Design patterns

TS. Trịnh Tuấn Đạt
Bộ môn CNPM, Viện CNTT,
ĐHBK Hà Nội

Introduction

- In the late 70's, an architect named Christopher Alexander started the concept of patterns. Alexander's work focused on finding patterns of solutions to particular sets of forces within particular contexts
- Christopher Alexander was a civil engineer and an architect, his patterns were related to architects of buildings, but the work done by him inspired an interest in the object-oriented (OO) community

Introduction

- Design patterns represent the best practices used by experienced object-oriented software developers
- Design patterns are solutions to general problems that software developers faced during software development. These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time.

What is Gang of Four (GOF)

- In 1994, four authors Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides published a book titled **Design Patterns - Elements of Reusable Object-Oriented Software** which initiated the concept of Design Pattern in Software development

GoF patterns: three categories

- *Creational Patterns* – these abstract the object-instantiation process
  - `Factory Method, Abstract Factory, Singleton, Builder, Prototype`
- *Structural Patterns* – these abstract how objects/classes can be combined
  - `Adapter, Bridge, Composite, Decorator, Façade, Flyweight, Proxy`
- *Behavioral Patterns* – these abstract communication between objects
  - `Command, Interpreter, Iterator, Mediator, Observer, State, Strategy, Chain of Responsibility, Visitor, Template Method`

Main elements of a design pattern

- Pattern Name:
  - A common name to talk about
- Problem:
  - Context: when to apply the pattern
  - May include a list of conditions for applying the pattern
- Solution:
  - Abstract description of a design problem and how a general arrangement of elements solves it
  - Elements making up the design, their relationships/responsibilities and collaborations
  - Like a template, language-neutral
- Consequences:
  - Results and tradeoff of applying patterns
  - Impacts on system's flexibility, extensibility or portability

Part I: Creational Design Patterns

Singleton

Motivation

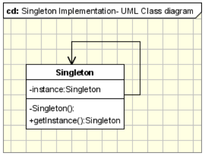
- Only one instance for a class?
- Centralized management of internal or external resources: provide a global point of access to themselves
- Only one class:
  - responsible to instantiate itself, to make sure it creates not more than one instance;
  - provides a global point of access to that instance

Intent

- Ensure that only one instance of a class is created.
- Provide a global point of access to the object

Implementation

```
class Singleton {
    private static Singleton instance;
    private Singleton() {
        ...
    }

    public static synchronized Singleton getInstance() {
        if (instance == null)
            instance = new Singleton();

        return instance;
    }
    ...
    public void doSomething() {
        ...
    }
}
```



Lazy instantiation using double locking mechanism

```
class Singleton {
    private static Singleton instance;

    private Singleton(){
        System.out.println("Singleton(): Initializing Instance");
    }

    public static Singleton getInstance(){
        if (instance == null){
            synchronized(Singleton.class){
                if (instance == null){
                    instance = new Singleton();
                }
            }
        }

        return instance;
    }

    public void doSomething(){
        System.out.println("doSomething(): Singleton does something!");
    }
}
```

Early instantiation using implementation with static field

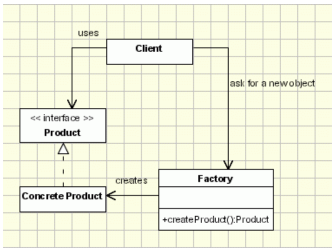
```
class Singleton{
    private static Singleton instance = new Singleton();

    private Singleton(){
        System.out.println("Singleton(): Initializing Instance");
    }

    public static Singleton getInstance(){
        return instance;
    }

    public void doSomething(){
        // …
    }
}
```

Factory

Intent

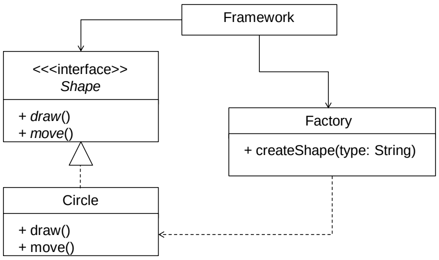
- creates objects without exposing the instantiation logic to the client.
- refers to the newly created object through a common interface

Implementation



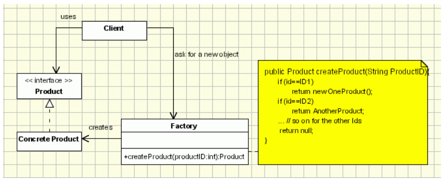
Example: a graphical framework works with shapes

- Client: drawing framework
- Product: Shape with 2 methods draw() and move()



Switch/case noob instantiation

```
public class ProductFactory{
    public Product createProduct(String ProductID){
        if (id==ID1)
                return new OneProduct();
        if (id==ID2) return
                return new AnotherProduct();
        ... // so on for the other Ids

      return null; //if the id doesn't have any of the expected values
    }
    ...
}
```



```
public Product createProduct(String ProductID){
    if (id==ID1)
        return new OneProduct();
    if (id==ID2)
        return AnotherProduct;
    ... // so on for the other Ids
    return null;
}
```

Class Registration - using reflection

```
class ProductFactory {
    private HashMap m_RegisteredProducts = new HashMap();
    public void registerProduct (String productID, Class productClass){
          m_RegisteredProducts.put(productID, productClass);
    }
    public Product createProduct(String productID){
          Class productClass = (Class)m_RegisteredProducts.get(productID);
          Constructor productConstructor = productClass.
                    getDeclaredConstructor(new Class[] {String.class});
          return (Product)productConstructor.newInstance(new Object[] { });
    }
}

public static void main(String args[]){
    Factory.instance().registerProduct("ID1", OneProduct.class);
}

class OneProduct extends Product{
    static {
          Factory.instance().registerProduct("ID1",OneProduct.class);
    }
    ...
}
```

