Areas of Application 17
Lesson 2: Adapter Design Pattern 20
Topic 1: Adapter Design Pattern Introduction 21
Topic 2: Implementing the Adapter Pattern 22
Topic 3: Case Study 24
Topic 4: Areas of Application 30
Lesson 3: Command Design Pattern 33
Topic 1: Command Design Pattern Introduction 34
Topic 2: Case Study 38
Topic 3: Command Pattern Applied in Java 42
Topic 4: Areas of Application of the Command Pattern 44
Lesson 4: Template Method Design Pattern 47
Topic 1: Template Method Design Pattern Introduction 48
Topic 2: Case Study 48
Topic 3: Areas of Application of the Template Method Design Pattern 54
Lesson 5: Façade 59
Topic 1: Façade Pattern Defined 60
Topic 2: Façade Implementation 65
Lesson 6: State Design Pattern 69
Topic 1: Introduction to the State Design Pattern 70
Topic 2: Understanding the State Pattern 73
Topic 3: Implementing the State Pattern 78
Topic 4: Advantages and Disadvantages of Using the State Design Pattern 83
Topic 5: Application Areas of the State Design Pattern 83
Lesson 7: Proxy Design Pattern 87

Topic 1: Proxy Definition 88
Topic 2: Case Study 92
Lesson 8: Other Design Patterns 103
Topic 1: Chain of Responsibility Design Pattern 104
Topic 2: Strategy Design Pattern 112
Topic 3: Mediator Design Pattern 118
Module 2: Other Industry Patterns 124
Lesson 1: Service-Oriented Architecture 125
Topic 1: Introduction to SOA 125
Topic 3: Services in SOA 126
Topic 4: Implementation of SOA 127
Lesson 2: AntiPatterns 130
Topic 1: What are Antipatterns 130
Topic 2: Common Object-oriented Antipatterns 131
Topic 3: Programming Antipatterns 137
Topic 4: Importance of Antipattern 138

Module 1: Collection of Design Patterns

Module Overview

In application development, a design pattern can be called a reusable solution to a problem that occurs commonly in a given context. A design pattern itself is not a finished design, but rather a template or description that can be used by developers to build applications for different situations. Design patterns can also be referred to as best practices adopted by developers, when programming in a particular context. Object-oriented design patterns describe the relationships and interactions between classes and objects in an application, without actually specifying the classes and objects involved. In this module, you will be introduced to the following design patterns, their use in writing code, their areas of applications, and their representation using the Unified Modeling Language (UML) diagrams.

- Abstract Factory design pattern
- Adapter design pattern
- Command design pattern
- Template design pattern
- Façade design pattern
- State design pattern
- Proxy design pattern
- Other design patterns

Module Objectives

At the end of this module, you will be able to:

- Define various design patterns
- Implement the various design patterns using Java

Lesson 1: Abstract Factory Design Pattern

Lesson Overview

In the first lesson, you will be introduced to the Abstract Factory design pattern which is a creational design pattern used in application development. In applications, when you use a pattern to create objects, classes, or interfaces, that pattern is called a creational design pattern. By using the Abstract Factory design pattern, you can create a group of factories, with each factory having a common theme without specifying their concrete types. The Abstract Factory design pattern is useful in creating an application that performs different, yet similar tasks with the same set of elements by changing the method of implementation. Hence, developers can benefit by writing, just one set of code that satisfies multiple requirements of the same kind. This lesson discusses the definition, advantages, and uses of the Abstract Factory design pattern in detail.

Lesson Objectives

At the end of this lesson, you will be able to:

- Define the Abstract Factory design pattern
- Identify the situations where the Abstract Factory design pattern can be used
- Implement the Abstract Factory design pattern
- Provide the solution for a case study using UML diagram
- Define real-life applications of the Abstract Factory design pattern

Topic 1: Abstract Factory Definition

In real life, this pattern is found in the sheet metal stamping equipment used in the manufacture of automobiles. The stamping equipment is an Abstract Factory, which creates auto body parts. The same machinery is used to stamp right hand doors, left hand doors, right front fenders, left front fenders, hoods, etc. for different models of cars. Through the use of rollers to change the stamping dies, the concrete classes produced by the machinery can be changed within three minutes. [Michael Duell, "Non-software examples of software design patterns", Object Magazine, Jul 97].

The **Abstract Factory design pattern** provides a way to encapsulate a group of individual factories that have a common theme without specifying their concrete classes. In normal usage, the client software first creates a concrete implementation of the abstract factory. The factory then creates concrete objects for the client. These concrete objects are part of factory's theme. The client doesn't know (or care) which concrete objects it gets from each of these internal factories, since it uses only the generic interfaces of their products. This pattern separates the details of implementation of a set of objects from their general usage and relies on object composition, as object creation is implemented in methods exposed in the Factory interface.

Introduction to the Abstract Factory Design Pattern

Let us look at an example to understand the working of the Factory method. Consider an application that needs to run on both Windows and Mac operating systems. The application needs to generate a User Interface (UI) that will be able to support both types of Operating Systems (OS).

To solve this problem, there are concrete subclasses of GUIFactory to provide for standard look-and-feel. Each subclass (Winfactory and OSXFactory) implements the operations to create the appropriate UI component for the look and feel. For example, the createButton operation on WinFactory creates and returns a Windows look-and-feel button, while the corresponding operation on OSXFactory returns a button for MAC OSX. Clients create these UI components solely through the GUIFactory interface and have no knowledge of the classes that implement widgets for the particular OS. In other words, clients only have to interact with an interface defined by an abstract class, not a particular concrete class.

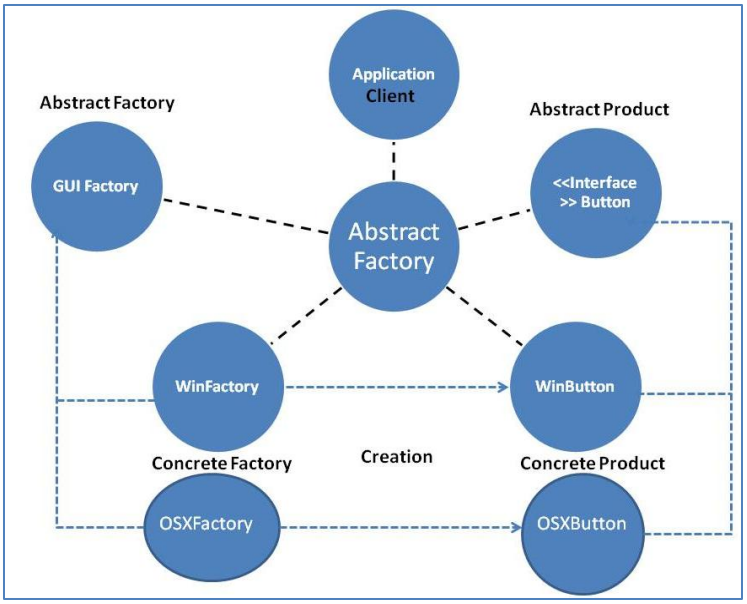


Figure 1.1: Abstract Factory Implemented In GUI Factory Example

Client code would simply look, as displayed in Code Segment 1.1.

```
public void clientCode(GUIFactory gui){
    Button button = gui.createButton();
    button.paint();

}
```

Code Segment 1.1: Client Code Interacting with GUI Factory

The `clientCode()` is not aware of the type of OS Factory being used or what kind of button is being rendered by the Abstract factory. This is shown in Figure 1.2.

The GUIFactory also helps to enforce dependencies between the concrete widget classes. A Windows scroll bar should be used with a Windows button, and that constraint is enforced automatically as a consequence of using a WinFactory.
This example demonstrates the essence of the Abstract Factory design pattern which is to provide an interface for creating families of related objects without defining their concrete classes. The reason for using the Abstract Factory design pattern is to:

- Decouple the creation of objects from their usage.
- Create families of logically related objects without depending on their concrete classes.

Let us now look at the class diagram of the Abstract factory pattern, as shown in Figure 1.2.

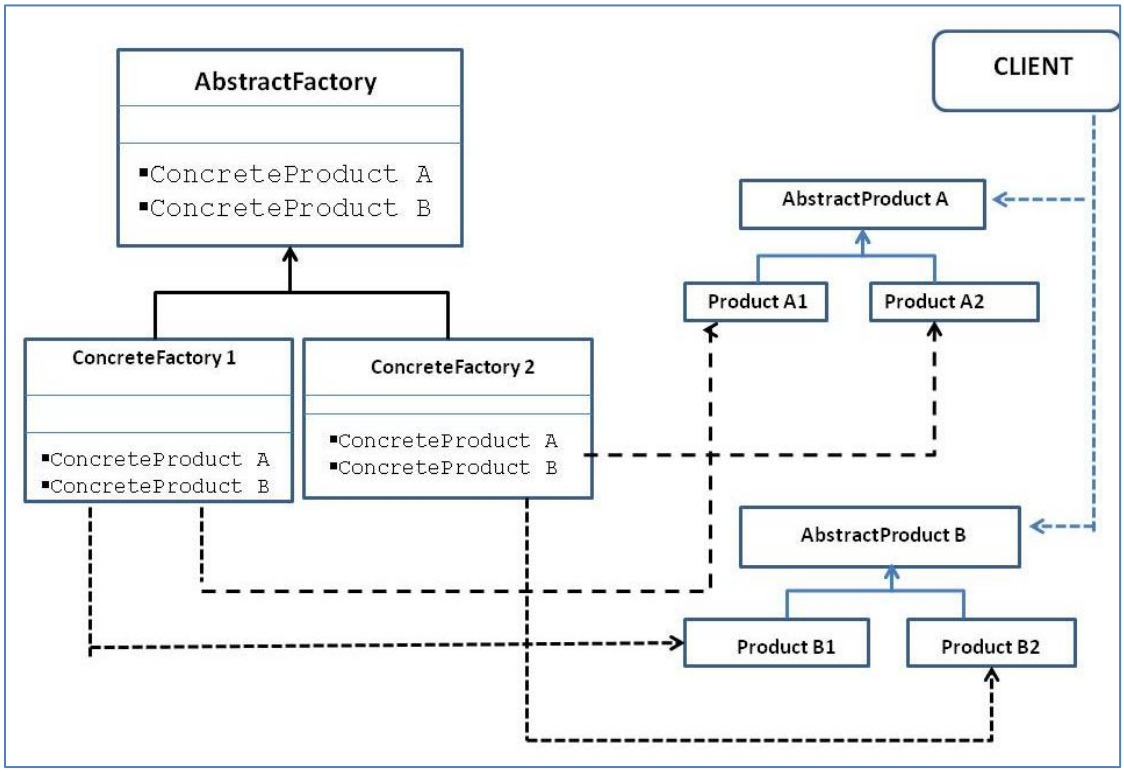


Figure 1.2: Class Diagram of Abstract Factory Design Pattern

Participants

- AbstractFactory: Is an interface for operations that create abstract products
- ConcreteFactory: Is a class that Implements the operations to create concrete products
- AbstractProduct: Declares an interface for a type of product.
- ConcreteProduct:

  - Is a concrete class that defines a product object to be created by the corresponding concrete factory
  - Implements the AbstractProduct interface

- Client: Interacts with AbstractFactory and use only the interfaces declared by the AbstractFactory and AbstractProduct classes.

Uses of Abstract Factory Design Pattern

Abstract factory pattern has the following uses:

- **Abstraction of types:** As shown in Figure 1.2, the client code has no knowledge, of the concrete type. The client code deals only with the abstract type and accesses such objects only through their abstract interface.

- **Easy modification of code:** Adding new concrete types is done easily by modifying the client code, to use a different concrete factory. The different factory then creates objects of a *different* concrete type, however still returns the *same* abstract type as before, thus the client code does not get impacted. This is significantly easier than modifying the client code to instantiate a new type, which would require changing *every* location in the code, when a new object is created.

- **Modularizing the application**: Abstract Factory design pattern helps make objects that follow a general pattern by creating the objects via a concrete factory to have a specific implementation. This helps create families of objects.

In the next topic, you will learn how the Abstract Factory design pattern is implemented, using a Java-based example.

Topic 2: Case Study

Think of a DAO class called CustomerDAO. Since you are going to support multiple databases, it makes sense to split it into two types, CustomerOracleDbDAO and CustomerMYSqlDbDAO both implementing the CustomerDAO interface. This way, other DAO classes will also need to be split into the two types. In other words, you need families of DAO classes, one family per database. You need to do this in such a way that it is completely invisible to the business layer. Some key points about the families are:

1. Each family will contain DAO classes pertaining to one specific database and can re-use same code for creating DB connection etc.
2. They may be dependent on each other too. Think of CustomerOracleDbDAO making use of AccountOracleDbDAO to fetch the account balance.

For such a design, you need to make use of factories (one factory per database) that will help in creating objects of DAO classes depending on what database the application is connected to. The business layer will not need to know or care about what DAO implementation class it is working with. It will only interact with interfaces.

To better understand how using the Abstract Factory pattern will solve the problem, take a look at the following diagram and try to relate it to the Abstract Factory class diagram.

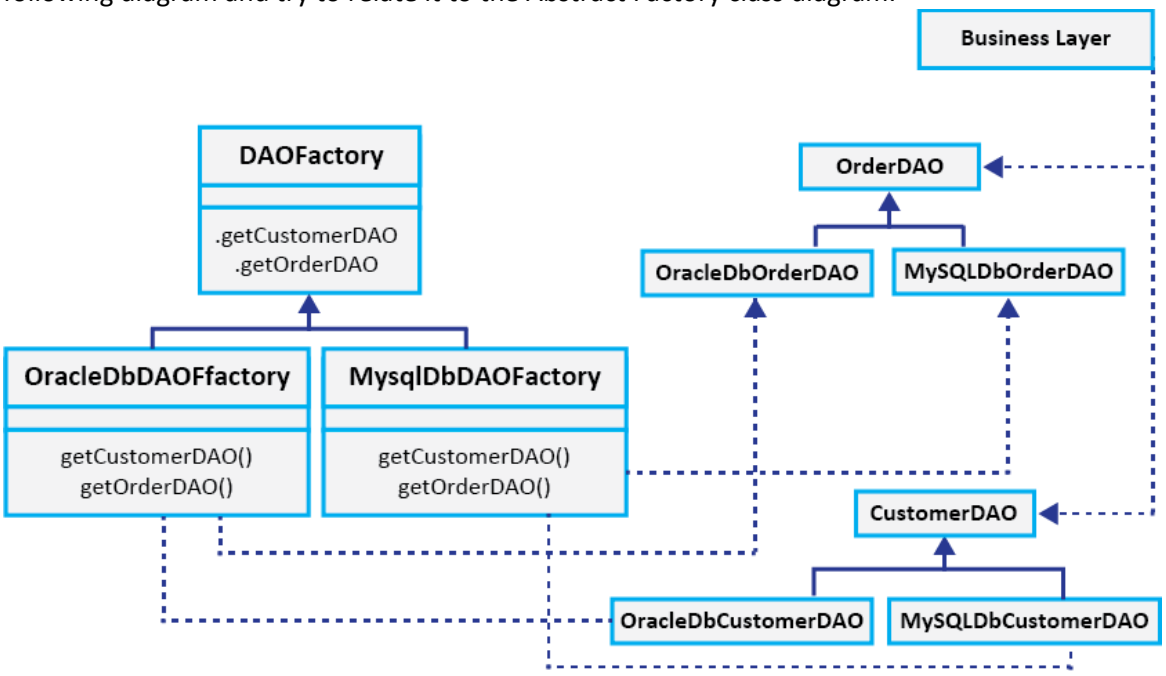


Figure 1.3: Example of Abstract Factory Design Pattern

As seen in the diagram, the service layer will directly interact with the abstract DAO Factory class and abstract products like CustomerDAO and OrderDAO. It will remain completely insulated from object creation by asking a factory object to create an appropriate DAO object. So if the application is connected to Oracle, then a CustomerOracleDbDAO will be returned via OracleDBDAOFactory while if the database is MYSQL, an object of CustomerMYSQLDbDAO will be returned via MySQLDBDAOFactory. In both cases, the return type will be CustomerDAO so the service layer remains insulted from the actual implementation class and will require no change even if you have to add support for another database in the future.

To implement the Abstract Factory pattern in the banking application, you need to perform the following steps:

1. **Define the service layer**: In this step, you have to first define a `Service Layer` class, which holds all the database-related operations that are to be performed by the application. These operations are:

    - Create a new customer account.
    - Find an existing customer account.
    - Modify the values in a customer's table based on transactions that have taken place.
    - Delete a customer's information when the customer decides to close an account.

The `Service` Layer class is shown in Code Segment 1.2.

```
public class ServiceLayer {

// create the required DAO Factory
DAOFactory daoFactory = DAOFactory.getDAOFactory();

// Create a DAO
CustomerDAO custDAO = daoFactory.getCustomerDAO();

// create a new customer
int newCustNo = custDAO.insertCustomer(...);

// Find a customer object.
Customer cust = custDAO.findCustomer(...);

// modify the values in the Customer Object.
cust.setAddress(...);
cust.setEmail(...);

// update the customer object using the DAO
custDAO.updateCustomer(cust);

// delete a customer object
custDAO.deleteCustomer(...);


}
```

Code Segment 1.2: `Service Layer` Class

2. **Create Abstract Factory class**: In this step, you have to create the `AbstractFactory` class named `DAOFactory`, which will further create concrete factories for each database. This is shown in Code Segment 1.3.

```
    // Abstract class DAO Factory
public abstract class DAOFactory {


  // There will be a method for each DAO that can be
  // created. The concrete factories will have to
  // implement these methods.
  public abstract CustomerDAO getCustomerDAO();
    public abstract OrderDAO getOrderDAO();
  ...


public static DAOFactory getDAOFactory() {

    switch (getDBType()) {  //read a configuration file to find which type of database to connect to
      case ORACLE:
          return new OracleDbDAOFactory();
      case MYSQL:
          return new MySQLDbDAOFactory();
      ...
      default           :
          return null;
    }
  }

}

public enum DBType {
// List of DB types that will be supported by the factory

    ORACLE(1), MYSQL(2);

    private final int factoryType;

    private DBType(final int newFactoryType) {
        factoryType = newFactoryType;
    }

    public int getValue() {
        return factoryType;
    }
}
```

Code Segment 1.3: Creating Abstract Class

3. **Create concrete factory classes**: Once `AbstractFactory` is created, you have to create the concrete factory classes for each database implementation. Code Segment 1.4, shows the creation of the `OracleDbDAOFactory` class that extends `DAOFactory`. Each of these concrete factory classes will be responsible to create products (DAOs) pertaining to their database.

```
// Oracle concrete DAO Factory implementation
import java.sql.*;

public class OracleDbDAOFactory extends DAOFactory {
  public static final String DRIVER=
    "oracle.jdbc.OracleDriver";
  public static final String DBURL=
    "jdbc:oracle:thin:@[host]:[port]:[sid]";


  public static Connection createConnection() {
    // Use DRIVER and DBURL to create a connection
    // Recommend connection pool implementation/usage
  }
  public CustomerDAO getCustomerDAO() {
    // OracleDbCustomerDAO implements CustomerDAO
    return new OracleDbCustomerDAO();
  }
  public OrderDAO getOrderDAO() {
    // OracleDbOrderDAO implements OrderDAO
    return new OracleDbOrderDAO();
  }
  public AccountDAO getAccountDAO() {
    // OracleDbAccountDAO implements AccountDAO
    return new OracleDbAccountDAO();
  }
  ...
}
```

Code Segment 1.4: Creating Class `OracleDbDAOFactory`

In the same way, you have to create the `MySqlDbDAOFactory` class. Once this is done, you have designed the interface that interacts with the business layer of the application.

4. **Create Abstract Product**: Once the abstract classes and concrete classes have been created, you have to create the abstract types that the concrete products (DAO classes) will implement. This is shown in Code Segment 1.5.

```
// Interface that all CustomerDAOs must support
public interface CustomerDAO {
  public int insertCustomer(...);
  public boolean deleteCustomer(...);
  public Customer findCustomer(...);
  public boolean updateCustomer(...);
  public RowSet selectCustomersRS(...);
  ...
}
```

Code Segment 1.5: Creating Interface for the Business Layer

Finally, you have a sample DAO (Concrete Product).

5. Once the DAO interface is created, implement it using the code shown in Code Segment 1.6.

```
// OracleDbCustomerDAO implementation of the
// CustomerDAO interface. This class can contain all
```

```
// Oracle specific SQL statements.
// The client is thus shielded from knowing
// these implementation details.

import java.sql.*;

public class OracleDbCustomerDAO implements
    CustomerDAO {

  public OracleDbCustomerDAO () {
    // initialization
  }

  // The following methods can use
  // OracleDbCustomerDAOFactory.createConnection()
  // to get a connection as required

  public int insertCustomer(...) {
    // Implement insert customer here.
    // Return newly created customer number
    // or a -1 on error
  }

  public boolean deleteCustomer(...) {
    // Implement delete customer here
    // Return true on success, false on failure
  }

  public Customer findCustomer(...) {
    // Implement find a customer here using supplied
    // argument values as search criteria
    // Return a Transfer Object if found,
    // return null on error or if not found
  }

  public boolean updateCustomer(...) {
    // implement update record here using data
    // from the customerData Transfer Object
    // Return true on success, false on failure or
    // error
  }

  public RowSet selectCustomersRS(...) {
    // implement search customers here using the
    // supplied criteria.
    // Return a RowSet.
  }

  ...
      }
```

Code Segment 1.6: Code for Implementation

Advantages of the Abstract Factory pattern

The Abstract Factory pattern:

- Isolates clients from concrete implementation classes. The code only deals with the product interfaces; therefore, it can work with any user-defined concrete product classes.
- Makes it possible to interchange concrete implementations without changing the code that uses them
- Allows the use of multiple families of concrete implementation (also called concrete products)
- Enforces the use of products only from one family
- Maintains consistency among the product types used
- Helps to increase the flexibility of an application during run time and design time
- Simplifies the testing of the application

Disadvantages

The disadvantages of the Abstract Factory design pattern are:

- This pattern might introduce unnecessary complexity and extra work in the initial writing of code.
- The application can be difficult to debug with the high levels of separation and abstraction that the Abstract Factory pattern implements. Therefore, as in all software designs, the trade-offs must be carefully evaluated.

The abstraction has to be well thought-out while designing using this pattern. For example, the selectCustomersRS() method declared in the CustomerDAO interface may not hold good if you were to use an XML data source in future.

Factory Method and Abstract Factory Pattern

The Abstract Factory design pattern is similar to the Factory Method pattern, which is also a creational design pattern that uses factory designs to create objects. The main difference between these two design patterns is in the use. The **Abstract Factory** design pattern is used when families of products have to be created, and the application has to ensure that the client chooses products of the same family.

Factory Method design pattern is used when the client code needs to be decoupled from the concrete classes within an application.

Topic 3: Areas of Application

The areas of application development where the Abstract Factory design pattern is best suited are when:

- The application needs to be independent of how its products are created, composed, and represented.
- The system needs to be configured with one of the multiple families of products.
- A family of products needs to be used together.
- Developers need to provide a library of products and expose their interfaces but not the implementation.

Let us look at two examples of this design pattern implemented in JDK.

Abstract Factory from JDK

The JDK has the DocumentBuilderFactory class, which uses a service locator to find a concrete implementation of the DocumentBuilderFactory class (ie. xerces or some other parser), as shown.

```
// Uses service locator approach to find an implementor like xerces
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
Document document = builder.parse(...);
```

Java.awt.Toolkit

java.awt.Toolkit is another great example of using Abstract Factory pattern. Here, it uses the JVM implementation itself to provide the instance:

```
Toolkit toolkit = Toolkit.getDefaultToolkit();
```

The actual concrete class is different based on what OS you're on, and whether you're running in headless mode or not.

Lesson Summary

This lesson introduced you to the Abstract Factory design pattern which is used to decouple an objects implementation from its actual use in the application's UI. The Abstract Factory design pattern works by creating abstract objects and defining multiple concrete implementations for these objects. This makes the application flexible, portable, and platform independent. This lesson also introduced you to the areas of application of the Abstract Factory design pattern and real-life examples of implementing the Abstract Factory design pattern.

References

- Web references:
  - http://sourcemaking.com/design_patterns/proxy (Last accessed on 20 May, 2014)
  - http://www.oodesign.com/proxy-pattern.html (Last accessed on 20 May, 2014)

- Book references:
  - Freeman, Eric; Freeman, Elizabeth. (2010). *Head First Design Patterns.* O'Reilly Media.
  - Metsker, Steven John; Wake, William. C. (2006). *Design Pattern in Java.* Addison-Wesley-Longman Publishing.

Lesson 2: Adapter Design Pattern

Lesson Overview

An adapter is a device that helps to share the resources of two incompatible systems or machines. It works by converting the features of a system that can be shared, into a form that is compatible with another system. In the same manner, the Adapter design pattern is used to convert an existing interface to a form that is compatible with a client's system which is, otherwise incompatible. This helps developers to use exiting classes and functions, to satisfy multiple requirements without having to re-code entire portions of the application. In this lesson, you will be introduced to the Adapter design pattern, its implementations, and areas of application in software development.

Lesson Objectives

At the end of this lesson, you will be able to:

- Define the Adapter design pattern
- Implement the Adapter design pattern
- Use the UML diagram to represent the Adapter design pattern
- Identify the situations where you can use Adapter design pattern
- Give real-life application of Adapter design pattern

Topic 1: Adapter Design Pattern Introduction

In the field of application development, a structural design pattern is used to define relationships between components of an application. The Adapter design pattern is a structural design pattern and is similar to an AC adapter, where a particular feature of an object is made accessible to another object to perform the required operations. The Adapter design pattern is used for accessing the interface of an existing class from a different interface without having to modify the source code. In this way, the Adapter design pattern helps two incompatible interfaces work together by adding an additional converter interface between them. The advantage of the Adapter design pattern is that, a given interface or method can be re-used in multiple contexts with minor changes. Instead of altering the application's main code, developers can design an adapter interface which allows a particular class, to be implemented from any external or client interface.

The simplest way to understand the Adapter design pattern is to take the real world example of an electric adapter. This is shown in Figure 2.1.

