

Design Patterns Explained

Lecture #10
Software Engineering and
Project Management

Instructed by Steven Choy on Dec 4, 2006



1

What will you learn?

- I will cover the following commonly used:
 - Singleton (covered in previous lecture)
 - Observer (covered in previous lecture)
 - Abstract Factory
 - Factory Method
 - Adapter
 - Strategy
 - Bridge
 - Façade
- If we have time, we'll discuss some more...



2

The Strategy Pattern



3

Business rules change from time to time. How do you handle and prepare your system for this variation challenge?



4

Coping with Changes



- A mediocre programmer handles changes by:
 - Plenty of switch cases
 - A lot of if-then-else conditional statements
 - Inherit from the base class and provide extra functionalities
- What a great mind does is to use:
 - The Strategy Pattern

5

Strategy Pattern



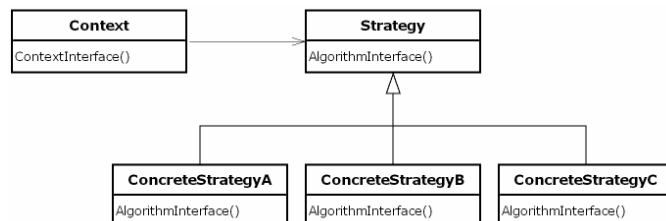
- What did GoF say?
 - Define a family of algorithms, **encapsulate** each one, and make them interchangeable. [The] Strategy [pattern] lets the algorithm vary independently from clients that use it.
- When is it applicable?
 - Allow your software to use different business requirements or algorithms depending on the context they occur
- Strategy pattern emphasizes two fundamental principles of OO design:
 - Program to an interface, not implementation
 - Encapsulate the concept that varies

6

Strategy Pattern



- How does it work?
 - Separate the selection of business rules/algorithms from the implementation of business rules/algorithms
 - Allows for the selection to be made based upon context

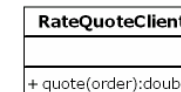


7

Strategy Pattern Example



- Consider you're the designer of an online shopping portal, any customers can query for shipment rate of its order via the system
- Currently, your company only liaise with one of the logistic companies for courier service and you encapsulate the calculation of shipment rate in RateQuoteClient object:



- One day, your boss asks you to change the system to support query of shipment rate from other logistic companies. How can you handle it?

8

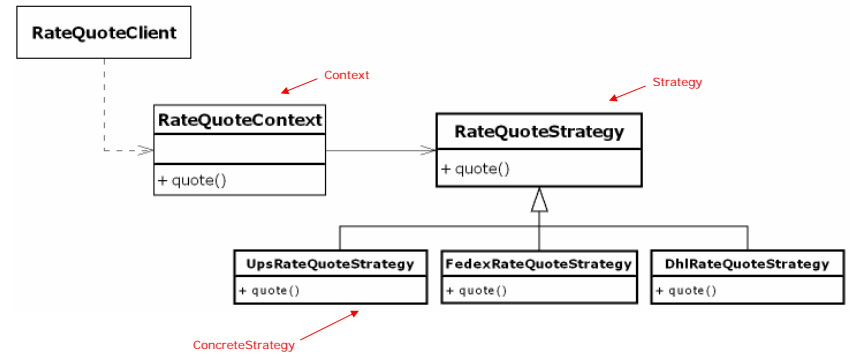
Strategy Pattern Example



- The obvious way is to use switch cases or if-else conditional statement
- But this makes the Order class complex
- Think if you have to support another new logistic provider, what do you need to alter?

```
class RateQuoteClient {
public double quote(Order order) {
switch(getLogisticProvider()) {
case UPS: UpsShipRateQuote(order);
break;
case DHL: DhlShipRateQuote(order);
break;
case FEDEX: FedexShipRateQuote(order);
break;
}
...
}
}
```

Bringing Strategy for Help



Code Sample...



```
abstract class RateQuoteStrategy {
public abstract double quote();
}
```

```
class UpsRateQuoteStrategy extends RateQuoteStrategy {
public double quote() {
System.out.println("[UpsRateQuoteStrategy] Quoting shipment rate...");
// Implementation omitted for demo purpose
// In real situation, implementation varies between different providers
return 100.0;
}
}
```