To ensure correct class loading

```
class Main {
   static {
        try{
                Class.forName("OneProduct");
                Class.forName("AnotherProduct");
        }
        catch (ClassNotFoundException any){
                any.printStackTrace();
        }
   }
   public static void main(String args[]) throws
                PhoneCallNotRegisteredException{
        ...
   }
}
```

Class Registration – avoiding reflection

```
abstract class Product {
    public abstract Product createProduct();
    ...
}

class OneProduct extends Product {
    ...
    static  {
          ProductFactory.instance().registerProduct("ID1", new OneProduct());
    }
    public OneProduct createProduct() {
          return new OneProduct();
    }
    ...
}

class ProductFactory {
    private HashMap m_RegisteredProducts = new HashMap();

    public void registerProduct(String productID, Product p){
          m_RegisteredProducts.put(productID, p);
    }
    public Product createProduct(String productID){
          ((Product)m_RegisteredProducts.get(productID)).createProduct();
    }
}
```

Factory Method

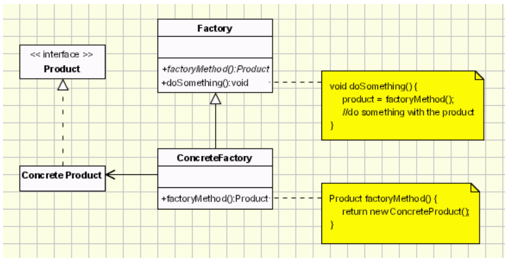
Motivation

- Also known as Virtual Constructor
- Similar to the idea of a library:
  - a library uses abstract classes for defining and maintaining relations between objects. One type of responsibility is creating such objects
  - the library knows when an object needs to be created, but not what kind of object it should create, this being specific to the application using the library

Intent

- Defines an interface for creating objects, but let subclasses to decide which class to instantiate
- Refers to the newly created object through a common interface

Implementation



```
void doSomething() {
    product = factoryMethod();
    //do something with the product
}
```

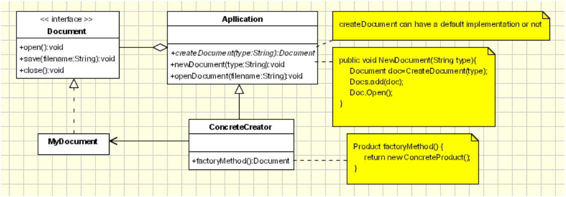
```
Product factoryMethod() {
    return new ConcreteProduct();
}
```

Source

```
public interface Product { }

public abstract class Creator {
    public void anOperation() {
        Product product = factoryMethod();
    }
    protected abstract Product factoryMethod();
}

public class ConcreteProduct implements Product { }

public class ConcreteCreator extends Creator {
    protected Product factoryMethod() {
        return new ConcreteProduct();
    }
}

public class Client {
    public static void main( String arg[] ) {
        Creator creator = new ConcreteCreator();
        creator.anOperation();
    }
}
```

Example: a framework for desktop applications

- A framework works with documents: opening, creating and saving a document



<< interface >>
Document
+open():void
+save(filename:String):void
+close():void
Aplication
+createDocument(type:String):Document
+newDocument(type:String):void
+openDocument(filename:String):void
ConcreteCreator
+factoryMethod():Document
MyDocument

createDocument can have a default implementation or not

```
public void NewDocument(String type){
    Document doc=CreateDocument(type);
    Docs.add(doc);
    Doc.Open();
}
```

```
Product factoryMethod() {
    return new ConcreteProduct();
}
```

```
public Document createDocument(String type){
    if (type.isEqual("html"))
        return new HtmlDocument();
    if (type.isEqual("proprietary"))
        return new MyDocument();
    if (type.isEqual("pdf"))
        return new PdfDocument ();
}

public void newDocument(String type){
    Document doc=CreateDocument(type);
    docs.add(doc);
    doc.open();
}
```

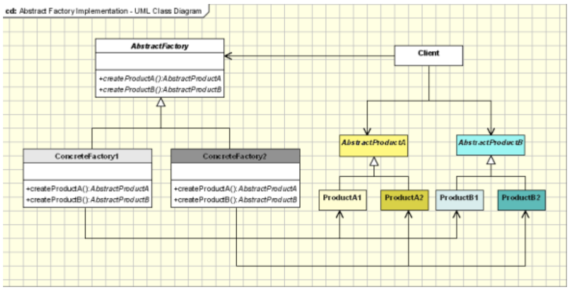
Abstract Factory

- Modularization is a big issue in today's programming
- Programmers try to avoid adding code to existing classes in order to make them support encapsulating more general information

Intent

- Abstract Factory offers the interface for creating a family of related objects, without explicitly specifying their classes

Implementation


cd: Abstract Factory Implementation - UML Class Diagram
AbstractFactory
+createProductA():AbstractProductA
+createProductB():AbstractProductB
Client
ConcreteFactory1
+createProductA():AbstractProductA
+createProductB():AbstractProductB
ConcreteFactory2
+createProductA():AbstractProductA
+createProductB():AbstractProductB
AbstractProductA
AbstractProductB
ProductA1
ProductA2
ProductB1
ProductB2

Implementation

```
abstract class AbstractProductA{
    public abstract void operationA1();
    public abstract void operationA2();
}

class ProductA1 extends AbstractProductA{
    ProductA1(String arg){
          System.out.println("Hello "+arg);
    } // Implement the code here
    public void operationA1() { };
    public void operationA2() { };
}

class ProductA2 extends AbstractProductA{
    ProductA2(String arg){
          System.out.println("Hello "+arg);
    } // Implement the code here
    public void operationA1() { };
    public void operationA2() { };
}
```

Implementation

```
abstract class AbstractProductB{
    //public abstract void operationB1();
    //public abstract void operationB2();
}

class ProductB1 extends AbstractProductB{
    ProductB1(String arg){
          System.out.println("Hello "+arg);
    } // Implement the code here
}

class ProductB2 extends AbstractProductB{
    ProductB2(String arg){
          System.out.println("Hello "+arg);
    } // Implement the code here
}
```