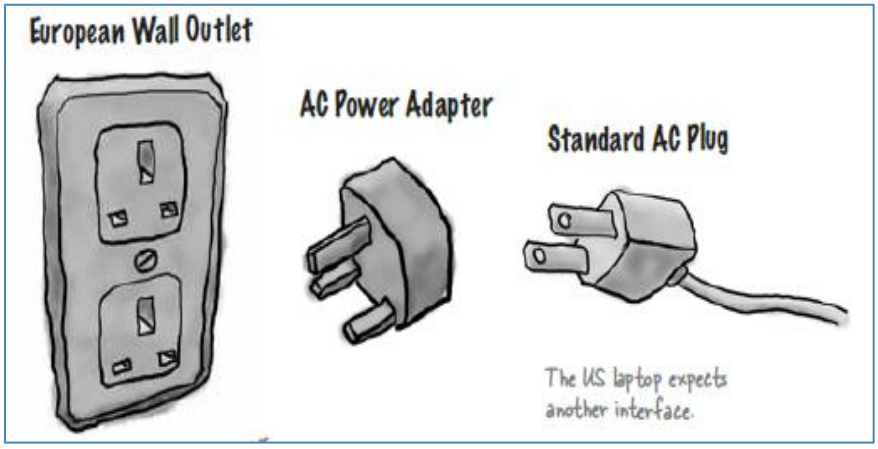


Figure 2.1: Electric Adapter

Anyone who has travelled to the United States of America and European countries, would have faced the difficulty of having to use an adapter to charge their electrical appliances such as a laptop. This is because the voltage and shape of the sockets available in the two regions are different. An AC power adapter is used as a bridge between two different electrical systems in order to perform the task of charging a device. This is the essence of the working of the Adapter design pattern.

For example, consider that a class has been designed to consume data in terms of bytes. But the data provider's output is in bits. So, you need an adapter that converts the bit stream to bytes. Or think of a software system that consists of numerous modules connected via interfaces. Sometimes, a needed service, such as a new vendor class library is available but offered through an interface that is incompatible with existing code. When there is a mismatch between the interface available and the interface needed, the adapter design pattern can be used to make the interface available and look and act like the interface needed, as shown in Figure 2.2.

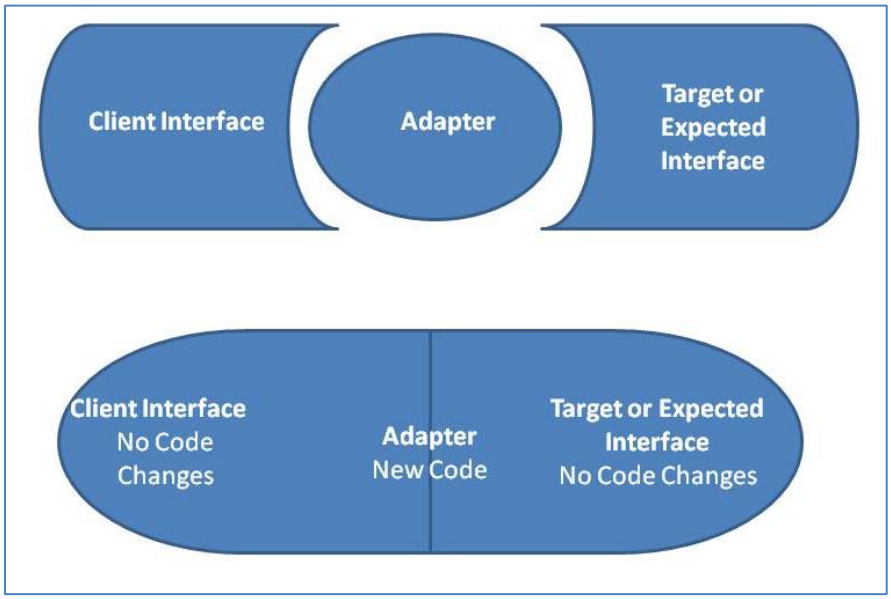


Figure 2.2: Adapter Design Pattern as a Bridge between Interfaces

The Adapter design 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. This is shown graphically in Figure 2.2.

The term "interface" 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. It refers 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.

This topic served as a brief introduction to the Adapter design pattern. In the next topic, you will learn about the implementation of the Adapter design pattern.

Topic 2: Implementing the Adapter Pattern

There are two ways of implementing the Adapter design pattern. The first way is to use composition while the other is to use inheritance.

With composition, the Adapter class implements the interface that the clients expect. At runtime, the adapter object is initialized with a reference to the service available. Clients call operations on the

adapter object and the adapter object forwards these calls to the service available. Figure 2.3 shows the visual representation of using adaptation via composition.

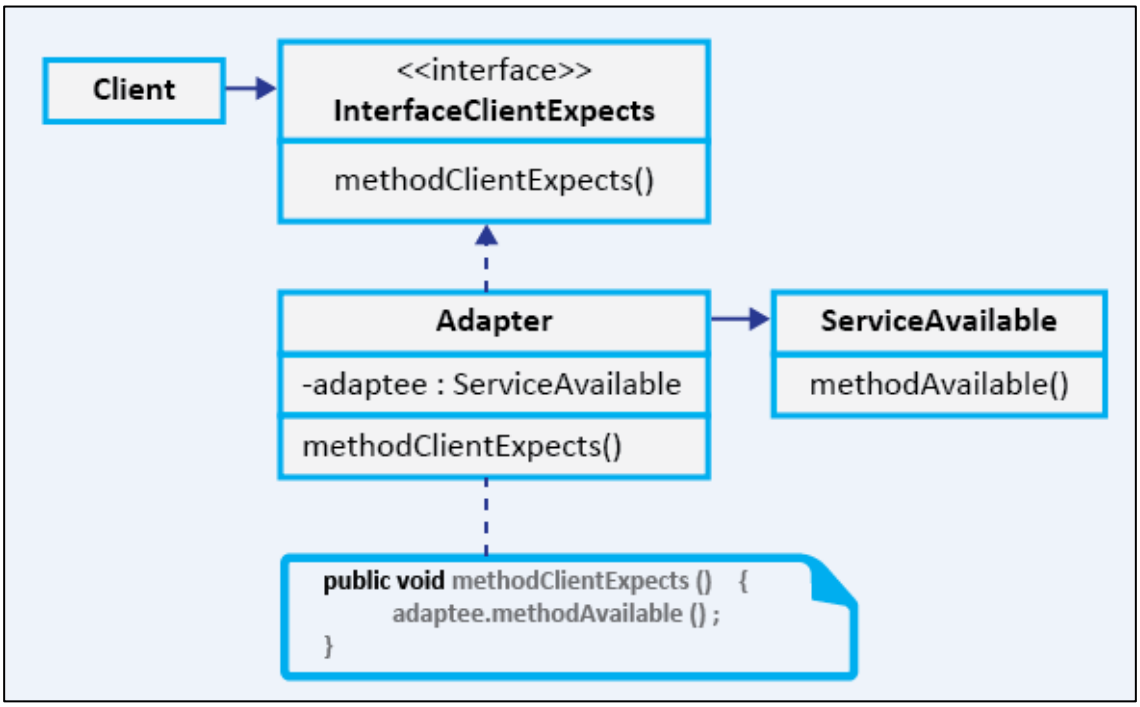


Figure 2.3: Adaptation by Composition

With inheritance, the Adapter class implements the interface the client expects (or inherits the interface from an existing class if the programming language being used allows multiple inheritance) and inherits from the class representing the service available. Clients call operations on the adapter object and the adapter object forwards these calls as necessary to the inherited operations of the available service.

Figure 2.4 shows the representation of adaptation using inheritance.

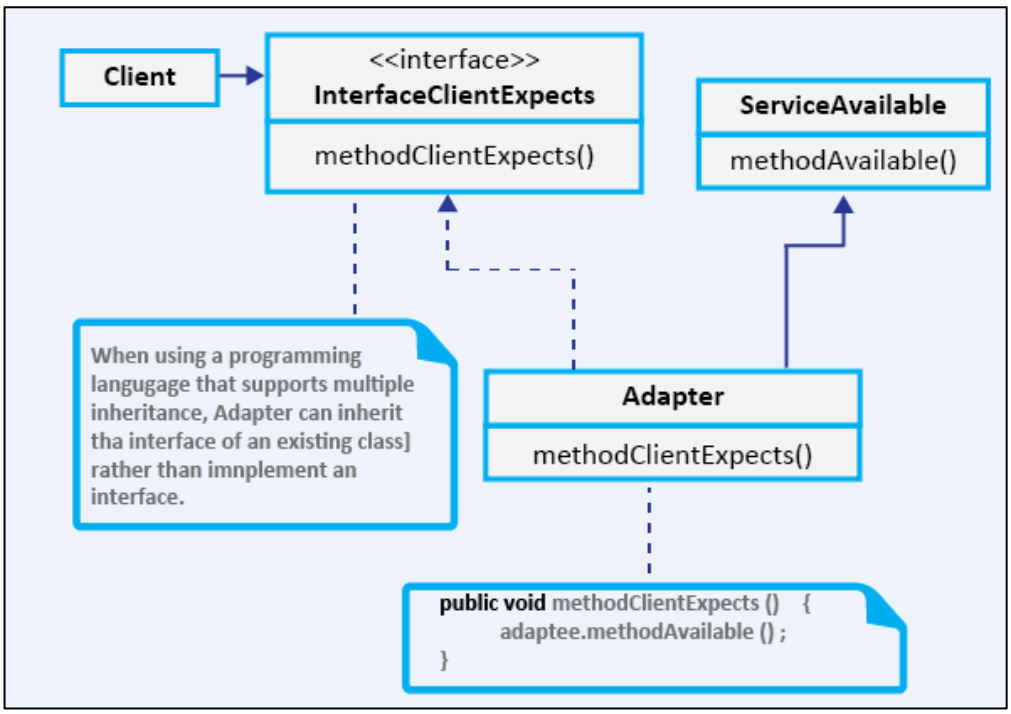


Figure 2.4: Adaptation using Inheritance

Topic 3: Case Study

To understand the working of the Adapter design pattern, let us look at the following case study. Let us assume that as a developer, you have to create a Java application that displays a computer's file system. As the file system data is displayed in a hierarchy, you can use Swing's `JTree` interface to control the display of data. Figure 2.5 shows an example of a computer's file system.

Figure 2.5: File System Display

The `JTree` interface is responsible only for the way is which the data is viewed by the user. The data actually is held in a separate object which conforms to the `TreeModel` interface. This is shown in Figure2.6.

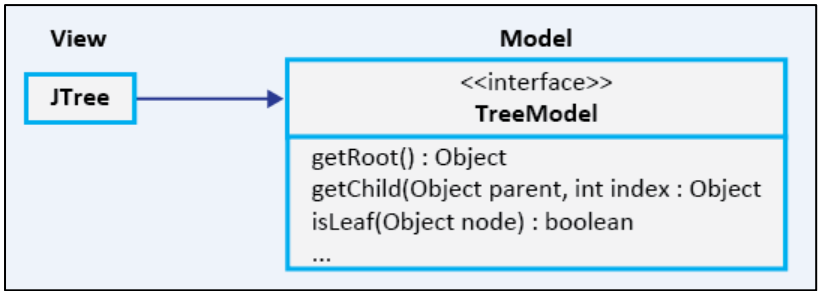


Figure 2.6: `TreeModel` Interface

The file system data to be displayed by the Java application being built resides in a tree of `File` objects. This is shown in Figure 2.7.

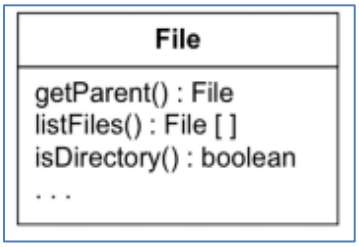


Figure 2.7: File System Data Held in `File` Object Interface

The `File` object represents a file or a directory in the file system tree. You can navigate through the tree to the location of the desired data by using methods defined for the `File` object. But the File object does not implement the `TreeModel` interface. In other words, the `File` object does not hold data the way `TreeModel` implementations do . To bridge together these two systems (interfaces), you can use the Adapter design pattern with an adapter class that converts the `File` object interface into the `TreeModel` interface. This is shown in the Figure 2.8.

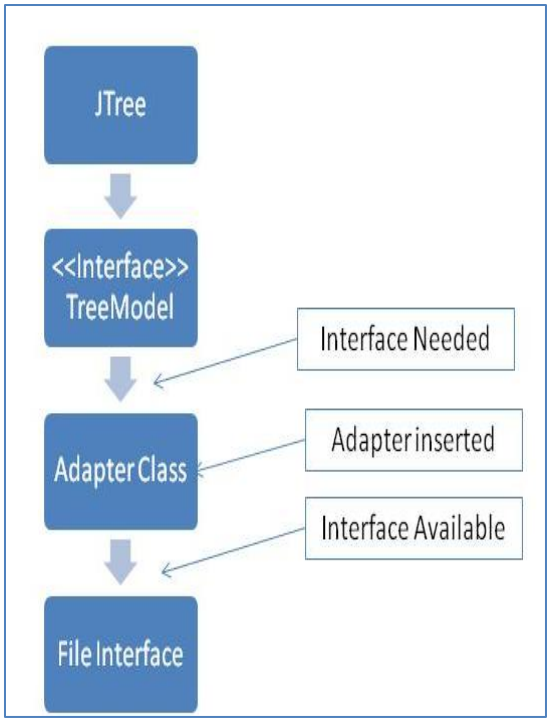


Figure 2.8: Adapter Design Pattern Used to Convert File Interface into TreeModel Interface

An adapter class can implement the `TreeModel` interface and wrap an instance of the `File` object. Since it implements `TreeModel`, an adapter class can serve as the data model for an instance of `JTree`, as shown in Figure 2.9.

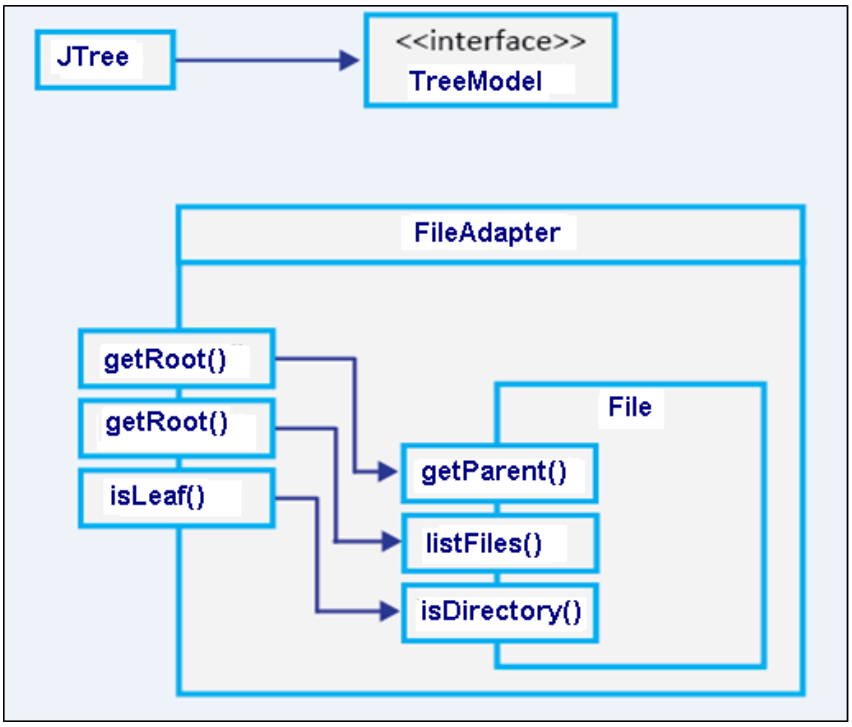


Figure 2.9: Adapter Class as Data Model

Requests made on the adapter class are delegated to the wrapped instance of File. The intent of the Adapter design pattern is to be useful in situations where an existing class provides a needed service but there is a mismatch between the interface offered and the interface clients expect.

In CodeSegment 2.1, FileAdapter implements the TreeModel interface and wraps an instance of File. Requests for data from JTree are forwarded to the wrapped instance of File. Note that not all of the methods in the interface TreeModel are implemented by the adapter. Specifically, methods needed to handle dynamic changes to the underlying data model are not implemented. Also note the extra code needed in the delegate methods to make the existing interface function like the expected interface. Delegation isn't always as simple as calling an identical method of a different name.

```
import java.awt.BorderLayout;

public class FileAdapterExample extends JFrame{


    private static final long serialVersionUID = 1L;

    public static void main(String args[]){
        new FileAdapterExample();
    }

    public FileAdapterExample() {
        JTree tree;
        FileAdapter treeNode;
        Container content = getContentPane();
        File f = new File("C:");
        treeNode = new FileAdapter(f);
        tree = new JTree(treeNode);
        content.add(new JScrollPane(tree),BorderLayout.CENTER);
        setSize(600, 375);
        setVisible(true);
    }
}
```

```
class FileAdapter implements TreeModel{
    private File root;
    public FileAdapter (File file) {
        root = file;
    }


    @Override
    public Object getChild(Object parent, int index) {

        File files[] = ((File)parent).listFiles();
        return files[index];
    }
    @Override
    public int getChildCount(Object parent) {
        File files[] = ((File)parent).listFiles();
        if (files== null) {
            return 0;
        }else{
            return files.length;
        }
    }
    @Override
    public int getIndexOfChild(Object parent, Object child) {

        return 0;
    }
    @Override
    public Object getRoot() {

        return root;
    }
    @Override
    public boolean isLeaf(Object node) {
        return !((File)node).isDirectory();
    }
    @Override
    public void removeTreeModelListener(TreeModelListener arg0) {


    }

    @Override
    public void addTreeModelListener(TreeModelListener arg0) {}

    @Override
    public void valueForPathChanged(TreePath arg0, Object arg1) {   }
}
```

Code Segment 2.1: Application using the Adapter Design Pattern

Some people tag the Adapter pattern as just a fix for a badly designed system, which didn't consider all possibilities. That said, it is best to avoid the use of the Adapter pattern if you are in control of both APIs

that need adapter, i.e. it is possible to refractor the APIs such that you would not need to create an adapter class.

Topic 4: Areas of Application

In many applications, the classes involved are designed to accept input in a specified format. The Adapter design pattern is used to ensure consistency in an application's input. The Adapter design pattern is used for developing applications when:

- There is a third party code, which cannot interact directly with the client application
- An application needs to extend an existing functionality, which is also being used by other components with a different intent
- A code depends upon sealed components for its working; the adapter object can be used to gain limited access to the component
- A new functionality is to be added to an existing application, which differs considerably from the original requirements

Topic 5: Disadvantages

The two main drawbacks of using the Adapter design pattern are:

- **Adapter code overhead**: Adapters are classes that need to be designed well and written explicitly. This is seen as an unnecessary overhead as adapters introduce a new set of utilities that need to be supported.
- **Out of context adapters**: If an existing API is not complete or the new interface to be used in an application is still under development, adapter classes remain out of context and unnecessary.

Lesson Summary

This lesson introduced you to the Adapter design pattern, which can be used in an application to bring together multiple interfaces that are not compatible with each other. You have seen that the Adapter design pattern works by using an adapter class. The objects of the adapter class carry references to the target interfaces, and convert these target interfaces into a format, acceptable by the client application. In this lesson, you have also been introduced to the implementation and uses of the Adapter design pattern in the world of application development.

References

- Web references:
  - http://sourcemaking.com/design_patterns/adapter (Last accessed on 8 June, 2014)
  - http://www.oodesign.com/proxy-pattern.html (Last accessed on 8 June, 2014)

- Book references:
  - Freeman, Eric; Freeman, Elizabeth. (2010). *Head First Design Patterns.* O'Reilly Media.
  - Metsker, Steven John; Wake, William. C. (2006). *Design Pattern in Java.* Addison-Wesley-Longman Publishing.

Lesson 3: Command Design Pattern

Lesson Overview

The Command design pattern is a behavioral design pattern where an object is used to encapsulate all information needed to call and run a method. The Command design pattern is used to manage the behavior of an application through its method implementation. In other words, this design pattern can be used to define not only what an application does, but also how and when an action is performed. This lesson aims at introducing the concepts of the Command design pattern, its implementation and uses while developing applications.

Lesson Objectives

At the end of this lesson, you will be able to:

- Define the Command design pattern
- Implement the Command design pattern
- Use the UML diagram to represent the Command design pattern
- Identify the situations where you can use the Command design pattern
- Give real-life applications of the Command design pattern

Topic 1: Command Design Pattern Introduction

Design patterns are beneficial in software development projects in a variety of ways. Firstly, design patterns accelerate the design phase of an object-oriented project. In addition, they also help in enhancing the quality of the software being developed and the productivity of the development team.

For effective software execution, it is important that the interface elements of the software are decoupled from the internal execution. A *Command pattern* enables you to achieve just that. A simple scenario to explain the benefits of the Command design pattern is that of a restaurant, where customers order food from a menu. A waiter takes the order, which is a command from the customer. Based on the time at which the order was placed, it is placed in a queue in the kitchen to be fulfilled by the chef. The chef has enough information in the form of recipes and procedures to prepare the order. The waiter has no knowledge of preparing the dish, and his role is limited to communicating orders to the chef.
A Command pattern allows you to enable complete decoupling between the sender and the receiver objects. Here, the sender object accepts the client's request, or the commands to be executed and passes it on to the receiver object without knowing about the interface of the receiver. The receiver object collects all necessary data and executes the methods to generate a result. Note that a *sender* is an object that invokes an operation while a *receiver* is an object that receives the request to execute the operation. The Command pattern provides flexibility and extensibility as it allows you to vary when and how a request is fulfilled.

In some situations, it is important that requests are issued without any information about the receiver or the operation to be performed. For example, consider an application that creates menus and adds actions to each menu item. To do this, the application can configure each menu item with a concrete instance of Command interface. When the user selects a menu item, it calls execute() method on its command and execute() carries out the operation. This way the menu items do not know which concrete command class they use. Instead, they just interact with the Command interface and call execute(), and remain unaware of the underlying concrete implementation. Concrete command classes help in storing the receiver of the request and executing operations on the receiver.

The decoupling feature of the Command pattern makes interface designing quite flexible. You just need to ensure that the object that issues a request need not know how to carry out the request. For example, you can share an instance of the same concrete Command to design both a menu and a push button interface. You can also replace commands in a dynamic manner to implement context-sensitive menus. The key to the Command pattern is the Command interface. The Command interface declares an interface for executing operations. In the simplest form, this interface includes an abstract execute operation (method). Concrete Command subclasses specify a receiver-action pair by storing the receiver as an instance variable and by implementing the execute() method to invoke the request. The receiver has the knowledge required to carry out the request.

By using this pattern to approach a problem, you will be able to make things organized and consistent, and remove repetitiveness of code.

When to Use the Command Design Pattern

Use the Command pattern in the following situations:

- When you need to perform parameterization of objects, such as MenuItems: Parameterization refers to a function that is registered somewhere to be called at a later point. Parameterization can be done in a procedural language with a callback function.
- When you need to specify, queue, and execute requests at different times: A Command object can have a lifetime independent of the original request.
- When you need to support undo operation: The Command's execute() operation can store the state for reversing its effects in the command itself. In that case, the Command interface must have an added undo operation that reverses the effects of a previous call to execute(). Executed commands are stored in a history list. Unlimited-level undo and redo is achieved by traversing this list backwards and forwards calling unexecuted() and execute(), respectively.
- When you need to support logging to re-apply the actions in the case of a system crash. A persistent log of changes, if kept, helps you to recover from a crash that involves reloading logged commands from the disk and re-executing them.

Components of Command Design Pattern

Let's look at the UML diagram of the Command design pattern, as shown in Figure 3.1.

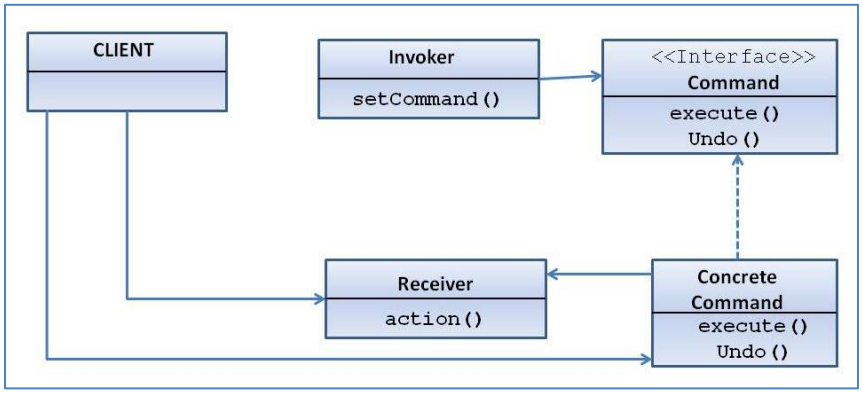


Figure 3.1: UML Diagram

Let us look at the components involved in this design pattern and understand their role in an application.

The components of the Command design pattern used to manage method invocations and executions are:

- **Client**: This component requests that the application perform a particular action, and associates the request to a specific receiver.
- **Command**: This is an interface, which is implemented by all Command objects. As you know, a command is invoked through its execute() method, which asks a receiver to perform an action.

- **Sender or invoker**: This is the object that holds a command and instructs it to perform a particular action.

- **Concrete Command**: This is an instance of the Command interface, which defines the binding between an action and a Receiver. The invoker makes a request by calling execute() and the Concrete Command carries it out by calling one or more actions on the Receiver.

- **Receiver**: This object knows the actual steps needed to perform a requested action.

Figure 3.2 shows the components and their relationship in the Command design pattern.

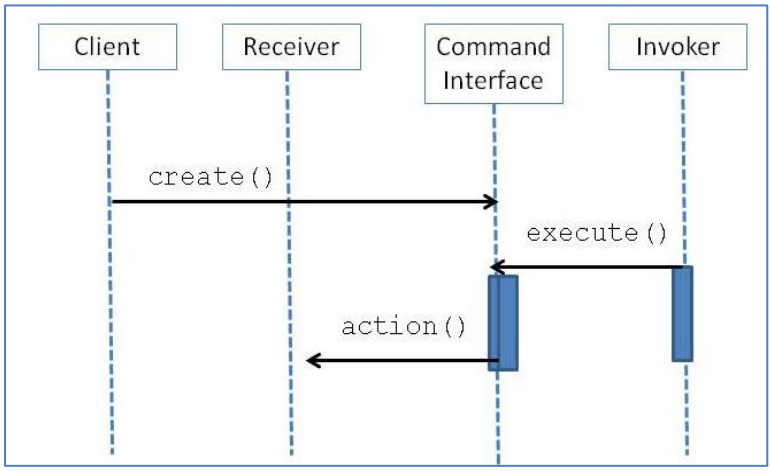


Figure 3.2: Component Relationship in Command Design Pattern

The components of the Command design pattern help the developers to implement request invocation and request execution in an application. The Command design pattern decouples the sender objects from the receiver objects, to keep the application flexible. The sender object is not aware of the steps needed to perform a requested action. The job of the sender object is to just transmit the request for an action, from the client to the receiver object. The Command interface receives the request from the sender object, and decides which of the concrete command classes needs to be used to satisfy the request. The concrete command classes store all possible actions in an application, and the appropriate class is invoked based on the client's request.