Code Sample...



```
abstract class RateQuoteStrategy {
public abstract double quote();
}
```

```
class DhlRateQuoteStrategy extends RateQuoteStrategy {
public double quote() {
System.out.println("[DhlRateQuoteStrategy] Quoting shipment rate...");
// Implementation omitted for demo purpose
// In real situation, implementation varies between different providers
return 99.5;
}
}
```

Code Sample...



```
abstract class RateQuoteStrategy {
    public abstract double quote();
}
```

```
class FedexRateQuoteStrategy extends RateQuoteStrategy {
    public double quote() {
        System.out.println("[FedexRateQuoteStrategy] Quoting shipment rate...");
        // Implementation omitted for demo purpose
        // In real situation, implementation varies between different providers
        return 110.0;
    }
}
```

13

Code Sample...



```
class RateQuoteContext {
    private RateQuoteStrategy strategy = null;
    public RateQuoteContext(String type) {
        if (type.equals("dhl")) {
            strategy = new DhlRateQuoteStrategy();
        } else if (type.equals("ups")) {
            strategy = new UpsRateQuoteStrategy();
        } else {
            strategy = new FedexRateQuoteStrategy();
        }
    }

    public double quote() {
        return strategy.quote();
    }
}
```

14

Code Sample...



```
public class RateQuoteClient {
    public static void main(String[] args) {
        RateQuoteContext context = new RateQuoteContext("dhl");
        System.out.println(context.quote());

        // Change to other strategy
        context = new RateQuoteContext("ups");
        System.out.println(context.quote());

        // Change to another strategy
        context = new RateQuoteContext("fedex");
        System.out.println(context.quote());
    }
}
```

15

The Façade pattern



16



Your system is growing with more subsystems.
It's too complex for client to learn how to
interact with the subsystems. You decide to
simplify the access of the existing subsystems.
How do you cope with it?

17

Façade Pattern



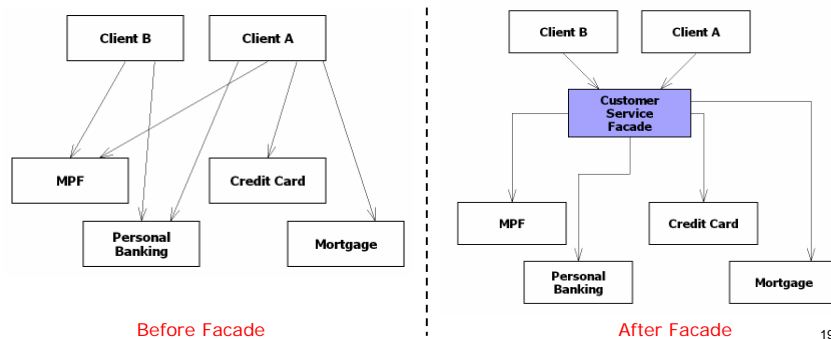
- What did GoF say?
 - Provide a unified interface to a set of interfaces in a subsystem
 - Façade defines a higher-level interface that makes the subsystem easier to use
- When is it applicable?
 - You need to simplify the interface for accessing subsystems
 - You need a central point to access the subsystems that aids:
 - Centralize security management
 - Provide common caching facility for the underlying system
 - Centralize transaction control
 - Etc...

18

Façade Pattern



- How does it work?
 - The Façade presents a new interface for the client to access the existing system



19

The Adapter pattern



20



You bought a new framework to replace your home-grown one. But you find that some of class interfaces do not compatible with your current one. How do you cope with it?

21

Adapter Pattern

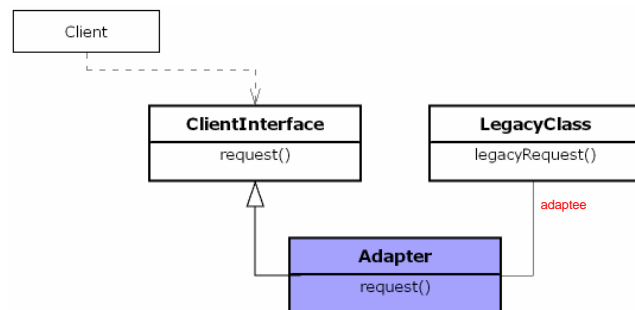


- What did GoF say?
 - Convert the interface of a class into another interface that the clients expect
- When is it applicable?
 - You want classes with incompatible interfaces to work together
- Also known as a wrapper

22

Adapter Pattern

- How does it work?
 - Adapter provides a wrapper with the desired interface



23

Adapter Pattern

- Implementation Issues
 - How much adaptation should be done?
 - Simply just a method name conversion
 - Incompatible method arguments
 - Totally different operations



24

Adapter Pattern Example



- Consider an example of online movie portal for ticket purchase, our system delegates the handling of credit card payment to a payment provider via a set of APIs
- Now your manager decides to switch to another payment provider, which offers lower service charges
- The new provider offers the same features but with APIs of incompatible interfaces with your current system. How do you adapt to the situation?