Implementation

```
abstract class AbstractFactory{
    abstract AbstractProductA createProductA();
    abstract AbstractProductB createProductB();
}

class ConcreteFactory1 extends AbstractFactory{
    AbstractProductA createProductA(){
          return new ProductA1("ProductA1");
    }
    AbstractProductB createProductB(){
          return new ProductB1("ProductB1");
    }
}

class ConcreteFactory2 extends AbstractFactory{
    AbstractProductA createProductA(){
          return new ProductA2("ProductA2");
    }
    AbstractProductB createProductB(){
          return new ProductB2("ProductB2");
    }
}
```

Implementation

```
//Factory creator - an indirect way of instantiating the factories
class FactoryMaker{
    private static AbstractFactory pf=null;
    static AbstractFactory getFactory(String choice){
          if(choice.equals("a")){
                    pf=new ConcreteFactory1();
          } else if(choice.equals("b")){
                    pf=new ConcreteFactory2();
          }
          return pf;
    }
}

// Client
public class Client{
    public static void main(String args[]){
          AbstractFactory pf=FactoryMaker.getFactory("a");
          AbstractProductA product=pf.createProductA();
          //more function calls on product
    }
}
```

Example 1: Personal Information Manager

```
interface AddressFactory{
    public Address createAddress();
    public PhoneNumber createPhoneNumber();
}
```

```
abstract class Address{
   private String street;
   private String city;
   private String region;
   private String postalCode;

   public static final String EOL_STRING =  System.getProperty("line.separator");
   public static final String SPACE = " ";

   public String getStreet(){ return street; }
   public String getCity(){ return city; }
   public String getPostalCode(){ return postalCode; }
   public String getRegion(){ return region; }
   public abstract String getCountry();

   public String getFullAddress(){
      return street + EOL_STRING + city + SPACE + postalCode + EOL_STRING;
   }

   public void setStreet(String newStreet){ street = newStreet; }
   public void setCity(String newCity){ city = newCity; }
   public void setRegion(String newRegion){ region = newRegion; }
   public void setPostalCode(String newPostalCode){ postalCode = newPostalCode; }
}
```

```
abstract class PhoneNumber{
   private String phoneNumber;
   public abstract String getCountryCode();

   public String getPhoneNumber(){ return phoneNumber; }

   public void setPhoneNumber(String newNumber){
      try{
         Long.parseLong(newNumber);
         phoneNumber = newNumber;
      }
      catch (NumberFormatException exc){
      }
   }
}
```

```
class USAddressFactory implements AddressFactory{
   public Address createAddress(){
      return new USAddress();
   }

   public PhoneNumber createPhoneNumber(){
      return new USPhoneNumber();
   }
}
```

```
class USAddress extends Address{
   private static final String COUNTRY = "UNITED STATES";
   private static final String COMMA = ",";

   public String getCountry(){ return COUNTRY; }

   public String getFullAddress(){
      return getStreet() + EOL_STRING +
         getCity() + COMMA + SPACE + getRegion() +
         SPACE + getPostalCode() + EOL_STRING +
         COUNTRY + EOL_STRING;
   }
}
```

```
class USPhoneNumber extends PhoneNumber{
   private static final String COUNTRY_CODE = "01";
   private static final int NUMBER_LENGTH = 10;

   public String getCountryCode(){ return COUNTRY_CODE; }

   public void setPhoneNumber(String newNumber){
      if (newNumber.length() == NUMBER_LENGTH){
         super.setPhoneNumber(newNumber);
      }
   }
}
```

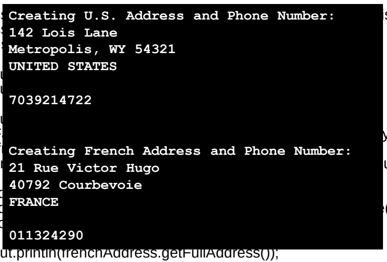
```
class FrenchAddressFactory implements AddressFactory{
   public Address createAddress(){
      return new FrenchAddress();
   }

   public PhoneNumber createPhoneNumber(){
      return new FrenchPhoneNumber();
   }
}
```

```
class FrenchAddress extends Address{
   private static final String COUNTRY = "FRANCE";

   public String getCountry(){ return COUNTRY; }

   public String getFullAddress(){
      return getStreet() + EOL_STRING +
         getPostalCode() + SPACE + getCity() +
         EOL_STRING + COUNTRY + EOL_STRING;
   }
}
```

```
class FrenchPhoneNumber extends PhoneNumber{
   private static final String COUNTRY_CODE = "33";
   private static final int NUMBER_LENGTH = 9;

   public String getCountryCode(){ return COUNTRY_CODE; }

   public void setPhoneNumber(String newNumber){
      if (newNumber.length() == NUMBER_LENGTH){
         super.setPhoneNumber(newNumber);
      }
   }
}
```