This decoupling of the sender object and receiver object gives developers a lot of flexibility in designing a User Interface (UI). Here, you can replace commands dynamically to implement context-sensitive menus in an application. This is supported in the Command design pattern, as objects that issue a request need not know how the request is being handled.

Drawbacks of the Command Design Pattern

Apart from offering benefits, like any other pattern, the Command design pattern may prove to be ineffective if used in the wrong context. The disadvantages of the Command design pattern are:

- **Increased number of classes**: Creating components with well-defined roles increases the number of classes for each individual command.

- **Difficult to debug**: The Command design pattern is difficult to debug, as there is a lot of indirection in request processing. Since command objects are created dynamically at runtime, the developers cannot be sure, which action is being invoked by which object.

These are the benefits and drawbacks of using the Command design pattern when used to develop applications. The next topic deals with the implementation of Command design pattern, using a code based example.

Topic 2: Case Study

Let us look at an example, to understand the use of the Command design pattern in an application. In this example, the application is used for working a remote control to regulate light settings in an automated environment. The remote control can be used for many actions such as turning the lights on, dimming the lights, setting timers for when the lights should turn on or off, and so on. Let us look at the methods used for switching the lights on or off. To write this method in the application, the following steps are performed:

1. **Create the `Command` interface**: The `Command` interface is at the core of the application and handles all request processing actions. Code Segment 3.1 shows the Command interface created for the lighting method.

```
//Command
public interface Command
{
    public void execute();
}


Now let's create two concrete commands. One will turn on the lights, another turns off lights:

//Concrete Command
public class LightOnCommand implements Command
{
    //reference to the light
    Light light;

    public LightOnCommand(Light light)
    {
        this.light = light;
    }

    public void execute()
    {
        light.switchOn();
    }

}
```

Code Segment 3.1: `Command` Interface

In Code Segment 3.1, the first line shows the creation of the Command interface. The next step is to define the concrete command methods for the application.

2. **Create the concrete commands**: In the application, the concrete command methods are those that actually carry out the actions necessary to execute client requests.
3. **Create a Receiver class that knows the actions to be performed.** Code Segment 3.2 shows the concrete command and Receiver for the remote control application.

```
//Concrete Command
public class LightOffCommand implements Command
{
    //reference to the light
    Light light;

    public LightOffCommand(Light light)
    {
        this.light = light;
    }

    public void execute()
    {
        light.switchOff();
    }

}


Light is our receiver class, so let's set that up now:
//Receiver
public class Light
{
   private boolean on;

   public void switchOn()
   {
      on = true;
   }

   public void switchOff()
   {
      on = false;
   }

}
```

Code Segment 3.2: Defining Concrete Command Methods

The next step in this process is to set the invoker object to accept the request from the client.

4. **Set invoker**: The invoker is an object that accepts a request from the client and forwards it to the `Command` interface. Code Segment 3.3 shows the invoker object called `RemoteControl`, being defined in the remote control application.

```
//Invoker
public class RemoteControl
{
    private Command command;

    public void setCommand(Command command)
    {
        this.command = command;
    }


    public void pressButton()
    {
        command.execute();
    }

}
```

Code Segment 3.3: Setting Invoker Object

In Code Segment 3.3, you can see that the invoker object `RemoteControl` does not perform any other operations, apart from passing on the client's request to the `Command` interface.

5. **Write main client code**: Code Segment 3. 4 shows the main client application that calls on the invoker to perform light on/off operations.

```
//Client
public class Client
{
    public static void main(String[] args)
    {
        RemoteControl control = new RemoteControl();

        Light light = new Light();

        Command lightsOn = new LightsOnCommand(light);
        Command lightsOff = new LightsOffCommand(light);

        //switch on
        control.setCommand(lightsOn);
        control.pressButton();

        //switch off
        control.setCommand(lightsOff);
        control.pressButton();

    }

}
```

Code Segment 3.4: Main Client Code

By executing the main client code, as seen in Code Segment 3.4, you can use the remote control application to turn the lights on or off in an automated environment. You can also extend the application to include concrete command methods to perform operations such as dimming the lights or setting timers.

Notice, when a button is pressed, all that the remote control object has to do is to call execute() on the corresponding command object. Also notice how you first pressed "light on" command and then later replaced it with a "light off" command. Since all Command objects implement the Command interface, the remote control isn't coupled with any one command object. Command objects can be dynamically changed and the remote control won't get impacted.

6. **Maintain State and Perform Undo Operation:** You talked about the unexecuted() method before. Let's say, you added capability in our remote control to control ceiling fans. To maintain state and perform undo, you need to add a new command, as shown in Code Segment 3.5. This command maintains the state to make undo possible.

```
public class CeilingFanHighCommand implements Command {
    CeilingFan ceilingFan;
    int prevSpeed;

    public CeilingFanHighCommand(CeilingFan ceilingFan) {
        this.ceilingFan = ceilingFan;
    }

    public void execute() {
        prevSpeed = ceilingFan.getSpeed();
        ceilingFan.high();
    }

    public void undo() {
        if (prevSpeed == CeilingFan.HIGH) {
            ceilingFan.high();
        } else if (prevSpeed == CeilingFan.MEDIUM) {
            ceilingFan.medium();
        } else if (prevSpeed == CeilingFan.LOW) {
            ceilingFan.low();
        } else if (prevSpeed == CeilingFan.OFF) {
            ceilingFan.off();
        }
    }
}
```

We've added local state to keep track of the previous speed of the fan.

In execute, before we change the speed of the fan, we need to first record its previous state, just in case we need to undo our actions.

To undo, we set the speed of the fan back to its previous speed.

Code Segment 3.5: Command Class for Maintaining State and Performing Undo Operation

Topic 3: Command Pattern Applied in Java

You will find that the Command pattern very commonly applied in JDK itself and various frameworks. Some popular implementations are discussed here.

Java Runnable Interface
You can find the Command pattern applied in Java's implementation of multithreading. Classes that implement Java's Runnable interface encapsulate knowledge of what to do on a thread. So creating a Runnable object essentially creates a Command object. See the following code.

```
Runnable backgroundTask = new Runnable(){
    public void run(){
       System.out.println("Hello World");
    }
};
```

This command object is then passed to a thread object in the following manner:
`Thread controller = new Thread( backgroundTask );`

You then start the thread as follows:
`controller.start();`

At this juncture, the controller asks the Command object to do whatever it does by calling run(). This is exactly what Command pattern advocates that the Invoker (the Thread object) does not actually know what the Concrete Command object (backgroundTask) is going to do.

JUnit
In JUnit, test cases are represented as command objects. In JUnit 3, each test class has to extend from the TestCase abstract class that, in turn, implements the interface Test, as shown:

```
public interface Test {
   public void run();
}

public abstract class TestCase implements Test {
  public abstract void run();

... }
```

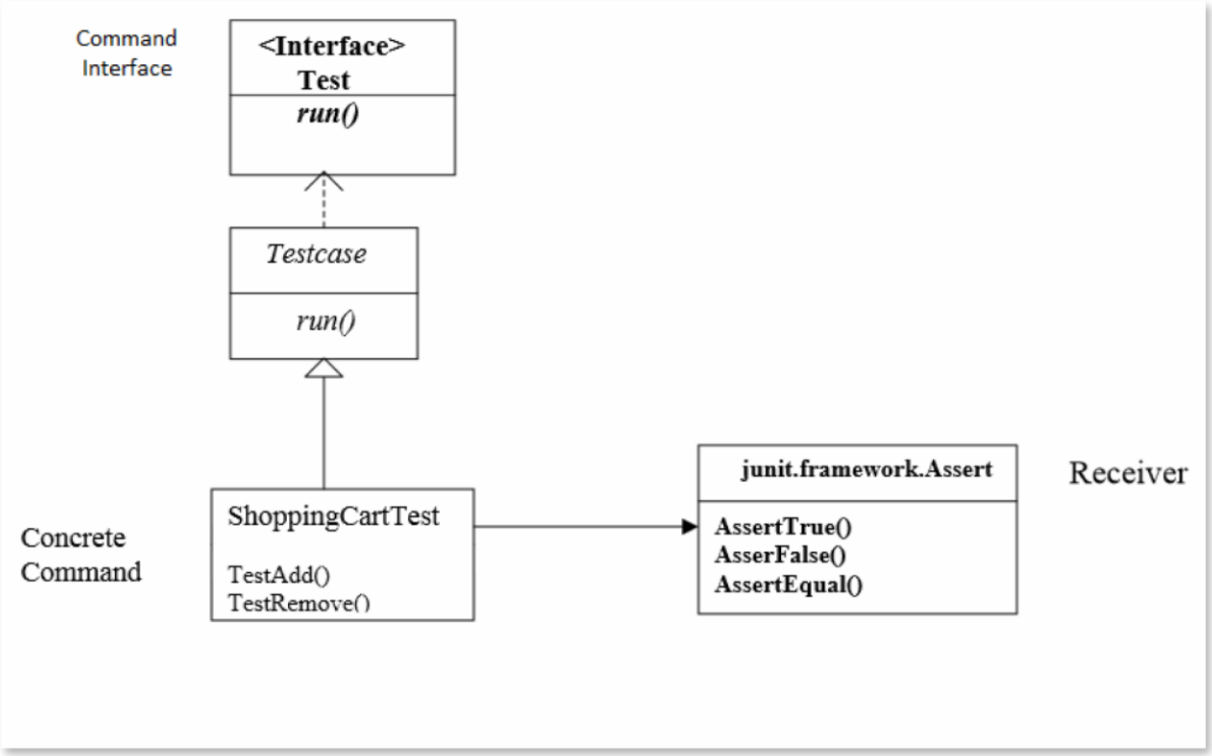


Figure 3.3: Command Pattern in JUnit

As you can make out from the figure, each test class you create is a concrete command that indirectly implements the Test command interface. The run() method declared in the interface is like the execute() of the Command pattern. In this case, the Test Suite is the invoker that executes each test class (commands) using the run () method that is similar to the execute(), method.

Topic 4: Areas of Application of the Command Pattern

The Command design pattern has many applications. Some of the real life uses of the Command design pattern are:

- **Designing Graphic User Interface (GUI) buttons and menu items**: In Java Swing based application development, any action (referred to by the `Action` object) being performed is a command object that may also be associated with an icon, keyboard shortcut, or a tooltip text. The `Action` object can be used to initialize a toolbar button or menu item.

- **Networking**: The Command design pattern can be used to send whole command objects, across the network to be used on other systems or machines. This feature is used to control player actions in multiplayer gaming applications.

- **Parallel processing**: Applications contain commands that are written as tasks that share resources and execute in many threads at the same time.

- **Developing progress bars**: A progress bar displays the completion status of an application's task. The status appears either as a colored bar or in percentage. You would have seen a progress bar when installing an application.

  A progress bar is used when an application has a sequence of commands to be executed where each command has a particular time limit. The `Command` interface contains a `getEstimateDuration()` method to calculate the time taken to complete the tasks and display the progress bar.

- **Creating configuration wizards**: A wizard handles several configuration tasks for a single action, which is triggered when a user clicks the "Finish" button of the wizard dialog box. Here, using a command object is an easy way to separate the user interface from the application code.

You are now familiar with some real life uses of the Command design pattern. Clearly, the Command design pattern can be used in situations where user interface can be decoupled from the main application code. This ensures the flexibility and efficiency of the application code as the `Command` interface redirects a client request to the appropriate concrete command classes.

Lesson Summary

This lesson introduced you to the Command design pattern, which can be used to develop applications that need a clear demarcation between a request invocation and request handling. This separation is achieved by decoupling the codes that accept a request from the client and the code that actually performs the actions necessary to generate appropriate response corresponding to the client's request. The Command design pattern is similar to a token that can be passed around within the application and invoked later. When developing Object-Oriented Programming (OOP) based applications, the Command design pattern allows developers to vary the when and how of fulfilling a request. Therefore, a Command design pattern provides flexibility as well as extensibility to an application.

References

- Web references:
  - http://sourcemaking.com/design_patterns/command (Last accessed on 10 June, 2014)
  - http://www.javaworld.com/article/2077569/core-java/java-tip-68--learn-how-to-implement-the-command-pattern-in-java.html (Last accessed on 10 June, 2014)

- Book references:
  - Freeman, Eric; Freeman, Elizabeth. (2010). *Head First Design Patterns.* O'Reilly Media.
  - Metsker, Steven John; Wake, William. C. (2006). *Design Pattern in Java.* Addison-Wesley-Longman Publishing.

Lesson 4: Template Method Design Pattern

Lesson Overview

Every application has a set of well-defined algorithms that help in meeting business requirements of the application. The structure of application algorithms can be defined using a programming skeleton. The design pattern used to define the behavior of the programming skeleton in a method is called the Template Method design pattern. The Template Method design pattern is used to manage the details of selecting and implementing methods in an application. This lesson aims at familiarizing you with the concept of the Template Method design pattern and its areas of application in the software development industry.

Lesson Objectives

At the end of this lesson, you will be able to:

- Define the Template Method design pattern
- Implement the Template Method design pattern
- Use the UML diagram to represent the Template Method design pattern
- Identify the situations where you can use Template Method pattern
- Give real-life application of Template Method design pattern

Topic 1: Template Method Design Pattern Introduction

In the Template Method design pattern, a method called the template method is used to provide a foundation for an application's algorithm. The template method defers some of its steps to a subclass within the application so that the deferred steps can be redefined according to the context of their use. The term template refers to a preset format such as HTML template for document creation. Similarly, the Template Method design pattern has a pre-defined structure for an application, consisting of abstract steps.

Thus, the Template Method design pattern is used to define an algorithm, though the steps involved in executing the algorithm can be defined in subclasses of the application. The Template Method design pattern can be thought of as the starting point for developing an application that can be used in multiple contexts. Instead of recreating an application for use in different scenarios, the same basic template can be applied with a few modifications to the abstract steps of the application.

A real world example where the Template Method design pattern is used is applications that help in drawing architectural plans. A template for a particular floor plan is created and then the same template can be customized by adding multiple features based on client requirements.

The Template Method design pattern is a behavioral pattern as it is used to manage algorithms, relationships, and responsibilities between objects in Object-Oriented Programming (OOP).

Let us now look at the benefits of using the Template Method design pattern while creating an OOP-based application.

Topic 2: Case Study

Consider two classes—CSVParser and DBDataParser— in an application. CSVParser reads and parses data from a CSV file and generates HTML output. DBDataParser reads data from a database and generates HTML output in the same format that CSVParser does.

One concern here is that the two classes seem to be containing duplicate code. Duplicate code complicates maintenance and increases the risk of introducing bugs when making changes. It is important to be able to recognize duplication and understand how to eliminate it in whatever form it may take. Though the two classes are similar, they are not exact duplicates of each other. You simply cannot replace one with the other because the details of each are different. Refer to Code Segment 4.1.

```
class CSVParser{
        public void readFiles() {
           ... read CSV files
        }
        public void parseFiles() {
           ... parse each row and column into objects
        }
        public void writeData(){
           ... output the data in HTML format
        }
        public void uploadToFtp(){
           ... upload the output file to FTP
        }
}
```

```
class DBDataParser{
        public void init() {
           ... open DB connection
        }
        public void loadData() {
           ... Loop through ResultSet..
        }
        public void writeData(){
           ... output the data in HTML format
        }
        public void uploadToFtp(){
           ... upload the output file to FTP
        }
}
```

Code Segment 4.1: Duplicating Algorithms

As you noticed from Code Segment 4.1, the structure of the algorithm for reading and parsing data is being duplicated. Also, the writeData() and uploadToFTP() functions are exact copies in both files.

Now imagine if, in the future, you wanted to add a sendMail() or a auditLog() function, you would have to update both the parsing procedures, thus duplicating even more code.
By analyzing the individual steps, you can extract the general algorithm for parsing and generating output, as shown in Figure 4.1.

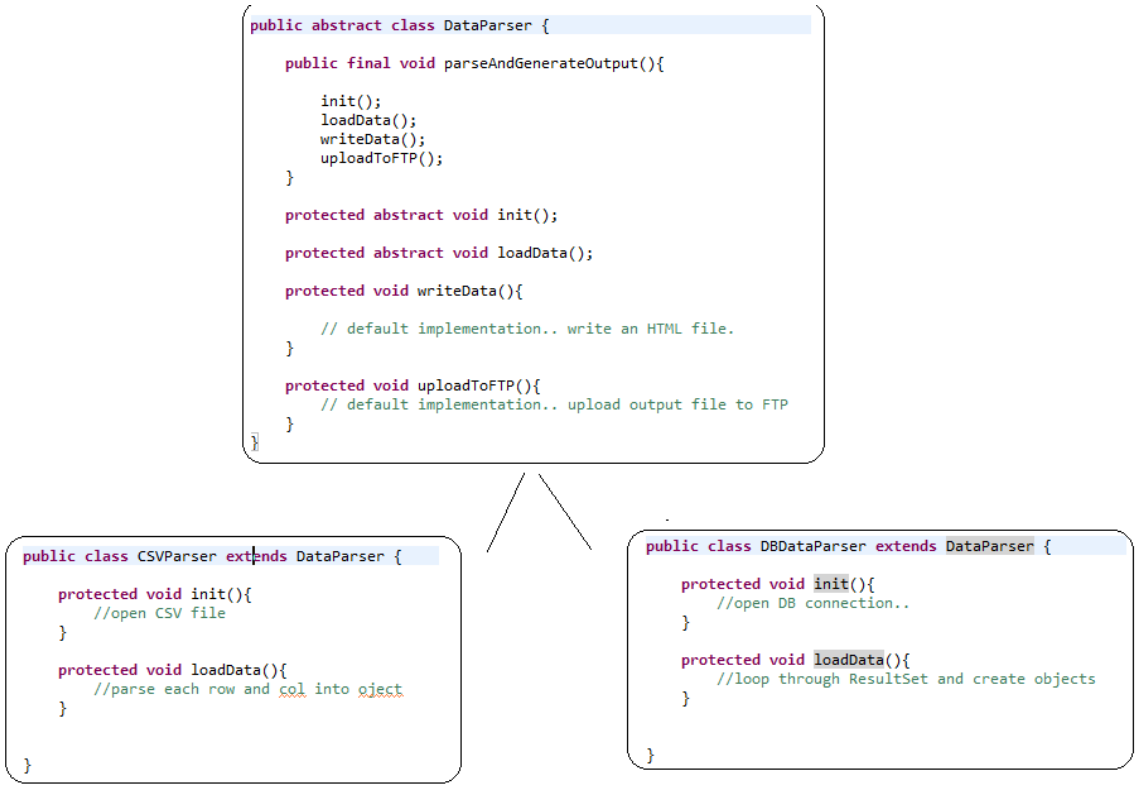


Figure 4.1: Extracting the General Algorithm from the Two Parsers

As you see, the algorithm in both classes is much more symmetric and easier to maintain. The common functionality is in the base class while the unique functionality of each parser is placed in the two

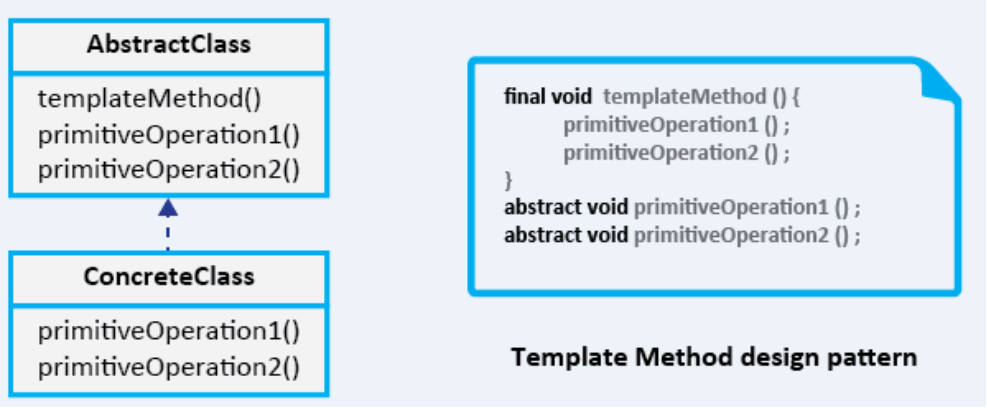
subclasses. It would be easier to maintain if you were to add another parser later. Thus, the Template Method design pattern helps to eliminate repetition in the algorithm structure. With the Template Method design pattern, the structure of an algorithm is represented once with variations on the algorithm implemented by subclasses. The skeleton of the algorithm is declared in a template method in terms of overridable operations. Subclasses are allowed to extend or replace some or all of these operations.

You saw how the algorithm or basic steps for parsing, outputting, and uploading is the same when the goal is to parse CSV or database. Only the detailed activities within the steps are different. The invariant parts of the procedure are declared in an abstract base class. The variant parts are declared in subclasses. The sequence of steps for parsing, writing the output file, and uploading the file to FTP are defined in the template method parseDataAndGenerateOutput(). Subclasses—one for each type of parser—override methods called from parseDataAndGenerateOutput() to implement variations on the basic procedure.

The following guidelines were used in the design of this example:

1. Methods expected to have unique implementation in most subclasses were declared abstract.
2. Methods expected to share the same implementation in many subclasses were declared concrete and given default implementation.
3. Methods needed for only a small fraction of subclasses were declared concrete and given empty implementation.

As you know now, that with the Template Method design pattern, an algorithm is broken down into primitive operations. The skeleton of the algorithm is defined in a template method that resides in an abstract base class. Concrete subclasses implement variations on the algorithm by overriding specific operations. Let's look at the class diagram of Template Method.



Template Method design pattern

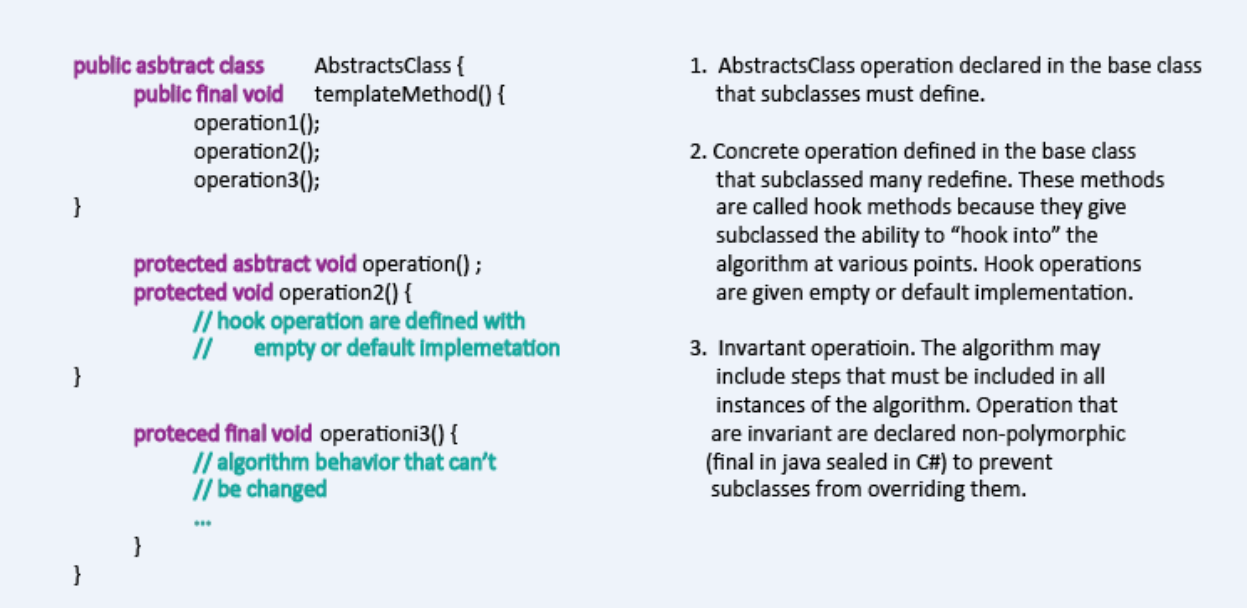


Figure 4.2: UML Class diagram for Template Method Design Pattern

Components of the Template Method Design Pattern

The Template Method design pattern has the following components in its structure:

- **Abstract class**: Defines the template method that should be overridden during execution. The template method defines the structure of the algorithm.

- **Concrete classes**: Implements the abstract operation of the super class and carry out subclass-specific steps of the algorithm. They can also override methods from super class if default behavior is not required.

Consider another example of the Data Access Object (DAO) framework that uses the Template Method pattern. Refer to Figure 4.3.

```
public abstract class AbstractDAO {
        public final void execute(){
                //This is our template method.
                connectToDB();
                loadData();
                outputResults();
                closeDBConnection();
        }
        //Loading data for each dao will be slightly different. Let the objects
        public abstract void loadData();
        //Outputting data for each dao will be slightly different. Let the objec
        public abstract void outputResults();

        //Opening the database is the same. We can define this here.
        private void connectToDB(){
                System.out.println("\n*****Connecting to database*****");
        }

        private void closeDBConnection(){
                System.out.println("******Closing database*****\n");
        }
}
```

```
public class Employees extends AbstractDAO{

        @Override
        public void loadData() {
                System.out.println("Loading data for employees");
        }

        @Override
        public void outputResults() {
                System.out.println("Outputting results for employees");
        }
}
```

```
public class Customers extends AbstractDAO{
        @Override
        public void loadData() {
                System.out.println("Loading data for customers");
        }

        @Override
        public void outputResults() {
                System.out.println("Outputting results for customers");
        }
}
```

Figure 4.3: Template Method Applied in DAO Classes

You will notice how in each DAO, only the loadData and outputResults methods are defined. The loadData() method in each subclass would have a select statement to call the database as per the DAO class it is in and the outputResults method would then iterate through the results and perform specific operations.

Code Segment 4.2 shows the client class.

```
1 public class DAOImpl {
2               public static void main(String[] args) {
3                               Employees employees = new Employees();
4                               Customers customers= new Customers();
5
6                               employees.execute();
7                               customers.execute();
8               }
9 }
```

Code Segment 4.2: Client Class

Notice that you are just calling the execute method here and not any other method directly. You allow the template method (execute) to call the functions within the DAOs. The benefit is that you don't need to explicitly take care of opening and closing database connections. The design handles that.