25

Adapter Pattern Example



```
Payment Client:
public void pay(PaymentProvider pp) {
    // Pre-processing operations
    pp.submitPayment(cc);
    // Post processing operations
}

Main Program:
public static void main(String[] args) {
    PaymentClient client = new PaymentClient();
    PaymentProvider pp = new ExistingProvider();
    client.pay(pp);
}

PaymentProvider Interface:
interface PaymentProvider {
    public void submitPayment(String cc);
}

Existing Payment Provider:
public class ExistingProvider implements
PaymentProvider {
    public void submitPayment(String cc) {
        // Implementation
    }
}
```

Code Snippet for the existing implementation

26

Adapter Pattern Example



- The new payment provider offers a different interface for submitPayment()
- The new method call is sendPayment()
- We need to adapt the old interface to the new one

```
New Payment Provider:
public class NewPaymentProvider {
    public void sendPayment(String cc) {
        // Implementation
    }
}
```

27

Adapter Pattern Example



```
Payment Client:
PaymentProvider pp = new
PaymentProviderAdapter(new
NewPaymentProvider());

public void pay() {
    // Pre-processing operations
    pp.submitPayment(cc);
    // Post processing operations
}

Main Program:
public static void main(String[] args) {
    PaymentClient client = new PaymentClient();
    client.pay(pp);
}

PaymentProvider Interface:
interface PaymentProvider {
    public void submitPayment(String cc);
}

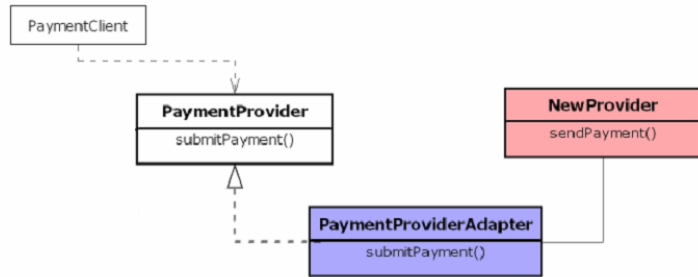
Adapter:
public class PaymentProviderAdapter
implements PaymentProvider {
    private NewPaymentProvider pp = new
NewPaymentProvider();

    public void submitPayment(String cc) {
        // Implementation
        pp.sendPayment(cc);
    }
}
```

Code Snippet for the existing implementation

28

Adapter Comes To Rescue



Adapter Pattern Solution – Code Sample



- We introduce a PaymentProviderAdapter to translate submitPayment() to the new sendPayment()

```
Adapter:
public class PaymentProviderAdapter implements PaymentProvider {
    private NewPaymentProvider pp;
    public PaymentProviderAdapter(p) {
        this.pp = p;
    }
    public void submitPayment(String cc) {
        // Implementation
        pp.sendPayment(cc);
    }
}
```

Factory Method Pattern



How do you defer instantiation of particular objects to derived classes?



Factory Method Pattern

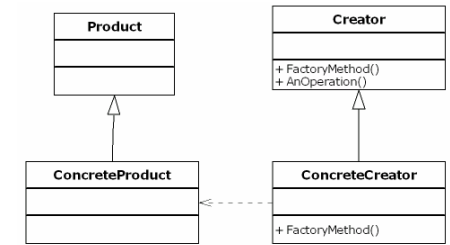


- What did GoF say?
 - Define an interface for creating an object, but let subclasses decide which class to instantiate
 - Let a class defer instantiation to subclasses
- When is it applicable?
 - You want to make object creation flexible (compared with using the "new" operator)
 - Factory method allows a derived class to make the decision on how to do instantiation
 - Normally used in defining a framework