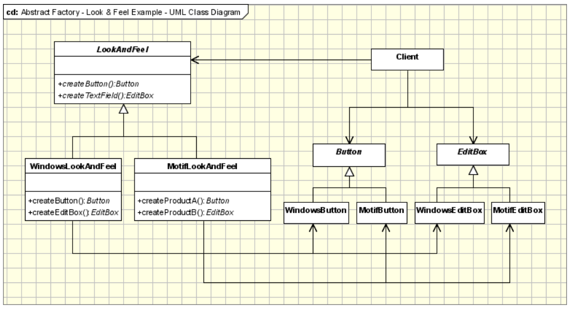
```
public class RunAbstractFactoryPattern {
  public static void main(String [] arguments){
     System.out.println("Creating U.S. Address and Phone Number:");
     AddressFactory usAddressFactory = new USAddressFactory();
     Address usAddress = usAddressFactory.createAddress();
     PhoneNumber usPhone = usAddressFactory.createPhoneNumber();

     usAddress.setStreet("142 Lois Lane"); usAddress.setCity("Metropolis");
     usAddress.setRegion("WY"); usAddress.setPostalCode("54321");
     usPhone.setPhoneNumber("7039214722");

     System.out.println(usAddress.getFullAddress());
     System.out.println(usPhone.getPhoneNumber());

     System.out.println("Creating French Address and Phone Number:");
     AddressFactory frenchAddressFactory = new FrenchAddressFactory();
     Address frenchAddress = frenchAddressFactory.createAddress();
     PhoneNumber frenchPhone = frenchAddressFactory.createPhoneNumber();

     frenchAddress.setStreet("21 Rue Victor Hugo");
     frenchAddress.setCity("Courbevoie"); frenchAddress.setPostalCode("40792");
     frenchPhone.setPhoneNumber("011324290");

     System.out.println(frenchAddress.getFullAddress());
     System.out.println(frenchPhone.getPhoneNumber());
  }
}
```

```
public class RunAbstractFactoryPattern {
  public static void main(String [] arguments){
     System.out.println("Creating U.S. Address and Phone Number:");
     AddressFactory usAddressFactory = new USAddressFactory();
     Address usAddress = usAddressFactory.createAddress();
     PhoneNumber usPhone = usAddressFactory.createPhoneNumber();

     usAddres                                              s");
     usAddres
     usPhone.

     System.ou
     System.ou

     System.ou
     AddressF                                              y();
     Address f
     PhoneNu                                               umber();

     frenchAdd
     frenchAdd                                             ("40792");
     frenchPho

     System.out.println(frenchAddress.getFullAddress());
     System.out.println(frenchPhone.getPhoneNumber());
  }
}
```



```
Creating U.S. Address and Phone Number:
142 Lois Lane
Metropolis, WY 54321
UNITED STATES

7039214722


Creating French Address and Phone Number:
21 Rue Victor Hugo
40792 Courbevoie
FRANCE

011324290
```

Example 2: Look & Feel


cd: Abstract Factory - Look & Feel Example - UML Class Diagram
LookAndFeel
+createButton():Button
+createTextField():EditBox
Client
WindowsLookAndFeel
+createButton(): Button
+createEditBox(): EditBox
MotifLookAndFeel
+createProductA(): Button
+createProductB(): EditBox
Button
EditBox
WindowsButton
MotifButton
WindowsEditBox
MotifEditBox

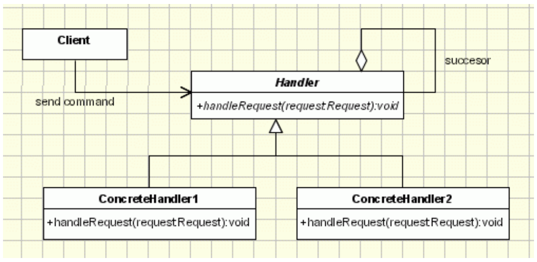
Part II: Behavioral Patterns

Chain of Responsibility

Motivation

- The event generated by one object needs to be handled by another one
- Two possibilities:
  - beginner/lazy approach: making everything public, creating reference to every object and continuing from there
  - expert approach: using the Chain of Responsibility
- The Chain of Responsibility: allows an object to send a command without knowing what object will receive and handle it.

Implementation


Client
send command
Handler
+handleRequest(requestRequest):void
succesor
ConcreteHandler1
+handleRequest(requestRequest):void
ConcreteHandler2
+handleRequest(requestRequest):void

Implementation – classic example

```
public class Request {
    private int m_value;
    private String m_description;

    public Request(String description, int value){
          m_description = description;
          m_value = value;
    }

    public int getValue(){
          return m_value;
    }

    public String getDescription()    {
          return m_description;
    }
}
```

Implementation – classic example

```
public abstract class Handler {
    protected Handler m_successor;
    public void setSuccessor(Handler successor){
          m_successor = successor;
    }

    public abstract void handleRequest(Request request){
          if (m_successor!= null)
                    m_successor.handle(request);
    }
}
```

Implementation – classic example

```
public class ConcreteHandlerOne extends Handler{
    public void handleRequest(Request request){
          if (request.getValue() < 0){ // if request is eligible, handle it
              System.out.println("Negative values are“
                    + "handled by ConcreteHandlerOne:");
              System.out.println("\tConcreteHandlerOne.HandleRequest: "
                    + request.getDescription() + request.getValue());
          } else {
                    super.handleRequest(request);
          }
    }
 }
```

Implementation – classic example

```
public class ConcreteHandlerTwo extends Handler{
    public void handleRequest(Request request){
          if (request.getValue() > 0){ // if request is eligible, handle it
              System.out.println("Positive values are"
                    + "handled by ConcreteHandlerTwo:");
              System.out.println("\tConcreteHandlerTwo.HandleRequest: "
                    + request.getDescription() + request.getValue());
          } else {
                    super.handleRequest(request);
          }
    }
}
```

Implementation – classic example

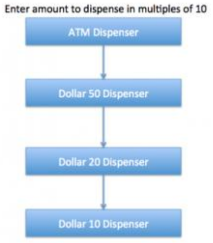
```
public class ConcreteHandlerThree extends Handler{
    public void handleRequest(Request request){
          if (request.getValue() >= 0){ // if request is eligible, handle it
              System.out.println("Zero values are"
                    + "handled by ConcreteHandlerThree:");
              System.out.println("\tConcreteHandlerThree.HandleRequest: "
                    + request.getDescription() + request.getValue());
          } else {
                    super.handleRequest(request);
          }
    }
}
```