In this way, the Template Method design pattern is used to manage the application's algorithm and the relationship between the template method and the derived classes.

Topic 3: Areas of Application of the Template Method Design Pattern

The Template Method design pattern allows applets to define initialization methods in the base abstract class. An applet is a small program that runs a web page and any applet must have other applets as subclasses. Web page related methods that depend on the user's input are defined in subclasses of the applet. Code Segment 4.3 shows the use of the Template Method design pattern in an applet program.

```
1  public class MyApplet extends Applet {
2      String message;
3
4      public void init() {
5          message = "Hello World, I'm alive!";
6          repaint();
7      }
8
9      public void start() {
10         message = "Now I'm starting up...";
11         repaint();
12     }
13
14     public void stop() {
15         message = "Oh, now I'm being stopped...";
16     }
17
18     public void destroy() {
19         // applet is going away...
20     }
21
22     public void paint(Graphics g) {
23         g.drawingString(message, 5, 15);
24     }
25 }
```

c

Code Segment 4.3: Applet Program Using Template Method Design Pattern

In Code Segment 4.3,

- Line 4: The `init()` hook allows the applet to execute the methods required to initialize the applet the first time.
- Line 6: Shows the `repaint()method` which is a concrete method in the Applet class that lets upper-level components know that the applet needs to be redrawn.
- Line 9: Shows the `start`() hook that allows the applet to do something when the applet is just about to be displayed on the web page.
- Line 14: Shows that if the user goes to another page, the `stop()` hook is used, and the applet can do whatever it needs to do to stop its actions.
- Line 18: Shows the `destroy()` hook which is used when the applet is going to be destroyed after the browser pane is closed.

- In JUnit 3, the Template Method design pattern is used to define the structure for all test cases. This is done by using the Template Method design pattern to:
  - Set up a test fixture

- Execute the test code
- Clean the fixture

Code Segment 4.4 shows an example JUnit test that implements the Template Method design pattern.

```
public abstract class TestCase {

    public void run() {          <-- Template method
      setUp();
      runTest();
      tearDown();
    }
    protected void setUp() { }
    protected void runTest() {..... }
    protected void tearDown() { }
}
```

Code Segment 4.4: Example JUnit that Implements Template Method Design Pattern

While writing a JUnit test case, the subclasses can be defined to override the basic methods of setup and teardown. With this, you have seen the various areas of application of the Template Method design pattern in the field of application development.

Servlets API - Servlet Life Cycle methods
init()
service() and
destroy() are also based on the Template method.

Developers are expected to extend HttpServlet and write meaningful code in – doGet, doPost. In this case, the template method service() defines a template for handling HTTP requests.

Using the Template Method Pattern

The Template Method pattern is used when:

- You have steps for an algorithm, but implementations of the steps are varied.
- You want to implement common code in base class with variations in the subclass. The pattern also helps in avoiding code duplication since common code goes into the super class.

Here are a few guidelines that you need to follow while implementing the Template Method pattern:

- Template method in super class follows “the Hollywood principle”: “Don't call us, we'll call you.” This refers to the fact that instead of calling the methods of super class from subclasses, the methods from the subclasses are called in the template method from super class.

- Template methods in super class are not supposed to be overridden, so make it final.
- Hook methods should represent optional parts of the algorithm. A hook method is a concrete method that appears in the abstract base class that has an empty method body, i.e.

  public void hook() {}

  Subclasses are free to override hook methods but do not have to since they provide a method body, albeit an empty one. Thus, hook methods should represent optional parts of the algorithm.

- Template methods are techniques for code reuse because with this, you can determine common behavior and defer specific behavior to subclasses.

Benefits of the Template Method Design Pattern

Using the Template Method design pattern to define the structure of an application that has to be used in multiple contexts provides developers with the following benefits:

- **Implementing varying behavior**: The Template Method design pattern delegates the actual action steps an application performs to its subclasses. By overriding the template method, the subclasses can be used to customize a general application to a specific condition. This allows developers to satisfy client needs with tailor-made solutions.

- **Avoiding code duplication**: The general structure of the application is defined in the Template Method design pattern. Once the structure is defined, it can be reused by implementing variations in subclasses. This eliminates the need to recreate duplicate code to perform the steps occurring commonly.

- **Controlling subclass definition**: In an application, some actions are used in all contexts and some steps are defined to context specifications. The Template Method design pattern allows developers to control points at which the subclasses take over the execution of steps.

Lesson Summary

In this lesson, you were introduced to the Template Method design pattern. This is a behavioral design pattern used to design applications where a common algorithm is implemented with multiple variations based on the context of use. The Template Method design pattern uses an application's inheritance to implement variations of the algorithm. This lesson also familiarized you with the uses of the Template Method design pattern in the software development industry.

References

- Web references:
  - http://sourcemaking.com/design_patterns/template_method (Last accessed on 12 June, 2014)
  - http://www.oodesign.com/template-method-pattern.html (Last accessed on 12 June, 2014)

- Book references:
  - Freeman, Eric; Freeman, Elizabeth. (2010). *Head First Design Patterns.* O'Reilly Media.
  - Metsker, Steven John; Wake, William. C. (2006). *Design Pattern in Java.* Addison-Wesley-Longman Publishing.

Lesson 5: Façade

Lesson Overview

The **Façade pattern** belongs to the category of Structural design pattern. A Façade is an object that provides a simplified access to a fairly large and complicated body of code, such as a class library.

Lesson Objectives

At the end of this lesson, you will be able to:

- Define the Façade pattern and it's benefits.
- Identify situations where you can use the Façade pattern
- Implement solutions using the Façade pattern
- Give real-life applications of the façade pattern

Topic 1: Façade Pattern Defined

The Façade pattern hides a complicated system and provides an easy access to it. Hence clients can make use of the façade to interact with the underlying complex system in a simplified manner. Façade is a structural pattern as it adds an interface to existing system to hide its complexities.

This pattern involves a single wrapper class which contains a set of methods required by the client. These methods access the system on behalf of the Façade client and hide the implementation details. The Façade design pattern is particularly useful when a system is very complex or difficult to understand because of the presence of a large number of interdependent classes or unavailability of source code.

A simple example that can help you to understand the Façade design pattern is the functioning of the microwave oven. Any microwave oven consists of components like transformer, capacitor, magnetron, and a wave guide. To use the microwave to heat or cook food, its components need to be activated in a given sequence. Having separate switches or knobs for each component will be cumbersome to use as you would have to remember all possible combinations and sequences to use for different operations such as, heating, cooking.. To simplify these operations, microwaves have a control panel. You can use the buttons provided in this control panel to select the power setting, duration of the operation and, type of operation for specific food items. The combination of input signals received by pressing these buttons will activate the microwave's components in the correct sequence needed for an operation. The control panel is a façade that provides you with a simple way to use the microwave oven, making it a useful and dependable tool in your kitchen.

Other examples for Façade are one-stop bill payment shops, a support desk in a big organization which takes all kinds of support requests, and so on.

A good example of the Façade design pattern in the Java Class Library is the JOptionPane class. The JOptionPane class simplifies and unifies the low-level interface for creating dialog boxes in Java.

To appreciate the benefits of using JOptionPane to create dialog boxes, consider the amount of code needed to create and show a simple dialog box using the base Swing classes in the Java Class Library, see Figure 5.1.

```
JPanel panel = new JPanel();
JLabel messageLabel =
        new JLabel("Press OK to continue.");
JButton OKButton = new JButton("OK");
final JDialog customDialog = new JDialog();
panel.add(messageLabel);
panel.add(OKButton);
customDialog.getContentPane().add(
        panel,BorderLayout.CENTER);
customDialog.pack();
OKButton.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e){
                customDialog.dispose();
        }
});
customDialog.setVisible(true);
```

Figure 5.1: Creating a Simple Dialog Box using the Base Swing Classes

The result of executing this code is a plain looking dialog box, as shown in Figure 5.2:

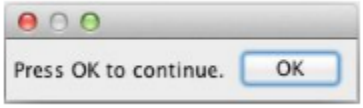


Figure 5.2: Plain Dialog Box

In comparison, you can use the JOptionPane class in one line of code to create a similar dialog box:

JOptionPane.showMessageDialog(null, “Press OK to continue.”);

Note that it takes just one line of code to create a dialog box with JOptionPane. The code is simple and the resulting dialog box has a standard look and feel that is much more visually appealing than the handcrafted one, as shown in Figure 5.3:

Figure 5.3: Dialog Box using JOptionPane

JOptionPane is a Façade class, as shown in Figure 5.4. A Façade class simplifies and unifies access to a larger more complex set of classes belonging to a subsystem. JOptionPane simplifies access to the Java subsystem for creating and showing dialog boxes. JOptionPane itself doesn't do all the work; it simply delegates the requests to existing classes.

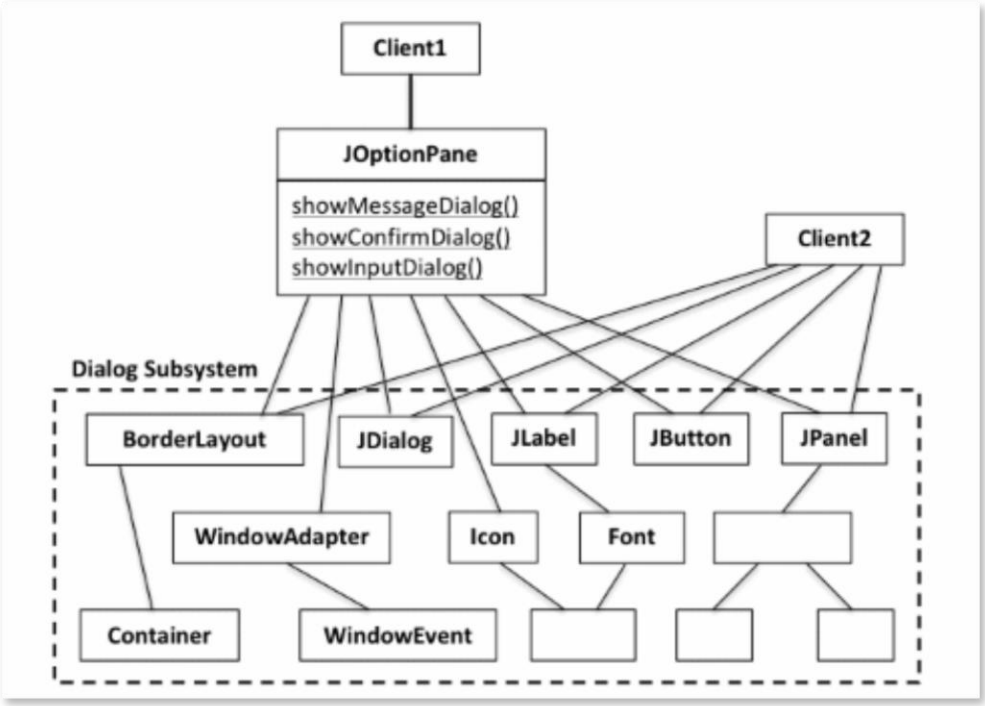


Figure 5.4: JOptionPane Façade

Notice also from Figure 5.4, that JOptionPane doesn't encapsulate the low-level support for creating dialog boxes. As such, clients are free to go around JOptionPane and can create dialog boxes directly using the primitive support provided in the Java Class Library. However, for simplified development of dialog boxes that has a standard look and feel, you could use JOptionPane.

Intent & Benefits of the Façade Pattern

The intent of the Façade design pattern is to "provide a single point of access to a set of classes in a subsystem. Think of Façade as a higher-level interface that makes the subsystem easier to use". There are three benefits of using the Façade design pattern:

- Unification: A Façade class defines a single point of access for clients. Clients deal with one class as opposed to a large interface or complex set of interfaces.
- Abstraction and simplification: Façade provides a level of abstraction for the clients. Façade methods are convenient to developers and provide higher-level functionality that may otherwise be tedious and painful to implement.
- Decreased coupling: A Façade class decouples clients from the details of a subsystem. This allows extra flexibility when developing applications.

The Façade design pattern is one of the easiest to understand and implement. It doesn't use inheritance or polymorphism and there are no interfaces to implement. You can implement the Façade design pattern using only one class. The class includes operations that provide a high-level interface onto an existing subsystem. The operations are implemented by delegating to lower-level components (usually classes) in the existing subsystem, as shown in Figure 5.5.

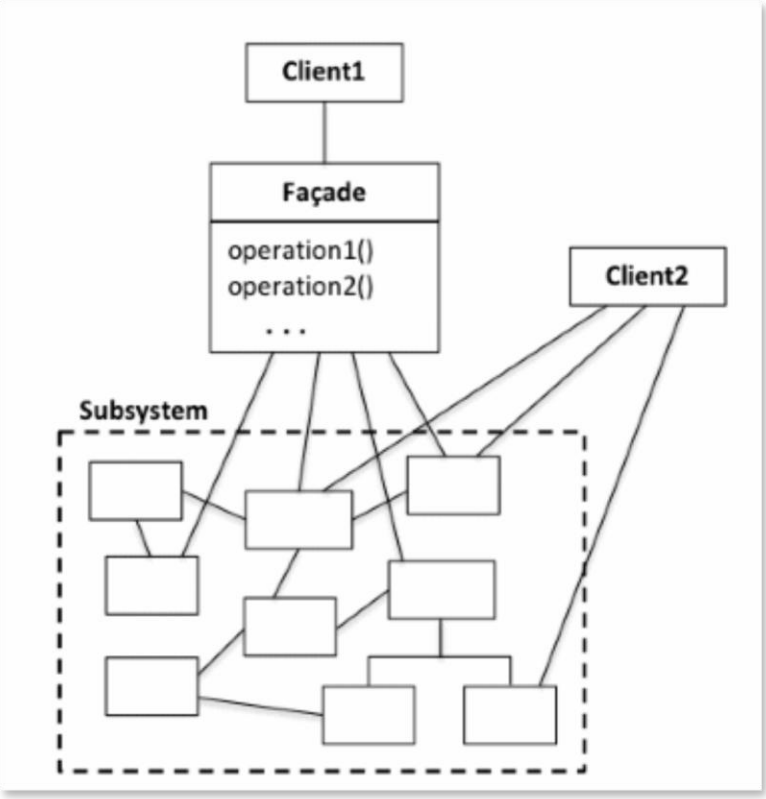


Figure 5.5: Façade Design Pattern

Topic 2: Façade Implementation

The basic steps needed to implement the Façade design pattern are as follows:

1. Identify a simple and unified interface for your application's components.
2. Design a Façade class that wraps the subsystem, is able to process the complexity of the subsystems, and delegates the user request to the appropriate methods.
3. Couple the client-side application only to the Façade class.
4. Try to determine if multiple Façades would add value to your application.

Façade as a Layered Architecture

The Façade design pattern is often used with the layered architecture style. One way to reduce coupling between layers is to have clients access the services of a layer through one or more Façade classes, as shown in Figure 5.6.

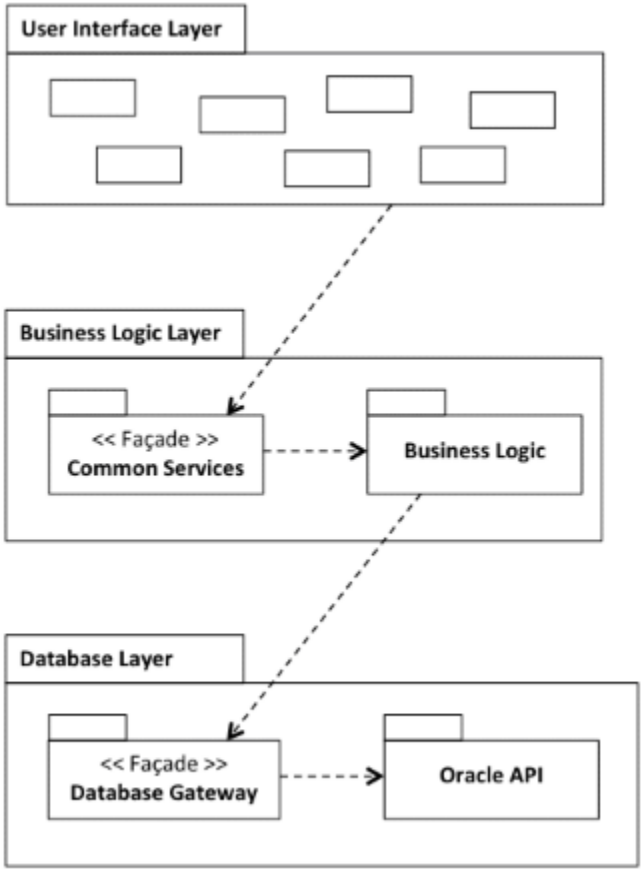


Figure 5.6: Façade, as a Layered Architecture

As you see, in Figure 5.6, Façade can be used to hide unnecessary details in a multi-layered architecture. In addition, the Façade class will undergo a change if the underlying layer changed while the other layer(s) will not be impacted.

Examples of Façade in popular APIs

The Simple Logging Façade for Java (SLF4J) serves as a simple Façade or abstraction for various logging frameworks, such as java.util.logging, logback, and log4j. SLF4J allows the end-user to plug in the desired logging framework.

Example usage:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class HelloWorld {
  public static void main(String[] args) {
    Logger logger = LoggerFactory.getLogger(HelloWorld.class);
    logger.info("Hello World");
  }
}
```

Façade and Adapter

The Adapter and Façade patterns may look similar but their intent is different. The Adapter design pattern helps to allow the interface of an existing class to be used from another interface (that clients use).

On the other hand, the Façade pattern is used when an easier and simplified access is required to a complex library.

Summary

With this, you come to the end of this lesson. In this lesson, you have learned about the Façade design pattern and where you can implement this pattern for developing OOP-based applications. You can use the Façade design pattern in any OOP application, where you need to deny access to a complex internal structure to a user and allow high-level operations using a common interface.

References

- Web references:
  - http://sourcemaking.com/design_patterns/Façade (Last accessed on 17 May, 2014)
  - http://thisblog.runsfreesoftware.com/?q=Façade+Design+Pattern+UML+Class+Diagram+Example(Last accessed on 17 May, 2014)
  - http://www.tutorialspoint.com/design_pattern/Façade_pattern.htm (Last accessed on 17 May, 2014)

- Book references:
  - Metsker, Steven John; Wake, William C. *Design Pattern in Java* (2006). Addison-Wesley Publishing.
  - Eckel, Bruce. *Thinking in J*ava (2006). Prentice Hall Publication

Lesson 6: State Design Pattern

Lesson Overview

Design patterns that recognize common communication patterns between objects in an application are called behavioral design patterns. In this lesson, you will be introduced to a specific type of behavioral design pattern called the State design pattern, which is used to define object communications or behaviors based on the current state of the objects involved in an application. The advantage of using the State design pattern is that it eliminates the need for using if-else loop iterations in an application's code. The State design pattern also provides a systematic and loose-coupled approach to achieve the necessary changes in an object's behavior triggered by changes to its state. In this lesson, you will learn to implement the State design pattern using a real life application.

Lesson Objectives

At the end of this lesson, you will be able to:

- Define the State design pattern
- Identify situations where the State design pattern can be used
- Implement the State design pattern
- Represent the State design pattern graphically using the Unified Model Language (UML) class diagram
- Discuss real-life applications of the State design pattern

Topic 1: Introduction to the State Design Pattern

The State design pattern is a behavioral pattern in which an object changes its behavior based on any changes made to its state. The State design pattern is a precise and simple way to control object changes at runtime, without having to use monolithic conditional statements in an application. Before going further into the State design pattern, let us briefly look at object states in Object-oriented Programming (OOP).

Object States in OOP

OOP-based development revolves around the objects present in an application where every object has:

- A state: The state of the object is stored in its fields or variables.
- A behavior: The behavior of an object is seen through its methods or functions.

In an application code, methods are used to change the internal state of an object and define object-to-object interaction. 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, it implies 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.

Intent of the State Pattern

The State pattern intends to:

- Allow an object to alter its behavior when it’s internal state changes.
- Implement a state machine that is object-oriented.

There are many cases where an object's behavior depends on its current state. For example, when you're sleepy, it's hard to think. However, thinking is easy when you have consumed coffee and chocolate.

Imagine a DisplayText class that needs to output text in such a way that the first line is in lowercase, the next two lines are in uppercase, followed next line in lowercase, and so on. Let’s look at how this can be implemented using the State pattern. The DisplayText class is displayed in Code Segment 6.1.

```
class DisplayText {
    private DisplayState displayState;

    DisplayText() {
        setState(new LowerCaseState());
    }
```

```
    /**
     * Setter method for the state.
     * Normally only called by classes implementing the State interface.
     * @param newDisplayState the new state of this context
     */
    void setState(final DisplayState newDisplayState) {
       displayState = newDisplayState;
    }

    public void display(final String name) {
       displayState.display(this, name);
    }
}
```

Code Segment 6.1: The DisplayText Class

You will notice how DisplayText delegates the task of displaying to DisplayState. Two classes (`LowerCaseState` and `MultipleUpperCaseState`) implement the DisplayText interface. In other words, these are the two states of DisplayText. As you will see in Code Segment 6.2, each of the states implement the behaviour (display() method) in it's own way.

```
public interface DisplayState {

      void display(DisplayText displayText, String text);

}
```

```
public class LowerCaseState implements DisplayState {

      @Override
      public void display(DisplayText displayText, String text) {
            System.out.println(text.toLowerCase());
        displayText.setState(new MultipleUpperCaseState());

      }

}
```

```
public class MultipleUpperCaseState implements DisplayState {

      /** Counter local to this state */
    private int count = 0;

      @Override
      public void display(DisplayText displayText, String text) {

            System.out.println(text.toUpperCase());
```

```
        /* Change state after MultipleUpperCaseState's display() gets invoked twice
*/
        if(++count > 1) {
            displayText.setState(new LowerCaseState());
        }
      }

}
```

Code Segment 6.2: The Two States

Notice how the display() method of `LowerCaseState` and `MultipleUpperCaseState` changes the state (from small and upper case and vice-versa) of DisplayText `after displaying`. As a result, the behaviour of the `displayState` variable in DisplayText seems to be changing each time the state changes. That is the essence of the **state pattern**.

Next, let's see the class to test the DisplayText class. Refer to Code Segment 6.3.

```
public class Demo {

      public static void main(String[] args) {

            final DisplayText sc = new DisplayText();

        sc.display("Monday");
        sc.display("Tuesday");
        sc.display("Wednesday");
        sc.display("Thursday");
        sc.display("Friday");
        sc.display("Saturday");
        sc.display("Sunday");


      }

}
```

Code Segment 6.3: Testing the DisplayTest Class.

With the above code, the output of the main() from Demo should be:

monday
TUESDAY
WEDNESDAY
thursday
FRIDAY
SATURDAY
sunday

Topic 2: Understanding the State Pattern

Let's now understand the State pattern with another example of a simple controller for a vending machine and map out the states and transitions of a vending machine, as shown in Figure 6.1.

Assume you start with the machine full of soda cans and waiting for a coin. Let's call this state as the NO_COIN state. You can insert a coin. The transition in this case will have a HAS_COIN state. If you eject the coin, the state would be a NO_COIN state, but you could also turn the crank and move it to the SOLD state. When the soda can is dispensed, it goes back to the NO_COIN state, but if the machine is out of soda cans, it goes to the SOLD_OUT state and remains in this state until the machine is refilled. Now, the common way to implement the state machine is to implement it as a set of **state** constants.

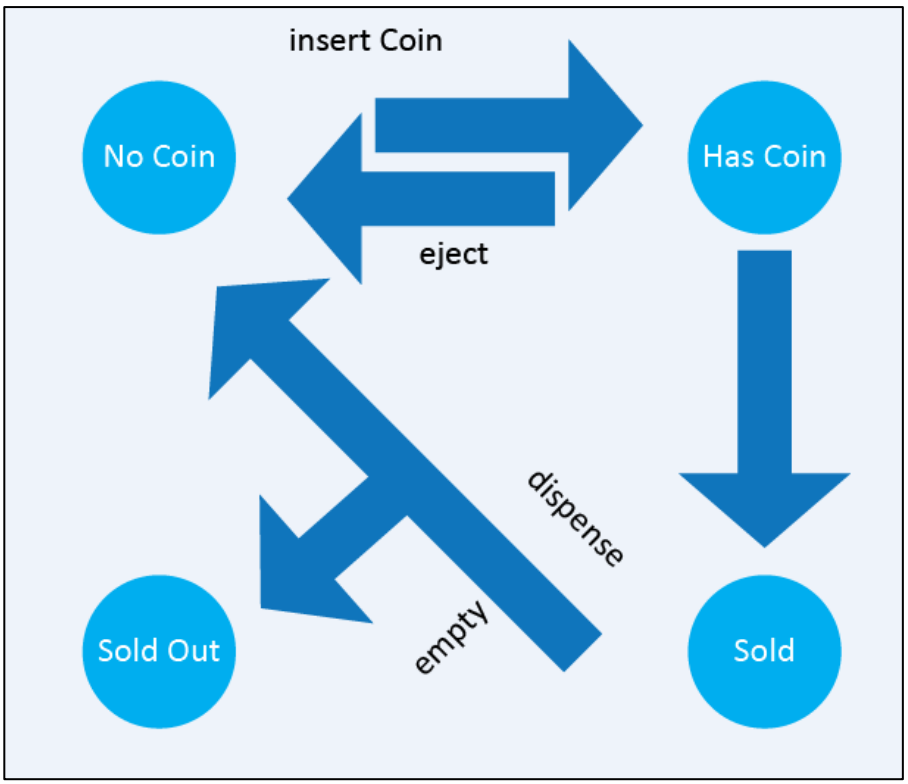


Figure 6.1: A Simple State Machine

Next, write a method for each transition, such as turnCrank(), as shown in Code Segment 6.4.