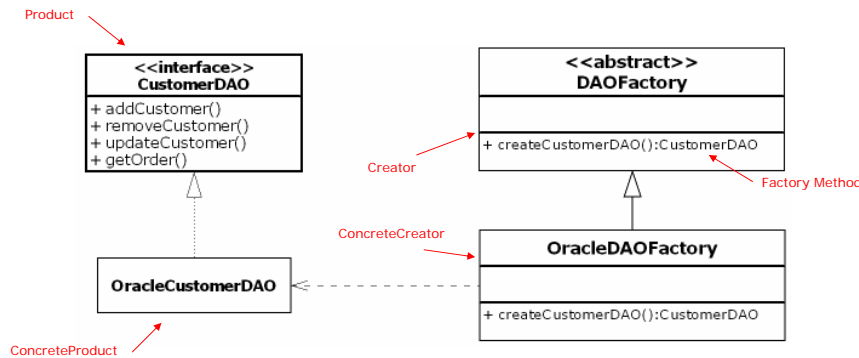
Factory Method Pattern



- How does it work?
 - Client does not use the **new** operator for direct object creation
 - An abstract Creator defines abstract Factory Method for object creation
 - A derived class makes decision on which class to instantiate and how to instantiate it



Factory Method Example #1



Code Sample: Abstract Factory



```
/**
 * DAO Abstract Factory (Creator)
 */
abstract class DAOFactory {
    // Factory method
    public abstract CustomerDAO createCustomerDAO();
}
```

Code Sample: Concrete Factory



```
/**
 * Concrete factory (Concrete Creator)
 */
class OracleDAOFactory extends DAOFactory {

    public CustomerDAO createCustomerDAO() {
        System.out.println("[OracleDAOFactory]
            Creating Oracle Customer DAO...");
        return new OracleCustomerDAO();
    }
}
```

37

Code Sample: Product



```
/**
 * Customer DAO interface (Product)
 */
interface CustomerDAO {
    public void addCustomer(Customer customer);
    public void removeCustomer(String customerId);
    public void updateCustomer(Customer customer);
    public String getOrder();
}
```

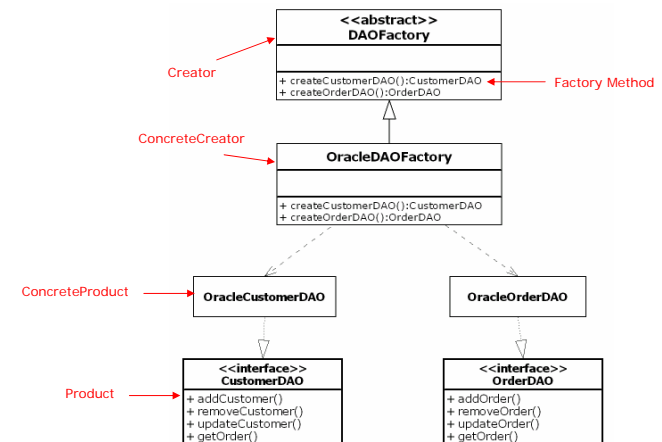
38

```
/**
 * Oracle customer DAO (Concrete Product)
 */
class OracleCustomerDAO implements CustomerDAO {
    public void addCustomer(Customer customer) {
        System.out.println("[OracleCustomerDAO] Adding customer...");
        // The actual implementation is omitted
    }
    public void removeCustomer(String customerId) {
        System.out.println("[OracleCustomerDAO] Removing customer...");
        // The actual implementation is omitted
    }
    public void updateCustomer(Customer customer) {
        System.out.println("[OracleCustomerDAO] Updating customer...");
        // The actual implementation is omitted
    }
    public String getOrder() {
        System.out.println("[OracleCustomerDAO] Retrieving order...");
        // The actual implementation is omitted
        return "";
    }
}
```



39

Factory Method Example #2



40

Factory Method Real-life Application



- Ample of examples for Factory Method Pattern in JDK
- Open the JavaDoc for JDK 1.4/5.0 and search for class name end with "Factory"

41

Abstract Factory Pattern



42

You may appreciate Factory Method on the beauty of object creation. But now you want more... Suppose your application need to further support multiple databases depending on client's environment.

How do you handle instantiation of families of objects?