Implementation – classic example

```
public class Main {
    public static void main(String[] args){
          // Setup Chain of Responsibility
          Handler h1 = new ConcreteHandlerOne();
          Handler h2 = new ConcreteHandlerTwo();
          Handler h3 = new ConcreteHandlerThree();
          h1.setSuccessor(h2);
          h2.setSuccessor(h3);

          // Send requests to the chain
          h1.handleRequest(new Request("Negative Value ", -1));
          h1.handleRequest(new Request("Zero Value ",  0));
          h1.handleRequest(new Request("Positive Value ",  1));
          h1.handleRequest(new Request(" Positive Value ",  2));
          h1.handleRequest(new Request("Negative Value ", -5));
    }
}
```

Example - ATM Dispense machine

- If the user enters an amount that is not multiples of 10, it throws error



Example - ATM Dispense machine

```
public class Currency {

    private int amount;

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

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

public interface DispenseChain {

    void setNextChain(DispenseChain nextChain);

    void dispense(Currency cur);
}
```

Example - ATM Dispense machine

```
public class Dollar50Dispenser implements DispenseChain {
    private DispenseChain chain;
    @Override
    public void setNextChain(DispenseChain nextChain) {
          this.chain=nextChain;
    }

    @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.chain.dispense(new Currency(remainder));
          } else{
              this.chain.dispense(cur);
          }
    }
}
```

Example - ATM Dispense machine

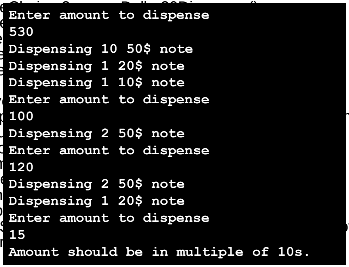
```
public class Dollar20Dispenser implements DispenseChain{
    private DispenseChain chain;
    @Override
    public void setNextChain(DispenseChain nextChain) {
          this.chain=nextChain;
    }

    @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.chain.dispense(new Currency(remainder));
          } else{
              this.chain.dispense(cur);
          }
    }
}
```

Example - ATM Dispense machine

```
public class Dollar10Dispenser implements DispenseChain {
    private DispenseChain chain;
    @Override
    public void setNextChain(DispenseChain nextChain) {
          this.chain=nextChain;
    }

    @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.chain.dispense(new Currency(remainder));
          } else{
              this.chain.dispense(cur);
          }
    }
}
```

```
public class ATMDispenseChain {
    private DispenseChain c1;
    public ATMDispenseChain() {
          // initialize the chain
          this.c1 = new Dollar50Dispenser();
          DispenseChain c2 = new Dollar20Dispenser();
          DispenseChain c3 = new Dollar10Dispenser();
          // set the chain of responsibility
          c1.setNextChain(c2);
          c2.setNextChain(c3);
    }
    public static void main(String[] args) {
          ATMDispenseChain atmDispenser = new ATMDispenseChain();
          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));
          }
    }
}
```

```
public class ATMDispenseChain {
    private DispenseChain c1;
    public ATMDispenseChain() {
          // initialize the chain
          this.c1 = new Dollar50Dispenser();
          Dispense
          Dispense
          // set the
          c1.setNe
          c2.setNe
    }
    public static v                                                    ain();
          ATMDisp
          while (tru
             int amo
             System
             Scanne
             amoun
             if (amo
                    S                                                  f 10s.");
                    r
             }
             // process the request
             atmDispenser.c1.dispense(new Currency(amount));
          }
    }
}
```



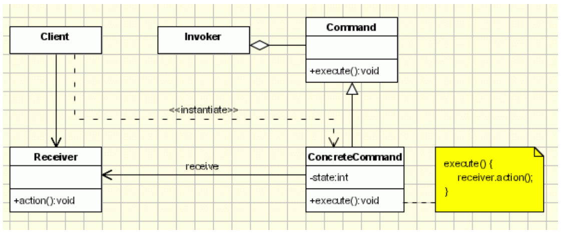
```
Enter amount to dispense
530
Dispensing 10 50$ note
Dispensing 1 20$ note
Dispensing 1 10$ note
Enter amount to dispense
100
Dispensing 2 50$ note
Enter amount to dispense
120
Dispensing 2 50$ note
Dispensing 1 20$ note
Enter amount to dispense
15
Amount should be in multiple of 10s.
```

Command

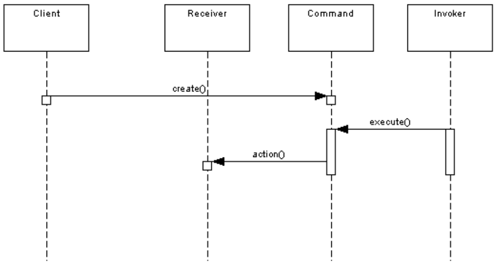
Motivation

- Command design pattern
  - is used to implement **loose coupling** in a request-response model
  - encapsulates commands/requests in objects.
  - provides the options to queue commands, undo/redo actions and other manipulations
- Where the Chain of Responsibility pattern forwarded requests along a chain, the Command pattern forwards the request to a specific module

Implementation



Implementation



Example 1 – Remote Control

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

Example 1 – Remote Control

```
//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();
  }
}

//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();
  }
}
```

Example 1 – Remote Control

```
//Receiver
public class Light{
   private boolean on;
   public void switchOn(){
      on = true;
   }

   public void switchOff(){
      on = false;
   }
}
```

Example 1 – Remote Control

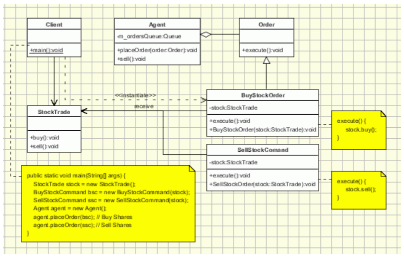
```
//Invoker
public class RemoteControl{
   private Command command;
   public void setCommand(Command command){
        this.command = command;
   }

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

Example 1 – Remote Control