```
public void turnCrank() {
    if (state == SOLD) {
        System.out.println("You can't turn crank twice");
    } else if (state == SOLD_OUT) {
        System.out.println("ERROR, machine sold out");
    } else if (state == NO_COIN    ) {
        System.out.println("Insert a coin    first");
    } else if (state == COIN_INSERTED ){
        state = SOLD;
        dispense();
    }
}
```

Code Segment 6.4: Method for Each Transaction

The turnCrank() method takes the respective action depending upon the state. This way, you will need to create a method for each transition and each method will check what state the object is in to carry out the respective task . Now, let's say that you have new state, Winner where 1 in 100 customers get a free soda bottle. Then, you will have to add another state as a constant and then open up every existing method to handle the new state. In this case, the code becomes complex each time you add a new state. Here are the problems associated with this approach of checking for the state in every method.

Any changes require way too many changes to the code.

It is not object-oriented.

It is difficult to understand and get a gist of all the states and transitions by looking at the code in the class.

Having all state and transition logic in one class violates the open-closed principle.

So, let's take another approach and use the State pattern to solve the design. Convert the states into objects so that the states implement a common State interface, which declares a method for each state transition. Refer to Figure 6.2.

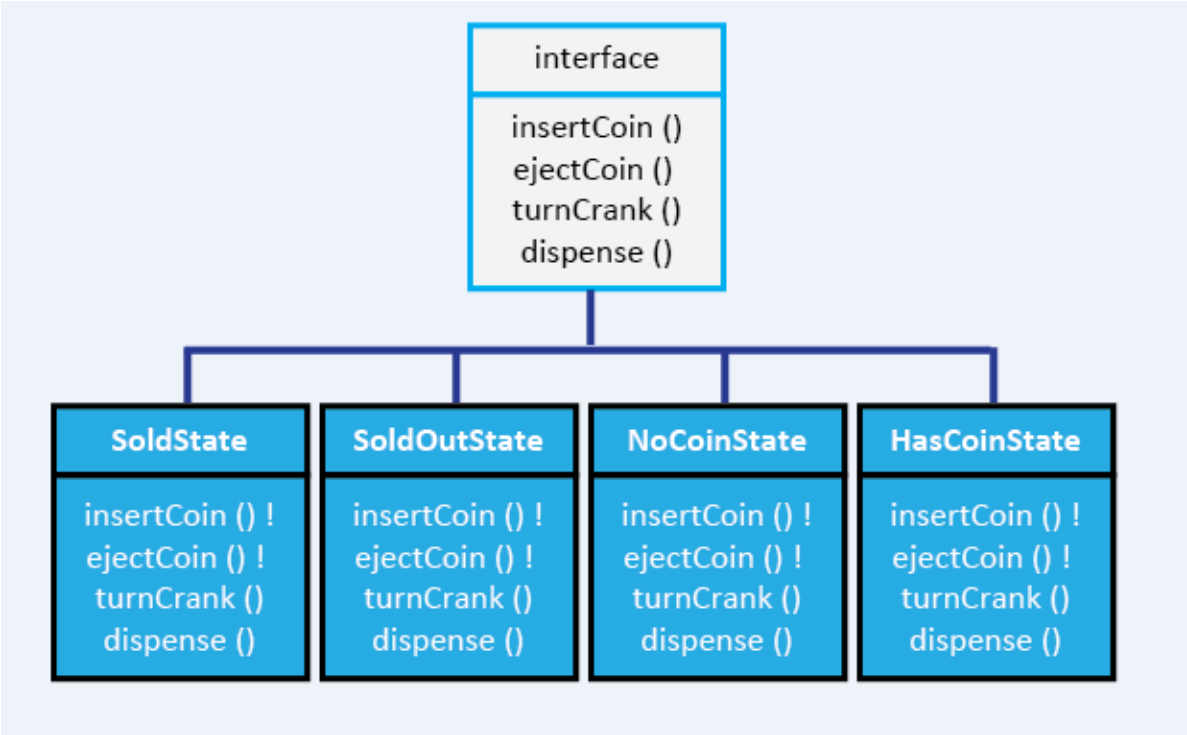


Figure 6.2: Each State as a Different Class

Each state class will be responsible for the behavior of the machine when it is in that state. For example, when the machine is in NO_COIN state, the insertCoin(), ejectCoin(), turnCrank() and dispense() states will all be implemented to handle that particular state. In addition, to handle a transition to a new state, the insertCoin() method in NoCoinState will transition the state from NoCoinState to the HasCoinState. This way, the conditional code in the soda can machine will be removed. You will be able to delegate to the current state and let the 'state' handle the behavior. Refer Code Segment 6.5 to see the VendingMachine class.

```
public class VendingMachine {
    ...
    State state = noCoinState;

    public void insertCoin() {
        if (state == NO_COIN) {
            ...
        }
        ...
    }

    public void insertCoin()    {
        state.insertCoin();
    }

    ...
}
```

Code Segment 6.5: The VendingMachine Class

Now that you have implemented the State pattern, let's take a look at the class diagram of State pattern, as shown in Figure 6.3.

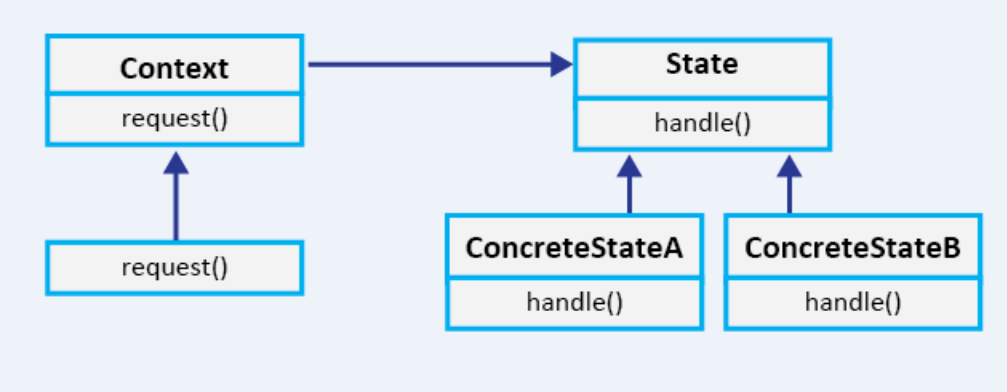


Figure 6.3: State Pattern Class Diagram

The Vending machine is the Context. It's the class that manages all the states. As per the State pattern, when a request is made on the context, i.e. a method is called on the Context, the context delegates the

request to a state. Every state class implements the same interface, i.e. the State interface. So, no matter what kind of request is made on the Context, the Context can delegate the request to any of the states. That means that the Context doesn't need to know how the States are implemented, so you would have a loose coupling between the Context and the States.

Key Points of the State Pattern

The key points of the State pattern are:

The pattern encapsulates state into separate classes.

The context delegates to the current state to handle requests: The state is responsible for handling the request and also for changing the state if and when necessary. The context is not responsible. For example, in the vending machine, if you get a request insertCoin(), the context just calls the insertCoin() method on the current state. If the current state is the NoCoin state, then the insertCoin() method will handle the transition to the HasCoin state.

Once a request is handled, the current state may change: Since each state has a different behavior, from the context's perspective, it appears that the current state is always changing behavior (as if it was changing its class). But, since all state classes implement the State interface, all state classes are similar.

By using the State pattern, you are adhering to a few design principles that you may already be aware of.

Encapsulate what Varies: We're encapsulating what is varying by making each state responsible for its own behavior.

Favor Composition over Inheritance: By separating that logic from VendingMachine class, you are favoring Composition over Inheritance. Rather than creating subclasses of VendingMachine to handle different states and then overriding the default behavior, you are instead composing the VendingMachine with a set of different states and putting the behavior for each state into its own class. The benefit of composition here is that the Context can delegate all requests to the current state without having to worry about which state that is.

Keep a class closed for modification but open for extension: With the State design, you can always add a new State with minimum amount of change required for any existing classes.

Topic 3: Implementing the State Pattern

Let's look at our Context class, i.e. VendingMachine.java, as shown in Code Segment 6.6.

```
public class VendingMachine {

	State soldOutState;
	State noCoinState;
	State hasCoinState;
	State soldState;

	State state = noCoinState;
	int count = 0;

	public VendingMachine(int numberSodas) {
		soldOutState = new SoldOutState(this);
		noCoinState = new NoCoinState(this);
		hasCoinState = new HasCoinState(this);
		soldState = new SoldState(this);

		this.count = numberSodas;

	}

	public void insertCoin() {
		state.insertCoin();
	}

	public void ejectCoin() {
		state.ejectCoin();
	}

	public void turnCrank() {
		state.turnCrank();
		state.dispense();
	}

	void releaseSoda() {
		System.out.println("A soda bottle comes out ...");
		if (count != 0) {
			count = count - 1;
		}
	}

	int getCount() {
		return count;
	}

	void refill(int count) {
		this.count = count;
		state = noCoinState;
	}
```

```
      void setState(State state) {
            this.state = state;
      }
    public State getState() {
        return state;
    }

    public State getSoldOutState() {
        return soldOutState;
    }

    public State getNoCoinState() {
        return noCoinState;
    }

    public State getHasCoinState() {
        return hasCoinState;
    }

    public State getSoldState() {
        return soldState;
    }

      public String toString() {
            StringBuffer result = new StringBuffer();
            result.append("\n The Soda Vending Machine");
            result.append("\nInventory: " + count + " sodas");
            if (count != 1) {
                  result.append("s");
            }
            result.append("\n");
            result.append("Machine is " + state + "\n");
            return result.toString();
      }
}
```

Code Segment 6.6: Context Class

Each state of the machine is represented by a separate state object, which is initialized in the VendingMachine constructor. You pass the VendingMachine instance to the states so that they can get and set the current state in state variable. The default state is the NoCoin state. Methods such as insertCoin(), ejectCoin(), and turnCrank() simply delegate the responsibility for handling the request to the current state.

Let's look at the State interface.

```
public interface State {

      public void insertCoin();
      public void ejectCoin();
      public void turnCrank();
      public void dispense();
}
```

The State interface is simple and has just four methods that each of the States implement.

Let's now look at the NoCoinState state, as shown in Code Segment 6.7, which is the state that the VendingMachine is initialized with.

```
public class NoCoinState implements State {
    VendingMachine vendingMachine;

    public NoCoinState(VendingMachine vendingMachine) {
        this.vendingMachine = gumballMachine;
    }

       public void insertCoin() {
              System.out.println("You inserted a coin");
              vendingMachine.setState(vendingMachine.getHasCoinState());
       }

       public void ejectCoin() {
              System.out.println("You haven't inserted a coin");
       }

       public void turnCrank() {
              System.out.println("You turned, but there's no coin");
        }

       public void dispense() {
              System.out.println("You need to pay first");
       }

       public String toString() {
              return "waiting for coin";
       }
}
```

Code Segment 6.7: NoCoinState

You will notice how the constructor of the NoCoinState takes an instance of VendingMachine and sets it to a VendingMachine class attribute. The only valid action in the NoCoin state is insertCoin(). When that is called, you see a message, 'You inserted a coin' and then the state transitions to the HasCoin state. Let's look at the HasCoin state, as shown in Code Segment 6.8.

```
public class HasCoinState implements State {
       VendingMachine vendingMachine;

       public HasCoinState(VendingMachine vendingMachine) {
              this.vendingMachine = vendingMachine;
       }

       public void insertCoin() {
              System.out.println("You can't insert another Coin");
```

```
        }

        public void ejectCoin() {
                System.out.println("Coin returned");
                vendingMachine.setState(vendingMachine.getNoCoinState());
        }

        public void turnCrank() {
                System.out.println("You turned...");
                vendingMachine.setState(vendingMachine.getSoldState());
        }

    public void dispense() {
        System.out.println("No soda dispensed");
    }

        public String toString() {
                return "waiting for turn of crank";
        }
}
```

Code Segment 6.8: HasCoin State

The HasCoin state also implements from the State interface like all the other State classes do. In this state, you can either eject the coin or you can turn the crank. If you eject the coin, you see a message – 'coin returned' and then the current state is set back to the NoCoin state.

If you turn the crank, then you see the message 'You turned' and the current state is set to the SoldState. Assume that you want to get a soda in the SoldState. To see how you can get a soda, let's take a look back at VendingMachine.

```
public void turnCrank() {
                state.turnCrank();
                state.dispense();
        }
```

When you turn the crank in the HasCoin state, the turnCrank() method in the VendingMachine is called and this method:

Delegates to the current state's turnCrank() method

Calls on the dispense method on the current state

The turnCrank() method of HasCoin changes the state from the HasCoin state to the SoldState so the dispense method is actually called on SoldState. For example, in SoldState, you can dispense the Soda by first calling the releaseSoda() method on VendingMachine and then checking if there any more sodas left. If sodas are there, you can set the state to the NoCoin state. If there are no Sodas left, you can set the state to the SoldOut state, as shown in Code Segment 6.9.

```
public class SoldState implements State {

    VendingMachine vendingMachine;

    public SoldState(VendingMachine vendingMachine) {
        this.vendingMachine = vendingMachine;
    }

      public void insertCoin() {
            System.out.println("Please wait, we're already giving you a soda");
      }

      public void ejectCoin() {
            System.out.println("Sorry, you already turned the crank");
      }

      public void turnCrank() {
            System.out.println("Turning twice doesn't get you another soda!");
      }

      public void dispense() {
            vendingMachine.releaseSoda();
            if (vendingMachine.getCount() > 0) {
                  vendingMachine.setState(vendingMachine.getNoCoinState());
            } else {
                  System.out.println("Oops, out of soda!");
                  vendingMachine.setState(vendingMachine.getSoldOutState());
            }
      }

      public String toString() {
            return "dispensing a soda";
      }
}
```

Code Segment 6.9: SoldOut State

By designing the vending machine to use the State pattern, you have localized the behavior of each state into it's own class and hence eliminated the need for conditional statements to manage the state. This makes the code easier to maintain and more flexible.

In the next topic, you will look at look at the implementation of the State design pattern and the benefits of using it.

Topic 4: Advantages and Disadvantages of Using the State Design Pattern

The advantages of using the State design pattern to develop an application are:

- **Elimination of the if-then-else loops**: The if-then-else loop takes a lot of time to compile and complicates the code. It also causes bugs if multiple loops are invoked for the same state. The most important benefit of using the State design pattern is that it eliminates the need for the if-then-else loop in application code. By avoiding the if-then-else loop, the application is also free of code duplication.

- **Increased cohesion**: Behaviors that are specific to a particular state are grouped into the same classes. These classes have names that define the state related functionalities and this makes it easier to read the code and make any necessary changes.

- **Increased extensibility**: In the State design pattern, state functionalities are invoked implicitly with event triggering. As the application does not need to parse the code to retrieve a particular functionality, newer states and object behaviors related to these new states can be easily added later. This keeps the application easy to extend as the requirements of the application evolve with time.

- **Better testability**: It is easy to create mock objects to test an application based on the State design pattern. So an application developed using the State design pattern is easy to unit test.

The disadvantages of using the State design pattern are:

- **Class explosion**: The concept of programming to an interface is taken to the extreme, as each state needs a class defined for it. This means that because of a large number of classes in the application, it becomes voluminous, difficult to review, and slow to implement.
- **Brittle interface**: When a new event is introduced into the application, maintaining the state interface becomes an expensive task. This is because for every new task added, the events have to be defined in the Context class and the methods have to be defined in specific state classes.

Topic 5: Application Areas of the State Design Pattern

The State design pattern is useful in applications that focus on the state of the objects they handle. The scenarios in which the State design pattern would be a feasible choice for application development are:

- When an application needs to use complex decision making algorithms that are represented by state diagrams or flow charts
- When an application contains objects with varying yet distinguishable states
- When an application is used to program User Interfaces (UIs)
- When an application is used to design process testing for other products

Let us look at some of the real life situations where the State design pattern is implemented:

- **Remote control**: Remotes controls used for online and offline gaming are based on the State design pattern. The states managed here are playing, pausing game, buffering, connecting, and seeking goals.

- **Controller design**: Any kind of machine operation controllers, such as garage door openers, work on applications designed with the State design pattern. In the garage door opener controller application, the states handled are OPEN, CLOSED, and DOOR MOVING. Another example of the controller design is the vending machine application, discussed as a case study earlier.

- An implementation of a TCP connection can be based on State pattern.

Similarities between State and Strategy Pattern

The State and the Strategy pattern are similar in a few ways. Both the patterns enable you to add new state and strategy easily, without affecting the Context object, which uses them. You follow the open closed design principle when using both these patterns.

Differences between State and Strategy Pattern

Here are the differences between the State and Strategy design patterns:

- The State pattern encapsulates the state of an object, while the Strategy pattern encapsulates an algorithm or strategy.
- Strategy implementations can be passed as parameter to the Object which uses them. For example, the Collections.sort() method accepts a Comparator, which is a strategy. On the other hand, State is part of the Context object itself, and over time, the Context object transitions from one State to other.
- Order of State transition is well defined in the State pattern. However, there is no such requirement for the Strategy pattern. The client is free to choose any Strategy pattern implementation of his or her choice.

Lesson Summary

In this lesson, you were introduced to the State design pattern, which is used to define changes in an object's behavior when the state of the object undergoes a change. The State design pattern works by using state classes that invoke methods on object, depending on the current state of the object. The State design pattern is also used to handle object state transitions, and to generate the final result of an application. You also learned about the advantages of using the State design pattern and its areas of applications in the software development industry. The State design pattern helps in developing quick and efficient applications, and in simplifying complicated state related logic within an application.

References

- Web references:
  - http://www.javacodegeeks.com/2013/08/state-design-pattern-in-java-example-tutorial.html (Last accessed 25 May 2014)
  - http://java.dzone.com/articles/design-patterns-state (Last accessed 25 May 2014)

- Book references:
  - Czapsk, Michael; Krueger, Sebastian; Marry Brendan; Sahai, Saurabh; Vaneris, Peter; Walker, Andrew. (2008). Java CAPS Basics: Implementing EAI Patterns. Prentice Hall Publishing.
  - Langr, Jeff. (1999). Essential Java style: Patterns for Implementation. Prentice Hall PTR.

Lesson 7: Proxy Design Pattern

Lesson Overview

As the name suggests a proxy is a stand-in or a substitute for an application object. For example, a cheque or credit card is a proxy for what is in our bank account. It can be used in place of cash and provides a means of accessing that cash when required. Similarly, the Proxy pattern controls and protects the object from direct access.
In topic 1 of this lesson, you will be introduced to the Proxy design pattern the different types of proxies. You will also learn about the benefits and the drawbacks of Proxy design pattern.

In topic 2, you will learn to represent the implementation of a Proxy design pattern based application using a Unified Modeling Language (UML) diagram. You will also learn about the scenarios in which using the Proxy design pattern will help in creating a well defined and efficient application and the patterns similar to Proxy design pattern.

Lesson Objectives

At the end of this lesson, you will be able to:

- Define Proxy pattern
- Identify the situation to use Proxy pattern
- Implement solution using Proxy pattern
- Provide solution for the Case Study using UML diagram
- Give real-life application on Proxy pattern

Topic 1: Proxy Definition

Proxy is a substitute or an agent authorized to take decisions on behalf of another application or object while executing methods called on the original object. In application development, the Proxy design pattern introduces a class known as the proxy class that acts as an interface to any one of the following components of the application:

- Another class
- Database
- File
- Interface
- Network connection
- Library
- Application object

To understand the working of the Proxy design pattern, let us look at an example of an application object, which has a proxy defined to it. A client trying to access the object obtains a reference to the proxy instead. From this point onwards the client invokes methods on the proxy instead of original object. This gives the proxy the liberty and flexibility to perform a number of housekeeping operations, such as initialization, check client authorization to invoke a given method, and so on.
Figure 7.1 shows how the object and proxy implement the same interface.

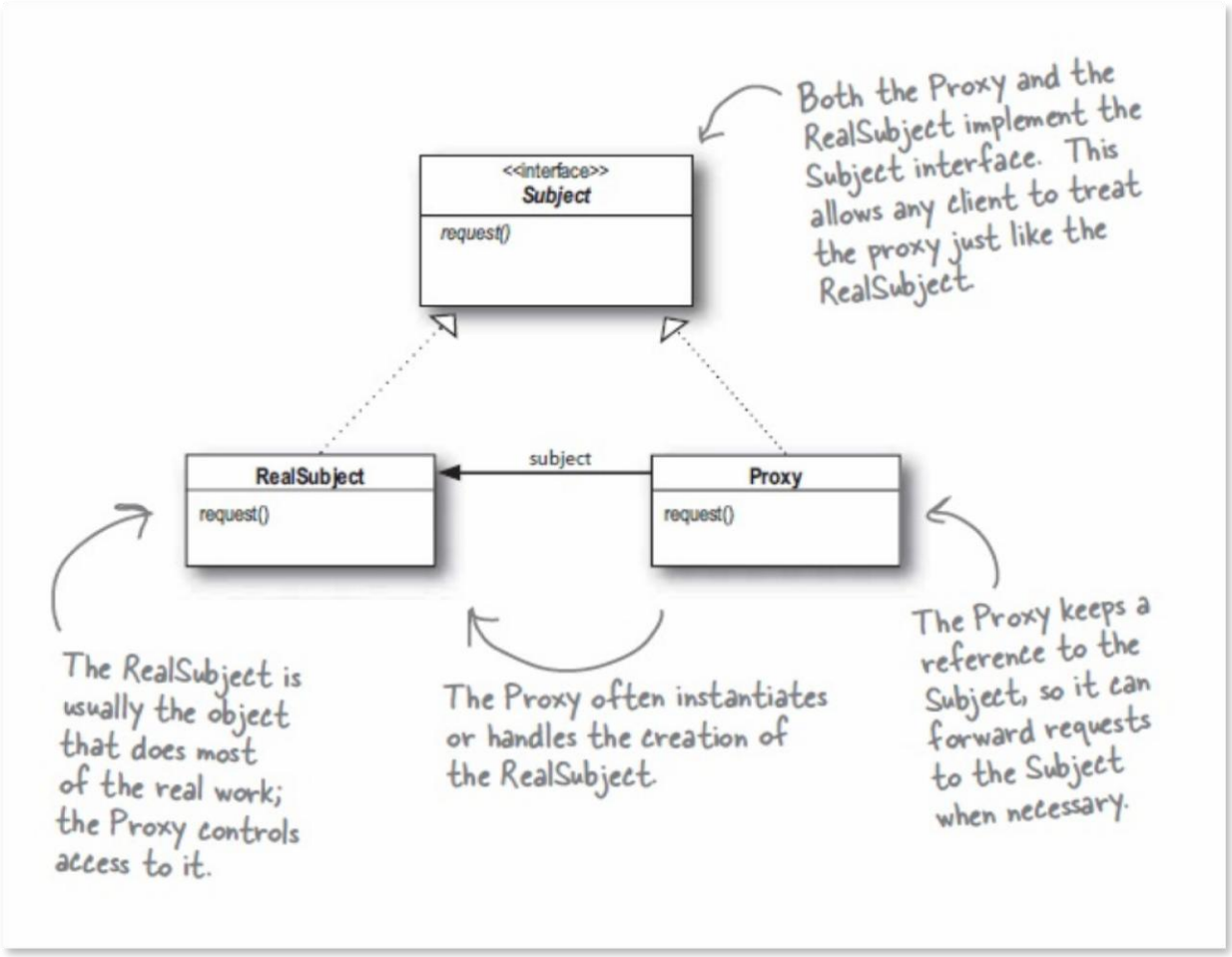


Figure 7.1: Object and Proxy Implement Same Interface

Figure 7.1 shows a Subject, which provides an interface for RealSubject and Proxy. The RealSubject and Proxy implement the same interface; therefore, it is possible to substitute Proxy for the RealSubject.

RealSubject is the object that the Proxy represents and controls access to.

Proxy holds a reference to RealSubject and the clients interact with RealSubject through Proxy. As Proxy and RealSubject implement the same interface, (Subject), Proxy can be substituted anywhere the subject can be used. As a result, Proxy controls access to RealSubject. This control comes in handy especially if the Subject interface is expensive to create in some way or if access to the subject is to be protected from the clients.

The intent behind using the Proxy design pattern is:

- To create a substitute or a placeholder for an object, to control access to the original object
- To add a wrapper to the original object, to protect it from unwanted complications
- To include an additional layer between users and objects, that supports distributed, controlled, or intelligent access to application objects

Different Types of Proxies

Proxies can be classified into the following types based on the type of control needed while defining access to an object:

- **Virtual proxies**: This type of proxy is used to delay the creation and initialization of an expensive object, such as database or network connections, until they are actually needed. The Virtual Proxy also acts as a surrogate for the object before and while it is being created. After that, the proxy delegates requests directly to the RealSubject. Figure 7.2 shows a Virtual Proxy.

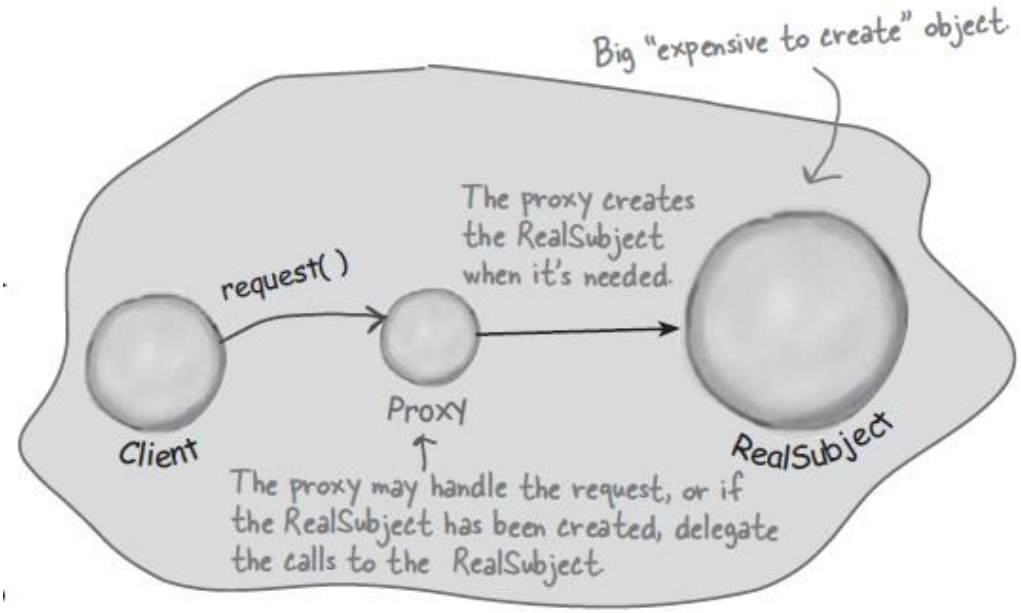


Figure 7.2: Virtual Proxy

- **Remote proxies**: This type of proxy is used to represent an object that is located at a different address space. For example, any Java Remote Method Invocation (RMI) stub is a proxy, which acts in place of the real object until the related methods are invoked in real time. Invoking methods on the proxy stub triggers the RMI stub to invoke the method on the real object that exists in a remote location.

- **Protection proxies**: This type of proxy is used to control access to the original object based on authorization of the client. The proxy acts as a security program and can decide whether to allow or deny access to a protected object.

- **Smart proxies**: A smart proxy introduces additional actions when the original an object is accessed. For example, a proxy may make the original object thread safe so that it can be accessed by only one thread at any given time.

Benefits and Drawbacks

The benefits of using the Proxy design pattern are as follows:

- **Providing security:** Using the Proxy design pattern can help to control access to the original objects, interfaces and classes. This is done using the proxy object to check the authorization of the client application to call on the original object.

- **Hiding addresses**: Using a proxy instead of the original object allows you to hide the location of the original object in the memory and prevent any modification to the original object by unauthorized methods and classes.

- **Performance optimization**: Using virtual proxies can optimize the execution of application. This is done by creating objects as and when needed in the application. The Proxy design pattern also handles common initialization, object management, and object release tasks using protection proxies and smart references. This reduces the amount of code needed to perform these housekeeping tasks making the application quicker and efficient.

- **Managing remote access -** The remote proxy allows treating a remote resource as a local object. This is a huge benefit as it reduces the amount of glue code to be written to access the remote resource and provides a single interface for interacting with it. Glue code lets existing parts of code to talk to each other that would otherwise be incompatible. If the API provided by the remote resource changes, the code only has to be changed in one place. It also stores all of the data associated with the resource including the URL of the resource, the data format used, and the structure of the commands and responses at one place.

Along with the benefits, there are also some drawbacks of using proxies in the application as follows:

- When an application developed using the Proxy design pattern, an extra layer of abstraction is introduced. This may negatively affect the code related to the original object when a client tries to access it because the application has one extra layer to traverse in order to access an object.
- If the RealSubject code is accessed by some of the clients directly this may cause confusion to future clients.

Topic 2: Case Study

The main purpose of the Proxy design pattern is to control access to an object. Access to an object is defined and controlled to instantiate expensive objects and to allow referencing of remote objects.

To understand the working of the Proxy design pattern, let us look at an example of an image viewer application. This application must be able to list and display images of high resolution contained in a folder on a system and must also be able to support an option to search for an image either, by using its name or thumbnail. Once the search results are shown as icons, clicking on one or more of the icons must render the images on the image viewer window in full resolution. The application is to be designed in such a way that the images should be loaded from disk and displayed one after another.

Implementing Image Viewer without Proxy Design Pattern

Let us first look at implementing this application without using the Proxy design pattern to understand the problems that arise while doing so and how the Proxy design pattern helps in solving these problems.

1. **Create an Image interface**: In the first step of writing the image viewer application, a simple interface called `Image` that handles image related tasks is created. This is shown in Code Segment 7.3.