43

Abstract Factory Pattern



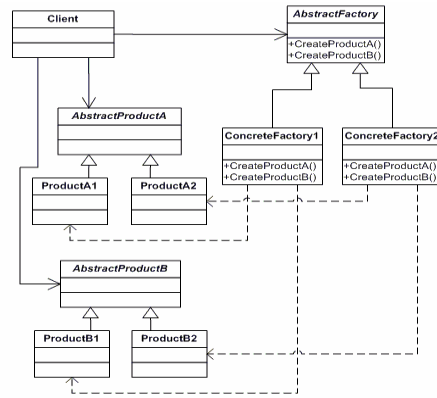
- What did GoF say?
 - Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- One step further than Factory Method to create families of objects
- When is it applicable?
 - Think about you have families of Data Access Objects such as OracleDAO, MySqlDAO and SybaseDAO
 - But you don't want the client to directly specify the concrete DAO class

44

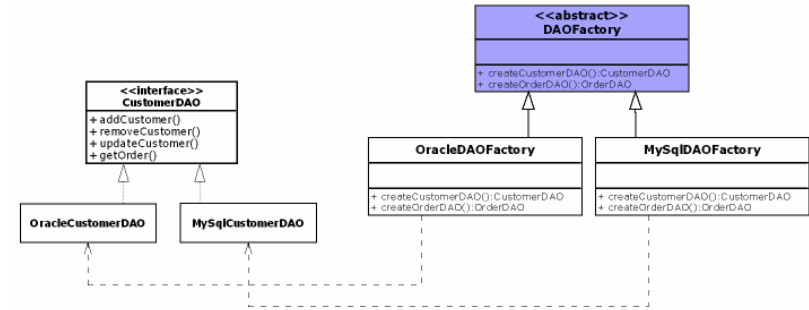
Abstract Factory Pattern



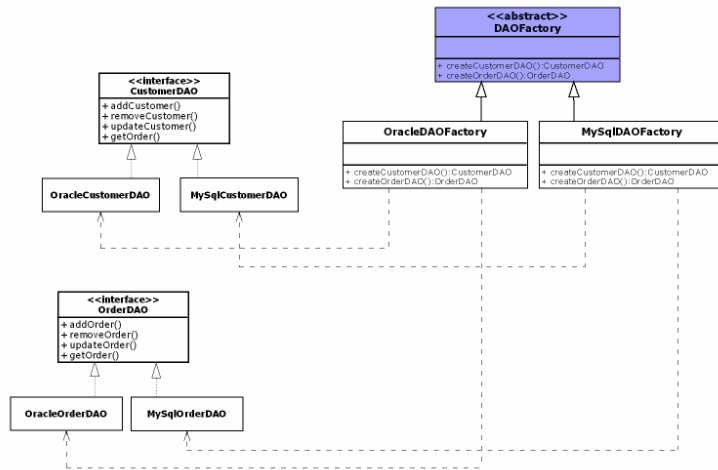
- How does it work?
 - Client does not use the **new** operator for direct object creation
 - Object creations are coordinated by a AbstractFactory



Abstract Factory Example #1



Abstract Factory Example #2



Bridge Pattern



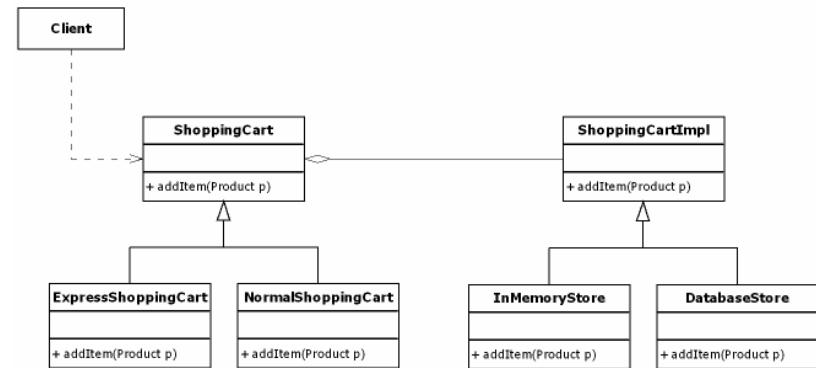
Bridge Pattern



- Intent: De-couple an abstraction from its implementation so that the two can vary independently.
- The Bridge pattern is one of the toughest patterns to understand because it is so powerful and applies to so many situations.

49

Bridge Pattern Sample



50

Design Pattern Summary



- Design for extensibility
 - Support for new vendor
 - Support for new implementation
 - Support for new views
- Design for portability
 - Support for other platforms

51

References



- GoF Patterns
<http://www.fluffycat.com/java/patterns.html>
- J2EE Patterns
<http://java.sun.com/blueprints/patterns/>

52