```
public class Client{
    public static void main(String[] args)    {
          Light light = new Light();
          Command lightsOn = new LightsOnCommand(light);
          Command lightsOff = new LightsOffCommand(light);

          RemoteControl control = new RemoteControl();

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

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

Example 2 – StockTrading

```
public interface Order {
    public abstract void execute ( );
}
```



```
public static void main(String[] args) {
    StockTrade stock = new StockTrade();
    BuyStockCommand bsc = new BuyStockCommand(stock);
    SellStockCommand ssc = new SellStockCommand(stock);
    Agent agent = new Agent();
    agent.placeOrder(bsc); // Buy Shares
    agent.placeOrder(ssc); // Sell Shares
}
```

```
execute() {
    stock.buy();
}
```

```
execute() {
    stock.sell();
}
```

Example 2 – StockTrading

```
// Receiver class.
class StockTrade {
    public void buy() {
        System.out.println("You want to buy stocks");
    }
    public void sell() {
        System.out.println("You want to sell stocks ");
    }
}
```

Example 2 – StockTrading

```
// Invoker.
class Agent {
    private ArrayList m_ordersQueue = new ArrayList();

    public Agent() {
    }

    void placeOrder(Order order) {
          ordersQueue.addLast(order);
          Calendar date = Calendar.getInstance();
          if(date.get(Calendar.DAY_OF_WEEK) != Calendar.MONDAY) {
               while (m_ordersQueue.size()!=0){
                    order.execute(ordersQueue.get(0));
                    ordersQueue.remove(0);
               }
          }
    }
}
```

Example 2 – StockTrading

```
//ConcreteCommand Class.
class BuyStockOrder implements Order {
    private StockTrade stock;
    public BuyStockOrder ( StockTrade st) {
        stock = st;
    }
    public void execute( ) {
        stock . buy( );
    }
}

//ConcreteCommand Class.
class SellStockOrder implements Order {
    private StockTrade stock;
    public SellStockOrder ( StockTrade st) {
        stock = st;
    }
    public void execute( ) {
        stock . sell( );
    }
}
```

Example 2 – StockTrading

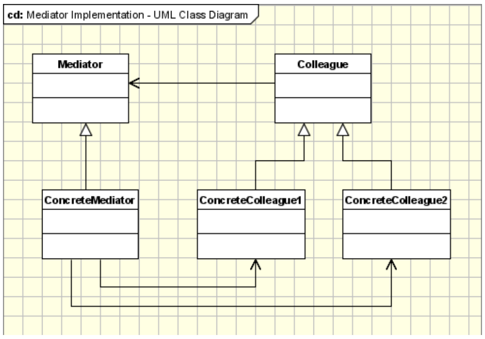
```
// Client
public class Client {
    public static void main(String[] args) {
        StockTrade stock = new StockTrade();
        BuyStockOrder bsc = new BuyStockOrder (stock);
        SellStockOrder ssc = new SellStockOrder (stock);
        Agent agent = new Agent();

        agent.placeOrder(bsc); // Buy Shares
        agent.placeOrder(ssc); // Sell Shares
    }
}
```

Mediator

Motivation

- In OO world, lots of classes interact with each other → final framework will end in a total mess where each object relies on many other objects in order to run
- We need a mechanism to facilitate the interaction between objects in a manner in that objects are not aware of the existence of other objects

Implementation



Example – Chatroom

```
//Mediator interface
public interface Mediator {
    public void send(String message, Colleague originator);
}
```

Example – Chatroom

```
//Colleague interface
public abstract Colleague{
    private Mediator mediator;
    public Colleague(Mediator m) {
          mediator = m;
    }
    //send a message via the mediator
    public void send(String message) {
          mediator.send(message, this);
    }
    //get access to the mediator
    public Mediator getMediator() {return mediator;}
    public abstract void receive(String message);
}
```

Example – Chatroom

```
public class ApplicationMediator implements Mediator {
    private ArrayList<Colleague> colleagues;
    public ApplicationMediator() {
          colleagues = new ArrayList<Colleague>();
    }
    public void addColleague(Colleague colleague) {
          colleagues.add(colleague);
    }
    public void send(String message, Colleague originator) {
          //let all other screens know that this screen has changed
          for(Colleague colleague: colleagues) {
                //don't tell ourselves
                if(colleague != originator) {
                      colleage.receive(message);
                }
          }
    }
}
```

Example – Chatroom

```
public class ConcreteColleague extends Colleague {
    public void receive(String message) {
          System.out.println("Colleague Received: " + message);
    }
}


public class MobileColleague extends Colleague {
    public void receive(String message) {
          System.out.println("Mobile Received: " + message);
    }
}
```

Example – Chatroom

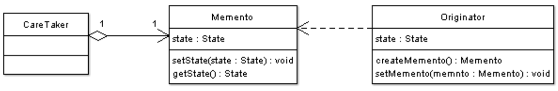
```
public class Client {
    public static void main(String[] args) {
          ApplicationMediator mediator = new ApplicationMediator();
          ConcreteColleague desktop = new ConcreteColleague(mediator);
          ConcreteColleague mobile = new MobileColleague(mediator);
          mediator.addColleague(desktop);
          mediator.addColleague(mobile);
          desktop.send("Hello World");
          mobile.send("Hello");
    }
}
```

Memento

Motivation

- Capture and externalize an object's internal state without violating encapsulation and thus providing a mean for restoring the object into initial state when needed.