```
public interface Image {

    public void showImage();
}
```

Code Segment 7.1: Create `Image` Interface

2. **Implement the `Image` interface**: Once the `Image` interface is created, it is implemented by creating a class called `HighResolutionImage`. This is shown in Code Segment 7.2.

```
public class HighResolutionImage implements Image {

    private String imagePath;

    public HighResolutionImage(String imagePath) {
        System.out.println("Object of HighResolution is Created");
        this.imagePath = imagePath;
    }

    @Override
    public void showImage() {
        System.out.println("I am in HighResolutionImage" + imagePath);
    }

}
```

Code Segment 7.2: Implementing `Image` Interface

In Code Segment 7.2, the method `HighResolutionImage` prints a message "Object of HighResolution is Created", when an object is created. The method `showImage()` prints "I am in HighResolutionImage" along with the address of the image in memory.

3. **Create viewer class**: Once the `Image` interface has been implemented, a class is created, which calls the `Image` interface to render an object on the screen of the system. This class is called `Viewer` and is shown in Code Segment 7.3.

```
public class ImageViewer {
    List<Image> images;

    public ImageViewer(List<Image> images) {
        this.images = images;
    }

    public void displayImage() {

        for (Image image : images) {
            image.showImage();
        }
    }

    public void setImage(List<Image> images) {
        this.images = images;
    }


}
```

Code Segment 7.3: Class `Viewer`

Code Segment 7.3 shows a list of images that can be displayed and a display method that iterates the list of image objects to display them.

4. **Write main method**: Now that all the interface and classes have been defined, e a main method is written, which invokes the `Image` interface and `Viewer` class. This is shown in Code Segment 7.4.

```
public static void main(String[] args) {

    HighResolutionImage highResolutionImage = new HighResolutionImage(
            "path1");
    HighResolutionImage highResolutionImage2 = new HighResolutionImage(
            "path2");
    List<Image> images = new ArrayList<>();
    images.add(highResolutionImage);
    images.add(highResolutionImage2);

    ImageViewer imageViewer = new ImageViewer(images);
    imageViewer.displayImage();

}
```

Code Segment 7.4: Writing the Main Method

In the main method seen in Code Segment 7.4:

- **Line 1 and 2**: Creates two instances of class `HighResolutionImage` which are the image icons selected by the user
- **Line 3, 4, and 5**: Adds the two instances to the list of images to be displayed
- **Line 6**: Creates an instance of `Viewer` class by passing the list of images to be displayed as an argument
- **Line 7**: Displays the image sequentially with the `displayImage()` method

Figure 7.3 shows the output for the image viewer application.

```
Object of HighResolution is Created
Object of HighResolution is Created
I am in HighResolutionImagepath1
I am in HighResolutionImagepath2
```

Figure 7.3: Output of Image Viewer Application

Although the application seen here works well, but when you look at the result, you will notice that there is something wrong. The Image objects are created before the image Viewer displays them. This may slow down the execution of the application, when there are large numbers of images to be selected by the user to view. If the application is implemented using the Proxy design pattern, the speed of execution of application and display of images will be much faster.

Implementing Image Viewer with Proxy Design Pattern

Let us now look at implementing this application using the Proxy design pattern. To implement the Proxy design pattern while developing an application, perform the following steps:

1. **Create an `Image` interface**: HighResolutionImage and ImageProxy both will implement this class.
2. **Implement the `Image` interface:**

3. **Create `Viewer` class**
4. **Create a proxy implementation of the Image object:** Once the `Image` interface and `Viewer` class have been created, the next step is to create a proxy for it. This is seen in Code Segment 7.5.

```
public class ImageProxy implements Image {
    /**
     * Private Proxy data
     */
    //Line 1
    private String imageFilePath;
    /**
     * Reference to RealSubject
     */
    //Line 2
    private Image proxifiedImage;

    //Line 3
    public ImageProxy(String imageFilePath) {
        this.imageFilePath= imageFilePath;
    }
    //Line 4
    @Override
    public void showImage() {
        //Line 5
        // create the Image Object only when the image is required to be shown
        proxifiedImage = new HighResolutionImage(imageFilePath);
        //Line 6
        // now call showImage on realSubject
        proxifiedImage.showImage();
    }
}
```

Code Segment 7.5: Create Proxy for HighResolutionImage

In Code Segment 7.5, the proxy object should implement the same interface as the original object. In Code Segment 7.5:

- **Line 1**: Specifies the `imagePath` to be displayed by the Image Viewer
- **Line 2:** Defines the local object that refers to the real object after creating the proxy
- **Line 3**: Shows the constructor that accepts the `ImageFilePath` and assigns it to a local variable
- **Line 4**: Shows the methods called by `imageViewer` to display image, which is a dummy method which in turn calls the actual method of the `real Object`
- Line 5: Shows the Proxy object that creates an instance of the actual object when `showImage` method is called
- Line 6: Shows the invocation of the actual `showImage()` method of `HighResolutionImage` class

5. **Write the main method**: Now, the main implementation that calls all of the interfaces and classes defined before is written. This is seen in Code Segment 7.6.

```
public static void main(String[] args) {
    //Line 1
    ImageProxy imageProxy = new ImageProxy("path1");
    ImageProxy imageProxy2 = new ImageProxy("path2");

    //Line 2
    List<Image> images = new ArrayList<>();
    images.add(imageProxy);
    images.add(imageProxy2);

    //Line 3
    ImageViewer imageViewer = new ImageViewer(images);
    imageViewer.displayImage();
}
```

Code Segment 7.6: Calling Main Method

In Code Segment 7.6:

- **Line1**: Shows the creation of proxy objects called `imageProxy` and `imageProxy2`.
- **Line2**: Shows the addition of proxies to the list of images to be passed on the `imageViewer`.
- **Line3**: Shows the invocation of the `imageView` instance and the `displayImage()` method to display all the images.

The output for the image viewer application using the Proxy design pattern is shown in Figure 7.4:

```
Object of HighResolution is Created
I am in HighResolutionImagepath1
Object of HighResolution is Created
I am in HighResolutionImagepath2
```

Figure 7.4: Output of Image Viewer Using the Proxy Design Pattern

Figure 7.4 shows when the Proxy design pattern is implemented, the objects are instantiated just before they are displayed. This helps the application to not consume too much memory and helps images load quicker (one after another). Now, the user does not have to wait until all the selected image objects are created, the images are displayed as soon as each corresponding object gets created.
Figure 7.5 shows how you used Proxy pattern to delay the object creation.

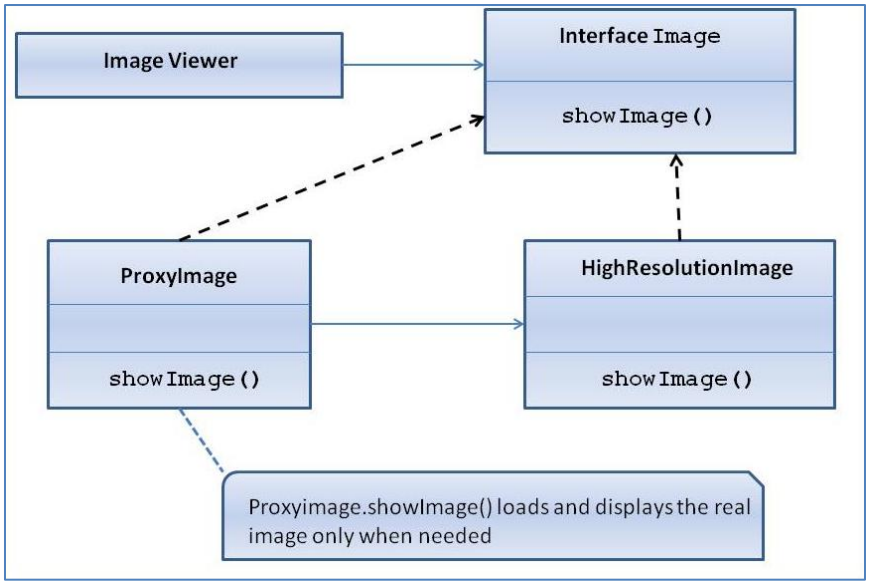


Figure 7.5: Image Viewer in UML Diagram Representation

Using Proxy Pattern

The Proxy design pattern is recommended when either of the following scenarios is true for the application:

- The original object being represented is external to the client system.
- Original objects needs to be created on demand. In this case, the proxy will create the original object only when a method is to be called on the original object.
- Access control for the original object is required.
- Added functionality is required when an object is accessed.

Typically, a proxy is used when communication with a third party is an expensive operation, perhaps over a network. The proxy holds the data until you are ready to commit, and can limit the amount of times that the communication is called.

The proxy is also useful to decouple actual implementation code to be directly accessible by clients and to access large files or graphics. By using a proxy, loading the resource can be delayed until the data in it is needed. Without the proxies, an application may become slow and appear non-responsive.

Patterns Similar to the Proxy Design Pattern

There are some patterns that are similar to the Proxy design pattern and may seem like a good substitute to the Proxy design pattern. However, it is necessary to understand that there are differences between the patterns and you have used the pattern most appropriate for a particular situation. Some of the design patterns that are similar to the Proxy design pattern are:

- **Adapter Design Pattern:** The Adapter design pattern allows the interface of one class to be used with another interface. It is different from the Proxy design pattern, which provides proxy objects using the same interface.
- **Decorator Design Pattern**: Though the structure of the Decorator design pattern is similar to the Proxy design pattern their purposes are different. The decorator design pattern adds new responsibilities to the objects, while the Proxy design pattern substitutes an existing behavior or object.

Common Areas of Application of Proxy Design Pattern

Some of the known uses of the Proxy design pattern are as follows:

- **Java Remote Method Invocation (RMI):** The Java RMI framework implements remote proxies to access data or object stored in remote locations. An RMI client can invoke methods on the remote object using a proxy which is also called a stub.

- **Database connection and transaction management**: Development tools, such as the Spring Framework use transaction proxies to start, commit, or rollback transactions automatically instead of depending on user written code. To protect the data stored in any database, protection proxies can be used to access specified objects within a database.

- **Dynamic mock objects in unit testing**: Dynamic proxies are used to implement stubs and mocks. Proxies can also be created while testing Data Access objects (DAOs).

- **Dynamic Proxies in Java:** Java supports the creation of dynamic proxy classes and instances. A dynamic proxy class gets created at runtime and implements a list of interfaces specified. Dynamic proxies in Java extend the java.lang.reflect.Proxy class.

Upon instantiation, each proxy instance gets associated to an invocation handler object, which implements the interface InvocationHandler. Any methods that are invoked on the proxy instance gets dispatched to the invoke() method of the instance's invocation handler. The invoke() method's signature is:

public Object invoke(Object proxy, Method method, Object[] args)

where:

- The proxy parameter is the reference to the proxy instance that the method was invoked on.
- The method parameter is the reference to the method instance corresponding to the method that was invoked on the proxy instance.
- The args parameter is an array of objects that contain the arguments that were passed in the method invocation on the proxy instance.

Let's take a look at a dynamic proxy example. Imagine that you want a List on which elements cannot be added. The first step is to create the invocation handler:

```
public class AddInvocationHandler implements InvocationHandler {

  private  List proxiedObj;

  public AddInvocationHandler(List _proxiedObj) {

    this.proxiedObj = _proxiedObj;
  }
```

```
  public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {

    if (method.getName().startsWith("add")) {

      return false;
    }

    return method.invoke(proxied, args);
  }
}
```

When the invoke method intercepts the method calls, it will do nothing if the method name starts with “add”. Else it will just pass the call to the real proxied object.

To create the proxy:

```
List proxy = (List) Proxy.newProxyInstance(
 AddInvocationHandlerTest.class.getClassLoader(),
 new Class[] { List.class },
 new AddInvocationHandler(list));
```

You will notice that the newProxyInstance method takes three arguments:

- The class loader
- An array of interfaces that the proxy will implement
- The invocation handler

Lesson Summary

In this lesson, you learnt about the Proxy design pattern in which a class known as the Proxy class represents the functionality of another class during application development. Proxies are powerful tools that can be implemented in many design patterns, such as Decorator and Adapter design patterns.

When the Proxy design pattern is used for application development, the application is easier to write the performance is optimized, errors are reduced, and the application is abstract. The Proxy design pattern is an efficient way to control access within an application and also helps developers to avoid code duplication. All of this contributes in making the application flexible, portable, efficient, and quick. You typically use a proxy when communication with a third party is an expensive operation.

References

- Web references:
  - http://sourcemaking.com/design_patterns/proxy (Last accessed on 20 May, 2014)
  - http://www.oodesign.com/proxy-pattern.html (Last accessed on 20 May, 2014)

- Book references:
  - Freeman, Eric; Freeman, Elizabeth. (2010). *Head First Design Patterns.* O'Reilly Media.
  - Metsker, Steven John; Wake, William. C. (2006). *Design Pattern in Java.* Addison-Wesley-Longman Publishing.

Lesson 8: Other Design Patterns

Lesson Overview

Enterprise applications can be developed using a variety of design patterns. These patterns are used to describe the structure, implementation, and mechanisms used to reach business goals. This lesson aims at introducing you to some of the design patterns used to develop applications in Object-Oriented Programming (OOP).

The patterns discussed in this lesson are the Bridge design pattern, Chain-of-responsibility design pattern, Strategy design pattern, and Moderator design pattern. The first design topic detailed in this lesson is the Bridge design pattern which separates the implementation and abstraction mechanisms in an application in order to keep the application loosely coupled.

The second design pattern discussed in this lesson is the Chain-of-responsibility pattern where a chain of objects are used to process user requests based on the object's capabilities. This design pattern is useful in developing request or response based applications.

The third design pattern introduced in this lesson is the Strategy design pattern. As its name suggests, the Strategy design pattern helps the application to decide which strategy needs to be implemented to perform a given task. This decision is taken only at run time, making the application platform independent.

The last design pattern this lesson defines in the Moderator design pattern. This design pattern is used to handle communication between application components by using a single Moderator class. By streamlining the communication component using the Moderator design pattern, the application's code can be designed to implement its components efficiently and with minimum intervention from the developer.

Lesson Objectives

At the end of this lesson, you will be able to:

- Define the Bridge pattern
- Define the Chain of Responsibility pattern
- Define the Strategy pattern
- Define the Mediator pattern
- Identify the situations to use these patterns

Topic 1: Chain of Responsibility Design Pattern

Introduction

Imagine a help system that contains information on how to use a graphical user interface. By clicking on any part of the interface, the end user can obtain help information about it. The help information also depends on the context. For example, a button in a dialog box might have different help information than a similar button in the main window. If there's no specific help information for that part of the interface, then the help system should be able to display a more general help message about the immediate context, for example, the dialog box as a whole.

Hence, it's natural to structure the help information according to its generality and relevance, from the most specific to the most general. To do this, you need a way to decouple the button that initiates the help request from the objects that might provide help information. The Chain of Responsibility pattern defines how that happens. The pattern decouples senders and receivers by giving multiple objects a chance to handle a request. The request gets passed along a chain of objects until one of the handlers handles it.

This topic deals with the Chain-of-responsibility design pattern in detail, starting with its purpose.

Purpose, Working and Characteristics of Chain-to-responsibility Design Pattern

The Chain of Responsibility pattern (CoR) helps in keeping a low degree of coupling between an object that sends out a request and the set of potential objects that handle requests. When there is more than one object that can handle or fulfil a client request, the CoR pattern recommends giving each of these objects a chance to process the request in some sequential order. Applying the Chain of Responsibility pattern in such a case, each of these potential handler objects can be arranged to form a chain, with each object having a reference 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, and 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 fulfil 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 let the request remain unprocessed. In other words, a request submitted to the chain of handlers may not be fulfilled even after reaching the end of the chain.

The purpose of using the CoR design pattern to develop OOP-based applications is to:

- Avoid attaching request source information to the object that receives the request
- Give multiple objects the possibility of handling a request
- Send a request across a chain of objects until it reaches one that can handle its processing

The CoR pattern is applicable if:

- You want to decouple the sender and receiver of a request.
- Multiple objects (handlers), determined at runtime, can be the candidates to handle a request.
- No need to specify handlers explicitly in the code.

Following are some of the important characteristics of the CoR pattern:

1. 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.
2. 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.
3. 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 fulfil the request.
4. Request handling is not guaranteed. This means that the request may reach the end of the chain without being fulfilled.

Figure 8.1 shows a CoR pattern class diagram.

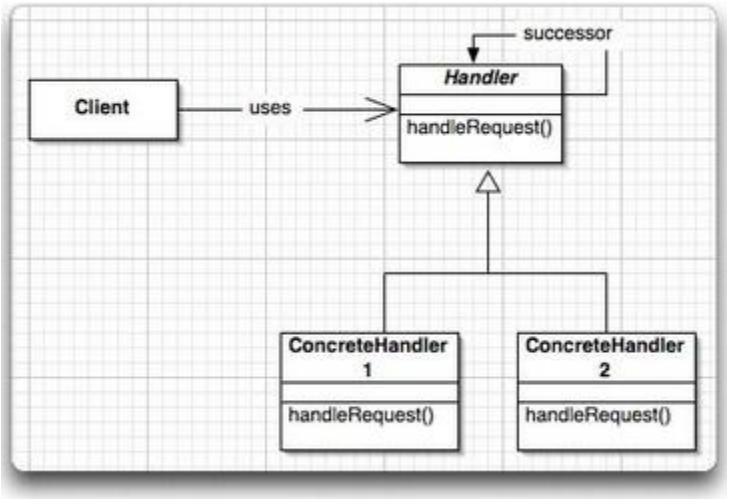


Figure 8.1 – Chain of Responsibility Pattern Class Diagram

Request handlers typically extend from a base class. Each handler maintains a reference to the next handler in the chain, known as the successor. The base class might implement handleRequest() like this:

```
public abstract class BaseHandler {
       ...
       public void handleRequest(RequestObject ro) {
          if(successor != null){
                successor.handleRequest(ro);
          }
       }
}
```

So by default, handlers pass the request to the next handler in the chain.

You must not use CoR when each request is only handled by one handler, or, when the client object knows which service object should handle the request.

Now, you are familiar with the components and structure of the Chain-of-responsibility design pattern. In the next section, let us look at some of the features of the Chain-of-responsibility design pattern.

Code Implementation

An apt example of using the CoR pattern is the ATM. The user enters the amount to be dispensed and the machine churns out the amount in terms of the defined currency bills such as 50$, 20$, 10$ etc. it throws an error if the user enters an amount that is not multiples of 10. You will use the CoR pattern to implement this solution. The chain will process the request in the same order as shown in Figure 8.2.

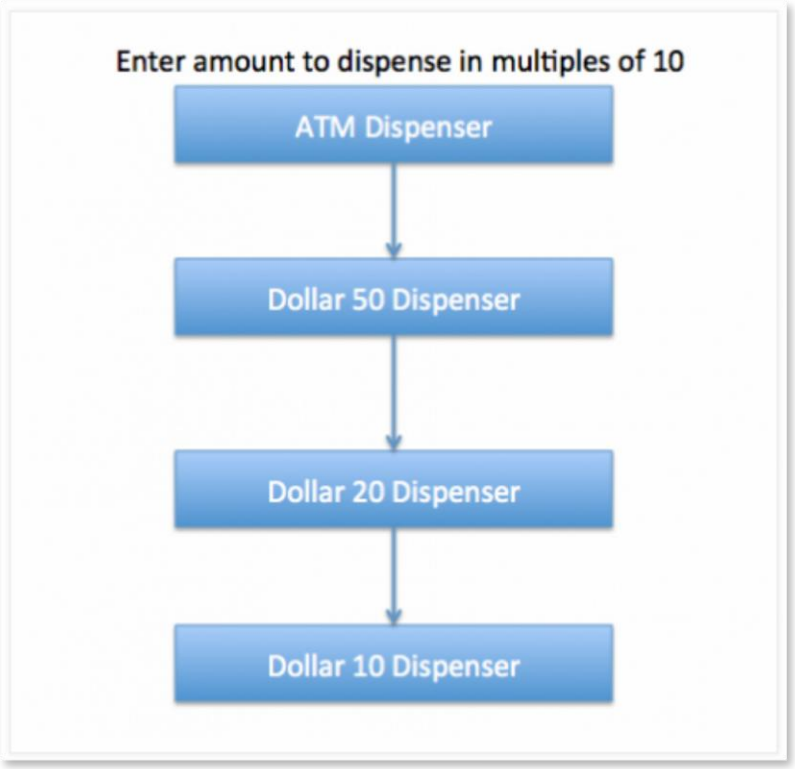


Figure 8.2: ATM Dispenser Model

To implement the pattern, first define the base classes and interfaces and then create the concrete handlers.

Base Classes and Interface

Currency is a simple class to store the amount to dispense. It's object will be used by the chain implementation objects (handlers).

Currency.java

```
public class Currency {

   private int amount;

   public Currency(int amt){
      this.amount=amt;
   }

   public int getAmount(){
      return this.amount;
   }
}
```

The base interface will have a method to define the successor in the chain and a method to process the request.

DispenseChain.java

```
public interface DispenseChain {

   void setNextHandler(DispenseChain nextHandler);

   void dispense(Currency cur);
}
```

Concrete Chain Handlers

You need to create 3 concrete handlers, one each for different types of bills - $50, $20, and $10.

50DollarHandler.java

```
public class 50DollarHandler implements DispenseChain {

   private DispenseChain nextHandler;

   @Override
   public void setNextHandler(DispenseChain nextHandler) {
      this.nextHandler=nextHandler;
   }

   @Override
   public void dispense(Currency cur) {
      if(cur.getAmount() >= 50){
         int num = cur.getAmount()/50;
         int remainder = cur.getAmount() % 50;
         System.out.println("Dispensing "+num+" 50$ note");
         if(remainder !=0) this.nextHandler.dispense(new Currency(remainder));
      }else{
         this. nextHandler.dispense(cur);
      }
   }

}
```

```
public class 20DollarHandler  implements DispenseChain{


  private DispenseChain nextHandler;

   @Override
   public void setNextChain(DispenseChain nextHandler) {
      this.nextHandler = nextHandler ;
   }

   @Override
   public void dispense(Currency cur) {
      if(cur.getAmount() >= 20){
         int num = cur.getAmount()/20;
         int remainder = cur.getAmount() % 20;
         System.out.println("Dispensing "+num+" 20$ note");
         if(remainder !=0) this. nextHandler.dispense(new Currency(remainder));
      }else{
         this.nextHandler.dispense(cur);
      }
   }

}

public class 10DollarHandler implements DispenseChain {


   private DispenseChain nextHandler;


   @Override
   public void setNextChain(DispenseChain nextHandler) {
      this. nextHandler = nextHandler;
   }

   @Override
   public void dispense(Currency cur) {
      if(cur.getAmount() >= 10){
         int num = cur.getAmount()/10;
         int remainder = cur.getAmount() % 10;
         System.out.println("Dispensing "+num+" 10$ note");
         if(remainder !=0) this. nextHandler.dispense(new Currency(remainder));
      }else{
         this. nextHandler.dispense(cur);
      }
   }
```

```
}
```

Creating the Handler Chain

Let's test our chain implementation to see if all handlers do their job perfectly

ATMDispenseTest.java

```
import java.util.Scanner;

public class ATMDispenseTest {

   private DispenseChain c1;

   public ATMDispenseChain() {
      // initialize the chain
      this.c1 = new 50DollarHandler();
      DispenseChain c2 = new 20DollarHandler();
      DispenseChain c3 = new 10DollarHandler();

      // set the chain of responsibility
      c1.setNextHandler(c2);
      c2.setNextHandler(c3);
   }

   public static void main(String[] args) {
      ATMDispenseTest atmDispenser = new ATMDispenseTest();
      while (true) {
         int amount = 0;
         System.out.println("Enter amount to dispense");
         Scanner input = new Scanner(System.in);
         amount = input.nextInt();
         if (amount % 10 != 0) {
            System.out.println("Amount should be in multiple of 10s.");
            return;
         }
         // process the request
         atmDispenser.c1.dispense(new Currency(amount));
      }