Implementation



CareTaker

1

1

Memento

state : State

setState(state : State) : void

getState() : State

Originator

state : State

createMemento() : Memento

setMemento(memnto : Memento) : void

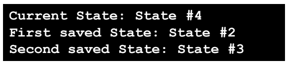
Example

```
public class Originator {
    private String state;
    public void setState(String state){
        this.state = state;
    }
    public String getState(){
        return state;
    }
    public Memento saveStateToMemento(){
        return new Memento(state);
    }
    public void getStateFromMemento(Memento Memento){
        state = Memento.getState();
    }

    public static class Memento {
        private final String state;

        private Memento(String state){
            this.state = state;
        }

        private String getState(){
            return state;
        }
    }
}
```

Example

```
import java.util.ArrayList;
import java.util.List;

public class CareTaker {
   private List<Originator.Memento> mementoList;

   public CareTaker(){
      mementoList = new ArrayList<Originator.Memento>();
   }
   public void add(Originator.Memento state){
      mementoList.add(state);
   }

   public Originator.Memento get(int index){
      return mementoList.get(index);
   }
}
```

Example



```
Current State: State #4
First saved State: State #2
Second saved State: State #3
```

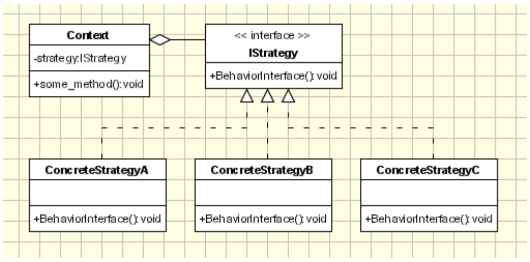
```
public class MementoPatternDemo {
   public static void main(String[] args) {

      Originator originator = new Originator();
      CareTaker careTaker = new CareTaker();

      originator.setState("State #1");
      originator.setState("State #2");
      careTaker.add(originator.saveStateToMemento());

      originator.setState("State #3");
      careTaker.add(originator.saveStateToMemento());

      originator.setState("State #4");
      System.out.println("Current State: " + originator.getState());

      originator.getStateFromMemento(careTaker.get(0));
      System.out.println("First saved State: " + originator.getState());
      originator.getStateFromMemento(careTaker.get(1));
      System.out.println("Second saved State: " + originator.getState());
   }
}
```

Strategy

Motivation

- Define a family of algorithms, encapsulate each one, and make them interchangeable.
- Strategy lets the algorithm vary independently from clients that use it

Implementation



Example – File Compression Tool

```
//Strategy Interface
public interface CompressionStrategy {
  public void compressFiles(ArrayList<File> files);
}

public class ZipCompressionStrategy implements CompressionStrategy {
  public void compressFiles(ArrayList<File> files) {
    //using ZIP approach
  }
}

public class RarCompressionStrategy implements CompressionStrategy {
  public void compressFiles(ArrayList<File> files) {
    //using RAR approach
  }
}
```

Example – File Compression Tool

```
public class CompressionContext {
    private CompressionStrategy strategy;
    //this can be set at runtime by the application preferences
    public void setCompressionStrategy(CompressionStrategy strategy) {
          this.strategy = strategy;
    }

    //use the strategy
    public void createArchive(ArrayList<File> files) {
          strategy.compressFiles(files);
    }
}
```

Example – File Compression Tool

```
public class Client {
    public static void main(String[] args) {
          CompressionContext ctx = new CompressionContext();
          //we could assume context is already set by preferences

          ctx.setCompressionStrategy(new ZipCompressionStrategy());
          //get a list of files...
          ctx.createArchive(fileList);
    }
}
```

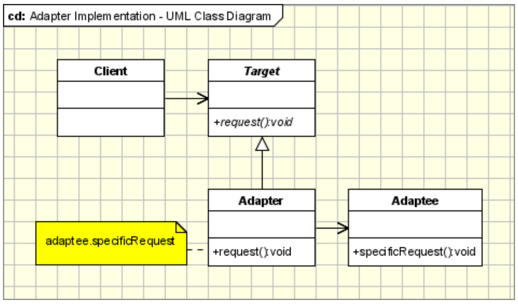
Structural Design Patterns

Adapter

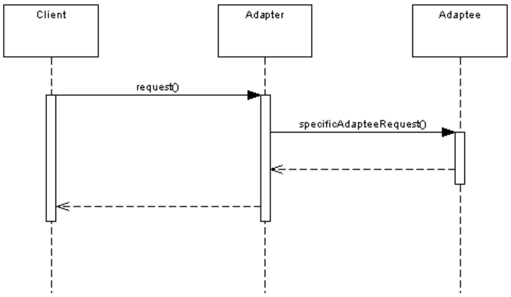
Motivation

- Convert the interface of a class into another interface clients expect.
- Adapter lets classes work together, that could not otherwise because of incompatible interfaces

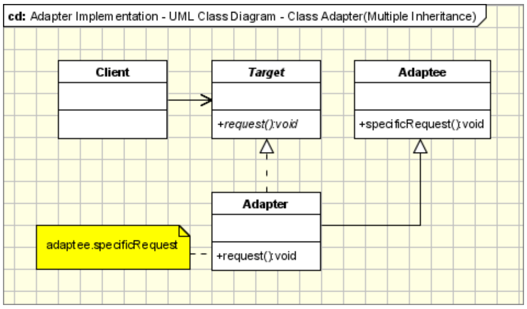
Implementation – Object Adapter



Interactions



Implementation – Class Adapter



Example

```
public class CelciusReporter {

    double temperatureInC;

    public CelciusReporter() {
    }

    public double getTemperature() {
          return temperatureInC;
    }

    public void setTemperature(double temperatureInC) {
          this.temperatureInC = temperatureInC;
    }

}
```