   }
```

```
}
```

Lets check the output -

```
Enter amount to dispense
530
Dispensing 10 50$ note
Dispensing 1 20$ note
Dispensing 1 10$ note
Enter amount to dispense
100
Dispensing 2 50$ note
Enter amount to dispense
15
Amount should be in multiple of 10s.
```

Areas of Application of Chain of Responsibility Pattern

- In web applications built in Java Servlet filters are an application of CoR pattern as the filter framework allows multiple filters to process an HTTP request.
- Struts2 uses interceptors which are again great example of CoR pattern implementation.

In this topic, you were introduced to the concepts, purpose, implementation, and uses behind the Chain-of-responsibility design pattern. The next topic aims at introducing you to the Strategy design pattern.

Topic 2: Strategy Design Pattern

The Strategy design pattern is a behavioral design pattern that supports selecting an algorithm's or code block's behavior at run time. This design pattern is used to keep the algorithm independent of the client that uses it. The Strategy design pattern is used to:

- Define a group or family of algorithms
- Encapsulate each algorithm
- Keep algorithms interchangeable within a family

The Strategy design pattern is used when the run time conditions for an application cannot be predetermined. In this pattern, the methods and functionalities are defined in a generic fashion and the specific behaviors of the application are specified only at run time. In this topic, you will be introduced to the Strategy design pattern and its implementation. Let us start by learning about the purposes and working of the Strategy design pattern.

Purpose of the Strategy Design Pattern

Computer programs depend on algorithms. There are fundamental algorithms for routine activities like sorting and searching as well as application-specific algorithms for things like rating an insurance policy or calculating customer discounts.
Many times, while programming, you have a choice as to which algorithm to use to solve a problem. For example, there are more than 10 different mainstream algorithms for sorting a list of elements.
The Strategy pattern helps select an algorithm based on runtime decisions. Runtime selection is preferable when the choice of algorithm depends on factors not available at design time. Such factors include:

- Nature of input
- Source of input
- User preferences
- Current conditions

This pattern works by using objects to reference the concrete algorithms that are to be used in specific conditions.

Components that postpone algorithm selection until runtime are good candidates for the Strategy design pattern. The Strategy design pattern shows how to design a class with interchangeable algorithms or behaviors.

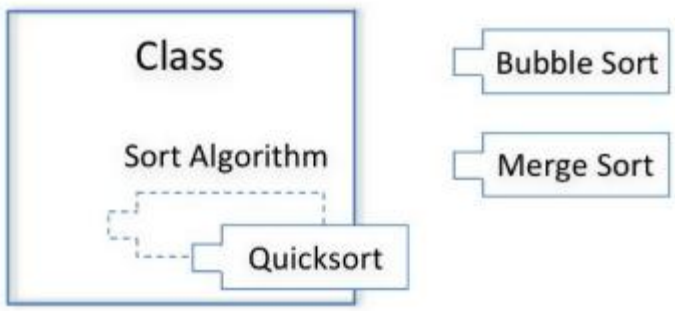


Fig 8.3: Designing Classes with Interchangeable Algorithms

Intent of the Strategy Pattern

The Strategy design pattern helps to define a collection or a family of algorithms. Each algorithm is encapsulated and can be used interchangeable. Using the Strategy pattern, you can change an algorithm without impacting the client. Often, you will find the Strategy design being used where you have a class with mutually exclusive algorithms or behaviors that are selectable at runtime. For example, the following code fragment, shown in Code segment 8.1 uses a compound if statement to select one of three algorithms:

```
class Context {
    void operation() {
        if (. . .) {
            // algorithm 1
            . . .
        }
        else if (. . .) {
            // algorithm 2
            . . .
        }
        else {
            // algorithm 3
            . . .
        }
    }
}
```

Code Segment 8.1: Selecting Algorithms

Strategy encapsulates each algorithm in a separate class and **replaces the conditional logic** in the context with delegation to an encapsulated algorithm, as shown in Figure 8.4

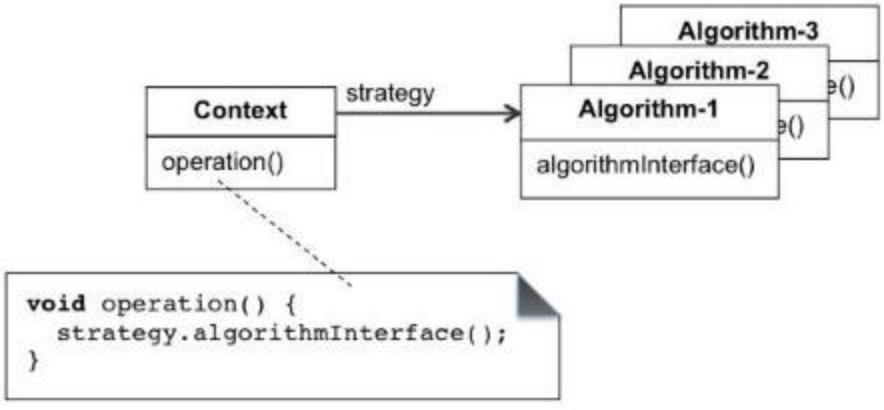


Figure 8.4: Strategy Pattern Encapsulates each Algorithm in a Separate Class

Separating the algorithms from the context allows the two to vary independently. New algorithms can be added or existing ones modified without modifying the context. Conversely, the context can be modified without affecting the algorithms.

The structure diagram of the Strategy pattern is shown in Figure 8.5.

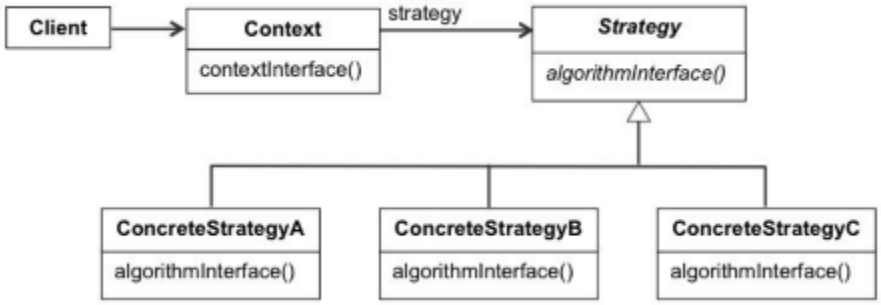


Figure 8.5: Strategy Pattern Class Diagram

In the Strategy Pattern class diagram, there is a concrete strategy class for each interchangeable algorithm or behavior. Note that in the context of the Strategy pattern, the term "strategy" is a synonym for algorithm or behavior. The Strategy pattern defines an interface common to all concrete strategy classes. All concrete strategy classes must implement the same interface.
Context is a class with a configurable algorithm or behavior. It keeps a reference to a Strategy object, which is the abstract interface for a concrete strategy. Clients usually choose which algorithm is used by the context.

Here, you have seen the purpose and intent of the Strategy design pattern. Let's now look at the implementation of the Strategy design pattern while developing applications.

Implementation

The Strategy design pattern is one of the simpler design patterns and is very easy to implement in an application. To implement the Strategy design pattern, the following steps are performed:

1. Create and implement a `Strategy` interface for strategy objects.
2. Create and implement a `ConcreteStrategy` interface for every strategy defined in the `Strategy` interface.
3. In the `Context` class, maintain private references to all `Strategy` objects.
4. In the `Context` call, implement the setter and getter methods for the strategy object selected at run time.

The `Strategy` interface is used to define the behavior of the `Strategy` objects. The `ConcreteStrategy` class implements the `Strategy` interface. Let us now look at the benefits and drawbacks of the Strategy design pattern.

Benefits and Drawbacks

The benefits of using the Strategy design pattern to write application code are:

- **Creates families of related algorithms**: The Strategy class hierarchies are used to define families of related algorithms or behaviors. The inheritance defined in the hierarchy can eliminate common functionalities of algorithms, thus helping in reducing the volume of code needed to be created.

- **Used as an alternate to sub classing**: The inheritance defined in the Strategy class can support a variety of algorithms. Developers need not create behavior sub classes from the Context class and thus avoid hardwiring behavior to a specific code, which keeps the application loosely coupled and flexible.

- **Encapsulates algorithms**: Encapsulating the algorithm in separate Strategy classes lets you to alter the algorithm individually of its context. This makes it easier to switch, understand, and extend.

- **Eliminate conditional statements**: The Strategy design pattern is an alternative to using conditional statements to select desired behaviors at run time.

You learned about the advantages of using the Strategy design pattern. However, there are certain drawbacks while using the Strategy design pattern, such as:

- The client must be aware of the nature of all the available strategies in the application to be able to make a well-informed decision in selecting the appropriate strategy at run time. Else, the application may become slow, have errors, or generate incorrect results.
- There is a communication overhead caused by the interactions between the Strategy and the Context interfaces.
- Providing greater number of strategy choices may seem like a good idea to a developer. However, they have to be aware that this increases the number of objects that has to be handled by the application.

Areas of Application

The areas of software development where the Strategy design pattern can be used are:

- In Java, a cipher input stream can be used to decrypt encrypted text.

How it is done in Java makes the cipher stream totally oblivious of the encryption algorithm that you use. String path = ... ;

Cipher strategy = ... ;

InputStream = new CipherInputStream(new FileInputStream(path), **strategy**);

The Cipher stream objects decrypts using the algorithm as specified by the strategy(algorithm) object.

- In the model-view-controller (MVC) model, the *view* and the *controller* implement the classic Strategy pattern. A view can pick the appropriate controller from the controller hierarchy to let user input be handled as required. For example, you may need the view to respond differently to the keyboard or a mouse action by selecting the appropriate controller. It is the controller that encapsulates the complex interactions of updating the model, hence the same view can be used with different controllers to have different behavior in different parts of your system and the controller can be replaced at run-time as needed. In this sense, the controllers become the strategy objects that the views can interchangeably make use of.

- In the Java Development Toolkit (JDK), the Strategy design pattern can be used to perform sorting operations in:
  - `Java.util.Collections#sort` interface
  - `java.util.Arrays#sort` interface

In this topic, you have been familiarized with the Strategy design pattern and its uses in application development. The next topic introduces you to the Mediator design pattern.

Topic 3: Mediator Design Pattern

Mediator design pattern is used to encapsulate interactions between a set of related objects. Air traffic controller (ATC) is a great example of mediator pattern, where the ATC acts as a mediator for communication between various flights. Mediator pattern is a behavioral pattern as it can alter the program's running behavior.

Purpose and Working of Mediator Design Pattern

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 8.6 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 8. 6 and Figure 8.7), 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.

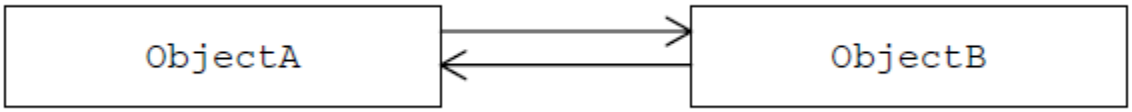


Fig 8.6: Point –to – Point Communication in Case of Two Objects

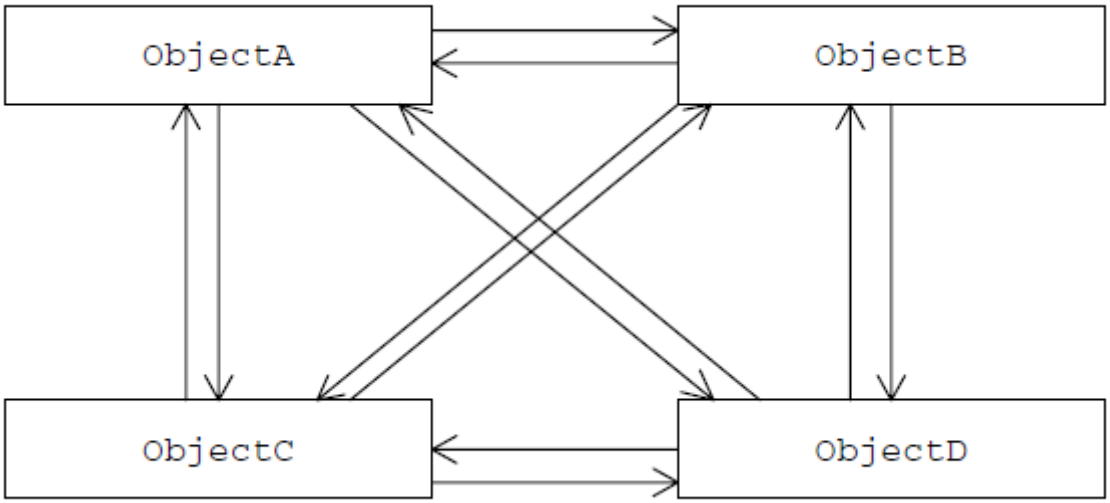


Figure 8.7: Point –to – Point Communication- Increased number of objects

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, as shown in Figure 8.8.

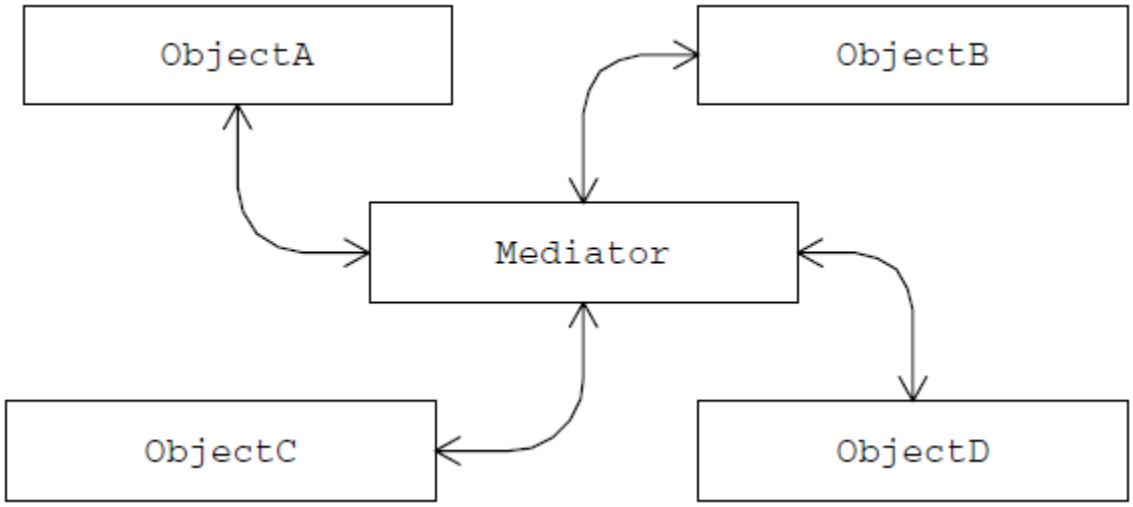


Figure 8.8: Object Interaction – with Mediator as a hub

The Mediator design pattern encapsulates communication between objects with a mediator object. As you see in Figure 8.8, by implementing the Mediator pattern, objects do not communicate directly with each other, but instead communicate through the Mediator object. This reduces the dependencies between communicating objects, thereby reducing object to object coupling.

The resulting design has the following key 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.

Example

One of the examples of the Mediator pattern is represented by the Dialog classes in GUI application frameworks. A Dialog window can be a collection of graphic and non-graphic controls. The Dialog class can provide the means to become a mediator and facilitate the interaction between controls. For example, a Label has to display the value that gets selected from a ComboBox object. Both the ComboBox and the Label are not aware of each other's structure and all the interaction is managed by the Dialog object. Each control is not aware of the existence of other controls.

In this way, the Mediator design pattern is used to partition a system into many objects that promotes code reusability. The Mediator object does the following:

- Encapsulates object interconnections
- Acts as a communication hub
- Is responsible for controlling and coordinating client interactions
- Promotes loose coupling by keeping objects from referring to each other

Figure 8.9 shows the UML class diagram that represents the structure of the Mediator design pattern used during application development.

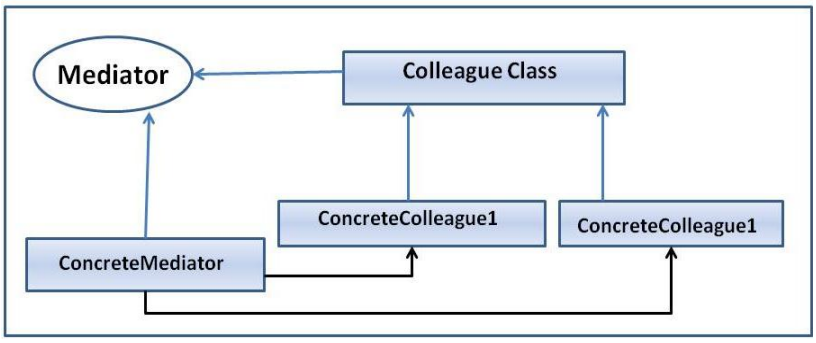


Figure 8.9: UML Class Diagram for Moderator Design Pattern

Figure 8.9 shows the components of the Mediator design pattern. The components of the Mediator design pattern which participate in managing and controlling object interactions are:

- **`Mediator`**: This is an interface defined for communication between `Colleague` classes.

- **`ConcreteMediator:`** This component keeps track of all the `Colleague` classes and maintains references to the `Colleague` objects. This interface component implements the operations that handle communication and transfers messages between the `Colleague` classes.

- **`Colleague classes:`** This component is used to keep a reference to its `Mediator` object. The `Colleague` classes communicate with the `Mediator` whenever it would have otherwise communicated with another `Colleague`.

Now that you have been introduced to the Mediator design pattern, let us learn about its implementation.

A disadvantage with the Mediator design pattern is the complexity of the `Mediator` class as more number of Colleague objects are added. A good practice is to make the `Mediator` classes responsible only for the communication. Let us learn about the areas in the software industry where the use of Mediator design pattern is feasible.

Mediator Versus Façade

In some aspects, the Mediator pattern looks similar to the Façade pattern, as discussed earlier. Table 8.1 lists the similarities and differences between the two.

Mediator	Façade
All objects interact with each other through the Mediator. The group of objects knows the existence of the Mediator.	Clients use the Façade to interact with subsystem components. The existence of the Façade is not known to the subsystem components.
Because the Mediator and all the objects that are registered with it can communicate with each other, the communication is bidirectional.	Clients can send messages (through the Façade) to the subsystem but not vice versa, making the communication unidirectional.

Areas of Applications

In the software industry, the Mediator design pattern can be used to develop:

- **GUI libraries**: To develop Dialog classes in GUI application frameworks to provide a mechanism for interactions between different control methods.

- **Chat applications**: Several participants in a chat application communicate with a Mediator in order to participate in the ongoing chat. Mediator usage found in Java API:

java.util.Timer (all scheduleXXX() methods)
java.util.concurrent.ExecutorService - submit() method
java.lang.reflect.Method#invoke()

In this topic, you were briefly introduced to the Mediator design pattern and its uses in managing communication between internal and external components in an application.

Lesson Summary

In this lesson, you were familiarized with the following three design patterns:

- Chain-of-responsibility design pattern
- Strategy design pattern
- Mediator design pattern

You learned about the features, advantages, and areas of application for each design pattern.

The CoR design pattern is used to create a sequence of request handler objects which either have the capability to process a request or pass it on to the next object in the chain. This design pattern is used to send commands to multiple objects in chain and the object most suited to satisfy the command is invoked to generate a result.

The Strategy design pattern is used to enable an application's algorithm to behave in a specific way that is decided only at run time. This design pattern is useful in creating applications where a family of algorithms are grouped together and can be used interchangeably as per the user's request parameters.

Finally, the Mediator design pattern is used to control and manage inter-object communications in an application. In this design pattern, the Mediator object acts as a communication hub and facilitates communication between multiple classes.

References

- Web references:
    - http://sourcemaking.com/design_patterns/bridge (Accessed 28 May 2014)
    - http://sourcemaking.com/design_patterns/strategy (Accessed 28 May 2014)
    - http://www.oodesign.com/chain-of-responsibility-pattern.html ((Accessed 28 May 2014)

- Book references:
    - Metsker, Steven John; Wake, William C. (2006). *Design Patterns in Java.* Addison-Wesley Publishing
    - Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John. (2000). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Publishing.

Module 2: Other Industry Patterns

Module Overview

In the previous module, you learnt about design patterns, such as Abstract Factory, Adapter, Façade, and Command. In addition to these, there are patterns that can be used when large problems need to be solved by breaking them into smaller number of steps. These patterns include Service Layer, Pipes and Filters, Blackboard Architectural pattern, and Antipatterns, which helps the developer in addressing large problems.

There are two lessons in this module. In lesson 1, you will learn about Service Layers which are repositories for the services defined in the Services-Oriented Architecture (SOA).
In lesson 2, you will learn about AntiPatterns, which help to recognize common problems in the software industry. In this lesson, you will also learn about the solutions for these problems.

Module Objectives:

By the end of this module, you will be able to:

- Describe other industry patterns
- List the uses of other industry patterns

Lesson 1: Service-Oriented Architecture

Service-Oriented Architecture (SOA) is a design pattern used to provide an application's functionalities as services to other applications. This kind of service orientation is independent of any vendor, product, or technology. Such services are used by applications with multiple responsibilities. For example, applications that are used to handle online banking transactions and other scenarios that needs multiple systems interacting over a vast network. In this lesson, you will learn to describe the SOA design pattern and also identify the various components and design considerations needed to implement the SOA design pattern.

Topic 1: Introduction to SOA

SOA is a software architecture design pattern based on discrete pieces of software called services that provide functionality to other applications. SOA is not dependent on a particular vendor, product, or technology. In the SOA pattern, often services are combined together to provide the complete functionality of a large software application.
SOA makes it easy to design an application using services that run on various computers connected over a network. These services work collectively to make the application. Each service is built in a way that ensures that the service can exchange information with any other service in the network. Each service implements at least one action, such as to return a credit rating for an individual, to reserve an airplane seat, to retrieve an online bank statement, or to modify an online booking.

There are certain protocols that services in SOA use. These protocols describe what message format and data types the service can understand and exchange messages in. If one is using Web Services to implement SOA, then Web Services Description Language (WSDL) is typically used describe the services, while Simple Object Access Protocol (SOAP) is used to describe the communications protocols.

Purpose of SOA

The purpose of SOA is to allow users to combine together fairly large chunks of functionality to form applications built mostly from existing software services.

In the software industry, numerous resources are used heterogeneously across systems, applications, software, and application infrastructure. Enterprise applications should be able to:

- Adapt to changes in business models.
- Support new channels of interactions with customers and partners.
- Have an architecture that supports organic growth within the organization.
- Support the development of applications by combining large chunks of functionalities from existing software services.
- Support existing applications so that they are compatible with new cross-functional business processes.

The SOA design pattern and its loosely coupled nature allows enterprise applications to plug in new features or upgrades to address new business requirements and makes communication across various

channels safe and easy. The SOA design pattern can also be used to expose existing enterprise infrastructures as services to other applications, and thereby, protect the existing investments in these infrastructures.

Topic 3: Services in SOA

The concept of a service is at the heart of the SOA architectural pattern. Each service is built as a discrete piece of code.

SOA enables loose coupling. Loose coupling is an important and fundamental principle in the SOA architecture.

Most web applications are tightly coupled to a large extent, in the sense that each of the subsystems that comprise the application is dependent on other subsystems at compile time and run time. This traditional design does not give us the ability to easily replace or make changes to subsystems if business requirements change.

On the other hand, using SOA, the application's functionality is exposed through a collection of services. These services are not tightly coupled; they are independent and encapsulate both the business logic and its associated data. The services are interconnected via messages with a schema defining their format, as shown in Figure 2.1.

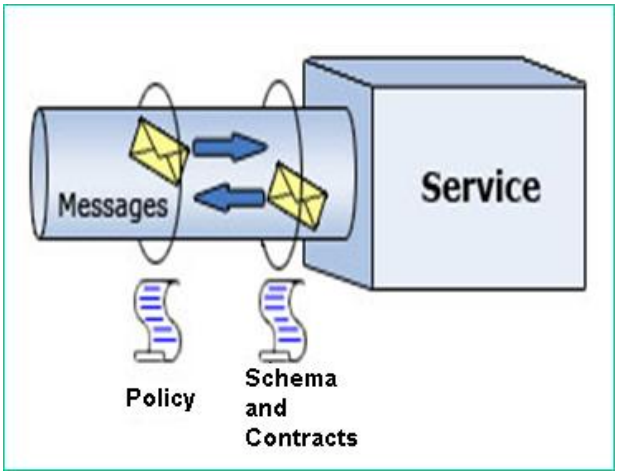


Figure 2.1: Service Defined by Schema and Policy

The services of an application are designed with the expectation that you cannot control where and who consumes them. A service can be used by any consumer outside of the control of the application. These consumers can be other services (incorporating the encapsulated business process) or user interfaces provided by the application itself. In addition to offering services and exploiting its own services, an application can also adapt to new services offered after deployment. The availability and stability of these services, therefore, becomes a critical factor.

Web Services Approach

Web services is one of the ways you can implement a service-oriented architecture, especially since they make functional pieces of code accessible over standard Internet protocols independent of platforms and programming languages.

The most important web services standards include:

- SOAP , which defines how messages are exchanged in a network
- WSDL, which describes public interfaces to available web services, including the message formats that must be used to interact with the services
- Universal Description, Discovery and Integration (UDDI), which is a standard for describing available services

Organizational Benefits of SOA Design Pattern

Benefits of SOA include:

- SOA helps in separation of large projects into smaller projects (services). Services can be delivered quickly and independently from the larger and slower-moving projects common in the organization. This gives the business more time to understand systems and simplify user interfaces calling on services. This also promotes agility and speeds up time-to-market.
- It is easier to document and test an individual service compared to a large monolithic application. This is important when the service needs to be reused later.

Some Real Life Examples of SOA

Some real-life applications of SOA are:

- First Citizens Bank, which not only provides services to its own customers, but also to about 20 other institutions. The services include check imaging, check processing, outsourced customer service, and "bank in a box" for getting community-sized bank everything they need to be up and running. Underneath these services, there is an SOA-enabled mainframe operation.
- Thomson Reuters, a provider of business intelligence information for businesses and professionals, which maintains a stable of 4,000 services that it makes available to outside customers. Example of one such service is Thomson ONE Analytics, which delivers a broad range of financial content to Thomson Reuter's clientele.

Topic 4: Implementation of SOA

Let's consider the scenario of a hypothetical reservation system, as shown in Figure 2.2. The solution consists of three sub-systems: a user management service which is responsible for managing user accounts, authentication, and authorization; a content management service which maintains necessary information, such as details, images, and so on regarding hotels and their offerings; and a core reservations sub-system, which is used for creating room reservations, and maintaining availability, and so on. The Booking Engine, which is the public portal, connects with all three sub-systems via the Service APIs.

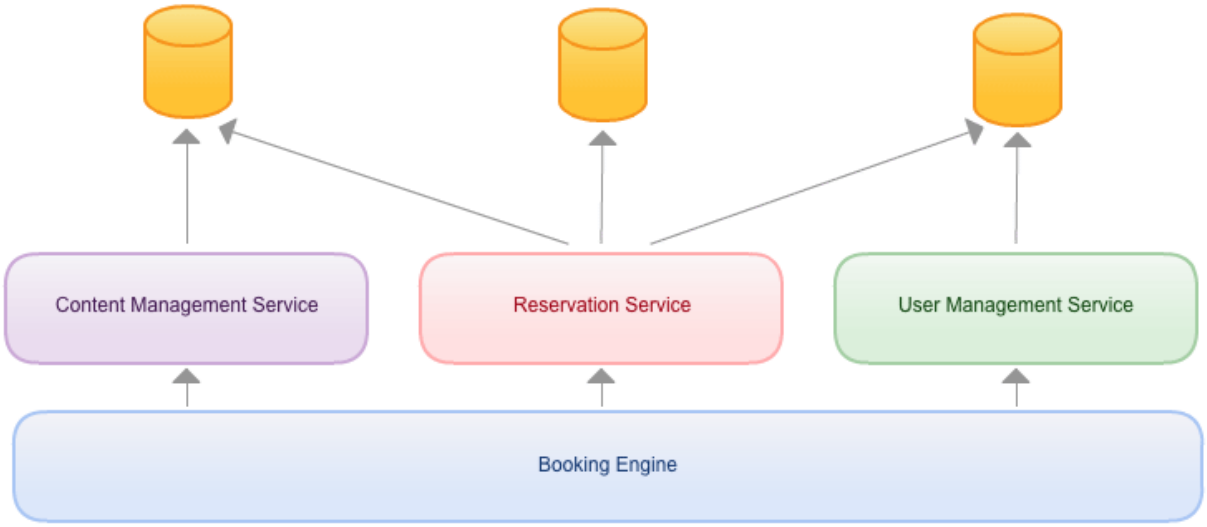


Figure 2.2: A Hypothetical Reservation System

The reservation system needs information from the content management service to get details on hotels, the room types being reserved, and so on. In addition, the reservation system requires information about the user creating the reservation to enforce access control and authorization levels.

It would be easier to get this information by directly reading it from the data store of the other system, as shown in Figure 2.2. However, this approach will not really be an SOA implementation. Remember that the services should talk to each other via well-defined interfaces but there should not be any direct-access of another data store of the service. The only communication that should happen between services is via service interface calls over the network.

If you are to implement it with SOA, it could be done as follows.

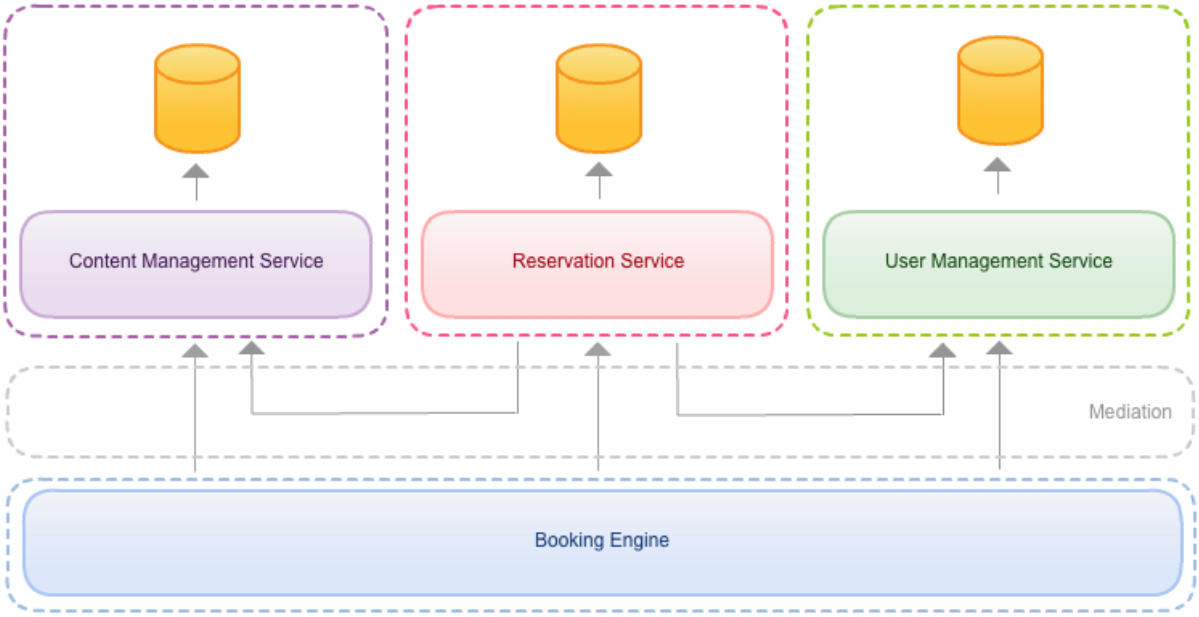


Figure 2.3: Implementing the System with SOA

The approach, as shown in Figure 2.3 is a better one as each sub-system (service) is self-contained. None of the services directly reads from another service's data store. All interactions between the services are wired through the Service APIs, which are well-defined and documented.

Challenges in implementing SOA

Here are a few challenges that you may face while implementing SOA:

- SOA-based environments can include many services that exchange messages. At times, managing how services interact can become complex. It becomes even more complicated when these services are delivered by different departments within the company or even different companies including partners and suppliers.
- SOA services are subject to network latency, network failure, and distributed system failures; however a local implementation is not. Therefore, you should consider those trade-offs while deciding to go for an SOA implementation.

- Another concern about services is that they can always fail their associated behaviors are subject to change. Hence, appropriate levels of exception handling and compensation logic must be associated with any service invocation.

  Also, per the needs of different clients of a service, you many need to maintain multiple versions of a service.

Lesson 2: AntiPatterns

Lesson Overview

In application development, a pattern is defined as a solution to a reoccurring problem within the application in various contexts. Patterns are a cohesion of good practices or good solutions that can be implemented and reused when designing applications to satisfy business requirements. Conceptually, antipatterns are similar to design patterns. While a design pattern describes how to solve a problem using good practices, antipatterns describe how to solve application-related problems by pointing out the bad practices in application development. The goal of using antipatterns while developing applications is to understand why the bad practices seem to be attractive, but are not effective, and how to replace the bad practices with good ones to overcome problems.

In this lesson, you will look at the commonly used antipatterns and understand why antipatterns are important.

Topic 1: What are Antipatterns

An antipattern is a common response to a recurring problem that is usually ineffective and can be counterproductive. Antipatterns present bad solutions in a manner that makes it easy for concerned people to understand the underlying problems and their consequences.

Antipatterns comprise a new research area, which is a derivative of design patterns. When this research began, it was assumed that it would be difficult to identify antipatterns. However, antipatterns were already common in the software industry. In fact, the industry had created and employed antipatterns since the invention of programmable computers.

Some believe antipatterns are a more effective way to communicate software knowledge than ordinary design patterns because of the following reasons:

- AntiPatterns clarify problems for software developers, architects, and managers by identifying the symptoms and consequences that lead to the dysfunctional software development processes.
- AntiPatterns convey the motivation for change and the need for refactoring poor processes.
- AntiPatterns are necessary to gain an understanding of common problems faced by most software developers. Learning from other developers' successes and failures is valuable and necessary.

Antipattern as a Solution

You have to remember that an antipattern is not a simple bad habit, practice, or idea. For an application solution to be an antipattern, it has to have two key elements:

- A common process, structure, or pattern that may initially seem to be an appropriate solution to a problem, but typically has more negative consequences than beneficial results.

- It should have an alternate solution that can be documented and proven to be effective in refactoring the code where necessary.

Antipatterns describe useful forms of software refactoring. Software refactoring is a form of code modification, used to improve the software structure to make it more extensible and make long−term maintenance possible. Good software structure is essential for system extension and maintenance.

Software refactoring is a great approach for improving software structure. The resulting structure does not have to resemble the original planned structure. When used properly, refactoring is a natural activity in the programming process.

Topic 2: Common Object-oriented Antipatterns

Object-Oriented antipatterns are the counter-productive techniques that can occur while designing applications using an object-oriented language. Let's now discuss a few of the popular ones. These include:

- Programming to Classes Rather than Interfaces
- Classes without OO
- Call-Super
- Blob or God Object
- Circular Dependency Antipattern
- Object Orgy
- Poltergeist Antipattern
- Yo-Yo Antipattern
- Sequential Coupling
- Swiss Army Knife
- Fear of Adding Classes

Here are some common object-oriented antipatterns.

Programming to Classes Rather than Interfaces

When developing an application, if the code is programmed around the interfaces, you can derive a lot of benefits. For example, the code will remain portable as you are not compelled to use only certain implementations only. You will also be able to modify the way your code behaves at run time- by changing the specifications of the run time environment. Many developers do not follow this practice and end with a rigid and difficult to implement code that needs a specific environment to run in. In applications where the programming revolves around classes, the behavior of the application is tied to the code and does not support necessary changes or modifications.

Classes without OO

Writing an application with a very large class that holds many methods and functions may be considered good functional design. However, this does not implement the OOP coding practices. Developers may declare all methods of a class as static methods which make the class seem like a package. This is also an abuse of constructing classes.

Call-Super

Call-Super is a design pattern in which a particular class makes it mandatory for a derived subclass to override a method and call back the overridden function at a particular point, as shown in Figure 2.3.

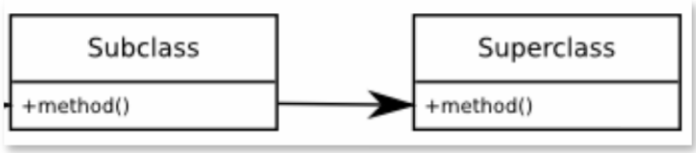


Figure 2.4: Call-Super

This is often required because the super class requires to perform some setup tasks for the class or framework to work correctly.

Antipattern deals with the requirement of calling the parent. There are many examples in real code where a method in a subclass may still want the functionality of the super class because in such a case, the method is only augmenting the parent functionality. If it still has to call the parent class even if it is fully replacing the functionality, there is the antipattern in force.

The problems with the Call-Super approach are:

- It expects you to override the method to make the functionality complete. However, it does so without making this expectation explicit. If you forget to call the super class, you will introduce unexpected errors.
- The intent of the programmer cannot be easily deduced from the code. If other team members need to work on it, they may not know what you expected the class to do, and therefore, are likely to encounter difficulty. In the chain of command, if a call to a super class fails, it becomes difficult to identify where the problem lies.

A better approach to solve these issues is to use the template method pattern instead, where the super class includes a purely abstract method that must be implemented by the subclasses and have the original method call the subclass method.

If you expect some parts of a method to be overridden, but other things need to stay in place, it is recommended to use the template method pattern. Here, you keep all the things that must not be replaced in one non-overridable (final) method, which calls other methods that must be implemented in order for the program to work. In this way, you can make sure that the template pattern clearly states what methods can be over-ridden and what cannot be. The important things remain where they have to be and anyone who extends the class will know exactly what to do, even if they do not know about the other implementation details.

Blob or God Object

The Blob or the god object antipattern consumes entire object-oriented architectures. Let us take the example of the movie, The Blob. In this movie, the blob is a drip-sized, jelly-like alien life form from outer space that comes to the Earth. Whenever it eats usually unsuspecting earthlings, it grows in size. The blob keeps growing and starts to threaten to wipe out the entire planet.

The movie is a good analogy for the Blob antipattern.

In object-oriented programming, a god object is an object that *knows too much* or *does too much*. The basic idea behind object-oriented programming is that a big problem is separated into several smaller problems (a divide and conquer strategy) and solutions are created for each of them. This means that there is only one object about which an object needs to know everything: *itself*. Likewise, there is only one set of problems an object needs to solve: its *own*.

A program that employs a god object does not follow this approach. A god-object holds so much data and methods that its role in the program becomes god-like (omnipotent). You will find the functionality of the program (with such an anti-pattern) to have its functionality coded into a single "all-knowing" object, which maintains most of the information about the entire program and provides most of the methods for manipulating this data. Instead of program objects communicating among themselves directly, the other objects within the program rely on the god object for most of their information and interaction. As the god object is tightly coupled to (referenced by) much of the other code, maintenance becomes more difficult than it would have been in a more evenly divided programming design.

The code shown in the example below has an object that does *everything*:

```
class God {
   function initialize() {}
   function readFile() {}
   function writeToFile() {}
   function display() {}
   function calculate() {}
   function validate() {}
   // and so on... //
}
```

In object-oriented programming, it would be preferable to have well-defined responsibilities for different objects to keep the code less coupled and ultimately more maintainable, as shown in the following code:

```
class FileInputOutput {
   function readFile() {}
   function writeToFile() {}
}

class UserInputOutput {
   function display() {}
   function validate() {}
}
```

```
class Logic {
   function initialize() {}
   function calculate() {}
}
```

Few Symptoms of God Object Antipattern

Few symptoms of the God object antipattern are:

- A single class with disparate collection of unrelated members and operations encapsulated in a single class.
- Blob limits the ability to modify the system without affecting the functionality of other encapsulated objects.
- It is greatly difficult to use the Blob class for reuse and testing. Since the Blob contains so much it could be very expensive to load it into memory.

Circular Dependency Antipattern

Circular dependency occurs when two or more classes, objects, or modules either directly or indirectly depend on each other to function properly. Such type of dependencies can cause many unwanted effects in software programs. Most problematic from a software design point of view is the ***tight coupling*** of the mutually dependent modules which reduces or makes it impossible to reuse a single module separately.

Programmers with not enough experience can introduce such type of circular dependencies, especially when implementing some kind of callback functionality. Mutual dependencies can be broken with the use of interfaces, like in the Observer pattern.

Object Orgy

This antipattern occurs when there is insufficient encapsulation in a system and objects just access each other's internal members directly rather than going through a public interface.

Consequences

The consequences of object orgy are:

- Difficult to find bugs because any part of the system could have potentially changed the state of an object to cause a bug
- Difficult to identify the behavior of an object because any part of the system can manipulate its internals
- Difficult to change the internal workings of a class because the change can affect many other parts of the system

Causes

Attributes of a class may be declared public to avoid the effort of providing proper accessors for them. This may well increase readability of the class itself, but at the expense of the consequences described above.

Poltergeist Antipattern

A poltergeist is a temporary and usually stateless object that is used to initialize or call the methods of a more permanent object. It is usually very short-lived and appears and disappears quickly. Poltergeists can sometimes be identified by their names, which often include the phrases *manager* or *controller*.

Imagine that you are building a banking system and have an account class to store information about accounts. Every time you need to create an account or make a transaction, you create an AccountManager object for that account which takes care of those objects by calling methods of the Account class.

In this case, AccountManager objects are poltergeists and appear and disappear as necessary. Instead, the functionality should be moved into the Account class and those methods should be called directly by clients. Sometimes, poltergeist classes are created because the programmer anticipated the need for a more complex architecture.

Consequences of Poltergeists

- Code may be difficult to understand and maintain since it can be hard to figure out where poltergeists come from and what they do.
- Resources are wasted because unnecessary objects have to be created and destroyed.

Poltergeists can be easily removed by deleting the poltergeist class and inserting its behavior into the class that it calls.

Yo-Yo Antipattern

The Yo-Yo antipattern occurs when a programmer works with a program where the inheritance hierarchy structure is deep and confusing such that the programmer has to constantly switch between classes in the hierarchy to keep track of what's happening in the program. It is called the Yo-Yo problem because the programmer's attention continually moves up and down the hierarchy when looking at different classes. This is one of the reasons why it is often recommended that inheritance hierarchies should be limited in depth to about six levels. If more than that, then it is recommended to use composition instead of inheritance.

Object-oriented design techniques, such as documenting layers of the inheritance hierarchy can reduce the effect of this problem because they collect the information that the programmer is required to understand and store it in one place.

Consequences of Yo-Yo antipattern

The deep and complicated class hierarchy in Yo-Yo antipattern makes it difficult to trace the flow of control in a program. This, in turn, makes the program harder to understand.

Sequential Coupling

This antipattern occurs when a class requires its methods to be called in a particular order. This can lead to problems especially if the contracts for these methods are not properly documented and do not mention the order in which methods should be called. Overall, it makes a class harder to read and understand and more error-prone.

This kind of antipattern can often be spotted by looking at method names. Names which include the words *start*, *begin,* or *end* may indicate sequential coupling.

For example, imagine that you are writing a class to read data from a file line by line. You make a startRead() method that opens the file and makes it ready for reading. The readLine() method then allows clients to read from the file line by line but only works if the file has been opened; that is if the startRead() method has been called. When the client no longer wants to read from the file, it needs to call the endRead() method that closes the file. If this is not done, the file remains open, wasting resources.

The methods of the class need to be called in a very particular order. In addition, you may need to set some conditions. For example, you cannot read from a file that is not open, you cannot close a file that is not open, and you cannot open a file that is already open. This creates more work and confusion than necessary for the client. It would be better if you encapsulated these conditions in the class itself rather than requiring the clients to keep track of the state of the file.

Consequences

A few consequences for sequential coupling are:

- Sequential coupling makes using a class harder and more error-prone for clients. It is easy to forget to call methods in the right order, potentially leading to bugs in the program.
- Clients are coupled to the class in a way that if the sequence of method calls changes, the clients will be affected by that change.

Swiss Army Knife

A Swiss Army Knife, is an anti-pattern where an interface is highly complex. The designer attempts to provide for all possible uses of the class. In the attempt, the designer adds a large number of interface signatures in a futile attempt to meet all possible needs.

The designer may not have a clear abstraction or purpose for the class, which is represented by the lack of focus in the interface.

Consequences:

- Makes the interface very difficult to understand how to use it and the classes that implement it.
- Makes debugging, documentation, and maintenance a very difficult job.

Fear of Adding Classes

New developers often feel that adding too many classes makes the code hard to read and compile. This may lead to having one massive class with too many methods, therefore, complicating the application code. Adding classes actually helps in simplifying code.

Topic 3: Programming Antipatterns

Programming antipatterns are applicable in every language, while common antipatterns are only applicable object-oriented languages. Here are a few programming antipatterns.

Blind Faith

Blind faith (also known as blind programming or blind coding) is a situation where a programmer develops a solution or fixes a computer bug and deploys it without ever testing the creation. The programmer, in this situation, has blind faith in own abilities.

Another form of blind faith is when a programmer calls a subroutine without checking the result. For example, a programmer calls a subroutine to save user-data on the hard disk without checking whether the operation was successful or not. In this case, the programmer has *blind faith* in the subroutine that it would perform as to the intention of the programmer.

Coding by Exception

Sometimes, programmers make use of exceptions to handle the error while the program is running and avoid crashing the system. Using these exceptions to handle specific errors that arise to continue the program is called coding by exception. This antipattern can quickly degrade software in performance and maintainability.

It is necessary that programmers use a generalized solution and not write code to specifically handle an error.

Cut and Paste Programming

Cut-and-Paste programming is a very common, but degenerate form of software reuse which creates maintenance nightmares. Inexperienced programmers often copy and paste code, as they find the act of writing code from scratch difficult and prefer to search for a pre-written solution.

Since the code often comes from disparate sources, such as friends or co-workers, Internet forums etc. it could result into multiple programming styles being introduced.

The Golden Hammer

When a particular technology or design pattern is overused while developing an application, the antipattern created is called the Golden Hammer. A developer or a team of developers who are

specialized in one particular approach to coding tend to rely on this method a lot. They perceive unfamiliar technology as risky and even if they are more appropriate to the situation. Resolving the Golden Hammer antipattern can be done, by constantly updating one's awareness of the latest technological advances and by not getting attached to any one design pattern or theory while developing applications.

Reinvent the Wheel (Development Related)

To reinvent the wheel is to duplicate a pattern, method, or framework which is already available in the market. At times, when developers are trying to customize an application to a given set of requirements, they end up creating services from the scratch instead of integrating with tools that already exist in the market and are compatible with their application's language. For example, instead of integrating the application with an ORM tool, such as Hibernate, if you write voluminous code within the application to handle data mapping, you are reinventing the wheel.

Vendor lock-in (Development Related)

When an application becomes too dependent on the tools or services provided by a particular vendor, vendor lock-in is created. The application is unable to use any other tool, product, or service without incurring substantial switch over costs. For example, an application that handles video decoding from camera recorded events gets locked in to using only one kind of decoder that is compatible with the software that controls the functioning of the camera.

Topic 4: Importance of Antipattern

The quality of an application is determined by two factors, the presence of good design practices and the absence of bad design practices or antipatterns. Therefore, to create efficient, structured, and flexible applications, it is just not enough to have good design patterns. It is also important to study antipatterns when researching software practices, such as application development. Developers should also learn to evaluate bad programming practices that cause antipatterns.

Following are some of the reasons due to which antipatterns is important:

- **Mapping a general situation to a specific class of solutions**: By using antipatterns, you can create a specialized class of solutions to a commonly faced development problem. Antipatterns have a general form that makes them easy to identify. Antipatterns are like templates that contain the problem in the code and the underlying cause for the problem. So a particular antipattern template can be used to develop a solution template that can be applied wherever the antipattern occurs. Therefore, antipatterns are important in fine-tuning the functionality of an application.

- **Providing a common vocabulary for problem identification**: Similar to design patterns, antipatterns outline an industry level vocabulary, such as Gold Plating and Golden hammer, for common defective processes and practices within organizations. This vocabulary establishes

communication between software practitioners and enables them to come up with high level application concepts.

- **Industry-wide support for sharing common problems**: While developing an application, it is often easier to recognize a defect than to implement a solution. While faced with such a situation, developers find it conducive to share the challenges they are facing with others in a similar situation. This can prove to be a catalyst for resolving a problem faced throughout the software industry.

To summarize the importance of antipatterns, you could say that they bring together the software development community to resolve common problems in development by identifying bad practices. These bad practices can then be resolved in such a way that the entire community benefits from the application of appropriate solutions.

Module Summary

In this module, you have been introduced to the following design patterns commonly used in the software industry:

- SOA
- Antipatterns

In the first lesson, you learnt about the Service Oriented Architecture which is based on distinct pieces of software providing application functionality as services to other applications. SOA is still an emerging technology and it has its own challenges. However, the advantages of SOA, by far, outweigh the challenges faced in implementing SOA.

In the second lesson, you learnt about antipatterns, which help to develop a sound and fail proof application in a way that can be called reverse engineering. Learning what not to do or what can go wrong is as important as learning how to do well and following good practices while developing an application.

References

Web references:

- http://msdn.microsoft.com/en-us/library/ee658090.aspx (accessed latest on May 12, 2014)
- http://msdn.microsoft.com/en-us/library/dn568100.aspx (accessed latest on May 12, 2014))
- http://www.eaipatterns.com/PipesAndFilters.html (accessed latest on May 12, 2014)
- http://sourcemaking.com/antipatterns (accessed latest on May 12, 2014)
- http://www.antipatterns.com/ (accessed latest on May 12, 2014)

Book references:

- Buschman, Frank; Rohnert, Hans; Sommerlad, Peter; Stal, Michael; Meunier, Regine. (2000). *Pattern Oriented* Architecture. Wiley Publishing.
- Bell, Peter. (2010). *Design Patterns: Exposing the Service Layer (Part 5)*: Apress
- Jorge Luis Ortega-Arjona . (). Patterns for Parallel Software Design. Wiley Publishing
- Brown, William H; Malveau, Raphael C. (2005). *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.* John Wiley and Sons, Inc.

Glossary

Term	Definition
Service-Oriented Architecture (SOA)	It is a layered structure that supports communications between services.
Service Layer	It is a layer of the SOA that contains services.
Architectural Building Blocks (ABBs)	It describes the capability and shapes the specification of Solution Building Blocks (SBBs)
Operational Systems Layer	It is a layer of the SOA that contains the runtime elements of the architecture.
Application Programming Interfaces (APIs)	It is a set of routines, protocols and tools used for building software applications.
ASM	This is a simple API that is used to decompose, modify and recreate binary Java classes. The term ASM is not an acronym but derived from the 'asm'' keyword from the C programming language.
Antipatterns	A general response to a frequent problem which is often useless and could also be detrimental
Common Base Event (CBE)	It is an IBM implementation of the Web Services Distributed Management (WSDM) Event Format standard.
Secure Socket Layer (SSL)	It is a protocol that is commonly used for ensuring the security of message transmission over a network.
Universal Discovery Description and Integration (UDDI)	It is an extensible mark-up language-based registry for businesses all over the world.
Idempotency	It refers to a situation where duplicate messages are received from the same consumer, but only the original has to be handled.
Commutatively	It refers to the order in which messages are received.
Cohesion	Cohesion refers to modules where the relation between the functionality of its components is strong, making the module robust, reliable and easy to understand.
Data sink	Data sink is a machine or a system that is capable of receiving data.
Decomposition	Software decomposition is the process by which a complex application is broken down into smaller components that are easier to understand, execute, and maintain.
Decryption	The process of decoding information that has been encoded in a secret, difficult to understand format is called decryption.
De-duping	The process by which repetitive units of code or duplicates of the same code segments are deleted in called de-duping.
Monolithic design pattern	Monolithic design pattern is which where the application performs a very small number of well defined and easy to

	implement functionalities. Applications with monolithic design do not have a complicated architecture or an inheritance based class structure.
Cut-and-paste development	The method of copying code from one source and pasting directly to another.
Golden Hammer	Excessive or fanatical use of a technology or model
Incomprehensible Codes	Codes that are 'unreadable' or whose functions are not known
Input Kludge	Software that misapplies basic user inputs
Monkey test	A unit test which runs with no specific goals is called a monkey test in software testing. It is a type of black box testing method that ensures that the application is capable of handling all possible user inputs.
Command objects	An object that is used to encapsulate all information needed to call a method.
Cocoa Touch framework	This framework drives iOS applications that are focused on touch based interfaces.
God Object	This is an object in an application that controls too many things and is a type of antipattern. This should not be used in an application.
iOS	The world's most advanced mobile operating system that is used in Apple machines.
Polymorphism	This is the provision of a single interface to entities of different types.
Singleton	This is a design pattern that confines the instantiation of a class to one object.