Example

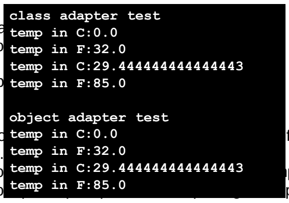
```
public interface TemperatureInfo {

    public double getTemperatureInF();

    public void setTemperatureInF(double temperatureInF);

    public double getTemperatureInC();

    public void setTemperatureInC(double temperatureInC);

}
```

```
// example of a class adapter
public class TemperatureClassReporter extends CelciusReporter
    implements TemperatureInfo {
    @Override
    public double getTemperatureInC() {
          return temperatureInC;
    }
    @Override
    public double getTemperatureInF() {
          return cToF(temperatureInC);
    }
    @Override
    public void setTemperatureInC(double temperatureInC) {
          this.temperatureInC = temperatureInC;
    }
    @Override
    public void setTemperatureInF(double temperatureInF) {
          this.temperatureInC = fToC(temperatureInF);
    }

    private double fToC(double f) {
          return ((f - 32) * 5 / 9);
    }
    private double cToF(double c) {
          return ((c * 9 / 5) + 32);
    }
}
```

```
// example of an object adapter
public class TemperatureObjectReporter implements TemperatureInfo {
    CelciusReporter celciusReporter;
    public TemperatureObjectReporter() {
          celciusReporter = new CelciusReporter();
    }
    @Override
    public double getTemperatureInC() {
          return celciusReporter.getTemperature();
    }
    @Override
    public double getTemperatureInF() {
          return cToF(celciusReporter.getTemperature());
    }
    @Override
    public void setTemperatureInC(double temperatureInC) {
          celciusReporter.setTemperature(temperatureInC);
    }
    @Override
    public void setTemperatureInF(double temperatureInF) {
          celciusReporter.setTemperature(fToC(temperatureInF));
    }
    private double fToC(double f) {
          return ((f - 32) * 5 / 9);
    }
    private double cToF(double c) {
          return ((c * 9 / 5) + 32);
    }
}
```

```
public class AdapterDemo {
    public static void main(String[] args) {
        // class adapter
        System.out.println("class adapter test");
        TemperatureInfo tempInfo = new TemperatureClassReporter();
        testTempInfo(tempInfo);

        // object adapter
        System.out.println("\nobject adapter test");
        tempInfo = new TemperatureObjectReporter();
        testTempInfo(tempInfo);
    }

    public static void testTempInfo(TemperatureInfo tempInfo) {
        tempInfo.setTemperatureInC(0);
        System.out.println("temp in C:" + tempInfo.getTemperatureInC());
        System.out.println("temp in F:" + tempInfo.getTemperatureInF());

        tempInfo.setTemperatureInF(85);
        System.out.println("temp in C:" + tempInfo.getTemperatureInC());
        System.out.println("temp in F:" + tempInfo.getTemperatureInF());
    }
}
```

```
public class AdapterDemo {
    public static void main(String[] args) {
        // class adapter
        System.out.println("class adapter test");
        TemperatureInfo tempInfo = new TemperatureClassReporter();
        testTempInfo(tempInfo);

        // object a
        System.o
        tempInfo
        testTemp
    }

    public static vo                              fo) {
        tempInfo.
        System.o                                  peratureInC());
        System.o                                  peratureInF());

        tempInfo.setTemperatureInF(85);
        System.out.println("temp in C:" + tempInfo.getTemperatureInC());
        System.out.println("temp in F:" + tempInfo.getTemperatureInF());
    }
}
```



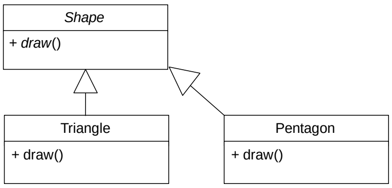
```
class adapter test
temp in C:0.0
temp in F:32.0
temp in C:29.444444444444443
temp in F:85.0

object adapter test
temp in C:0.0
temp in F:32.0
temp in C:29.444444444444443
temp in F:85.0
```

Bridge

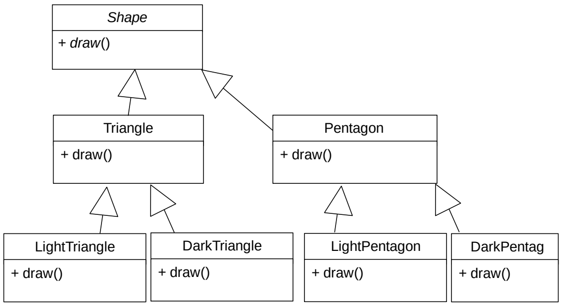
Motivation

- We need to decouple an abstraction from its implementation so that the two can vary independently.

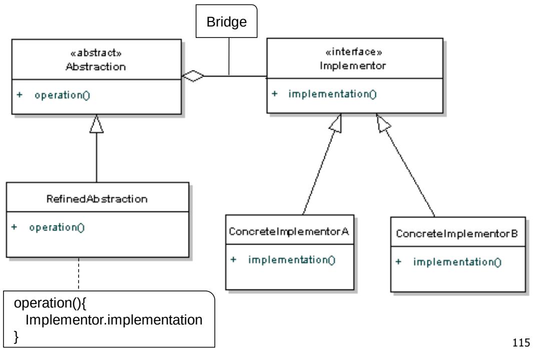
Motivation



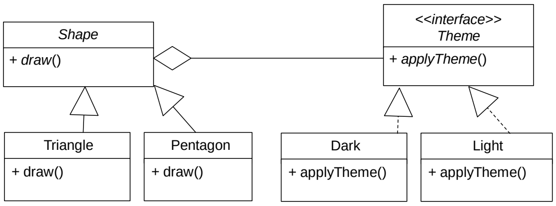
Motivation



Implementation



Example



Example

```
public interface Theme {
    public void applyTheme();
}
```

Example

```
public abstract class Shape {
    //Composition - implementor
    protected Theme theme;

    //constructor with implementor as input argument
    public Shape(Theme t){
        this.theme=t;
    }

    public setTheme(Theme t){
        this.theme=t;
    }

    abstract public void draw ();
}
```

Example

```
public class Triangle extends Shape{

    public Triangle(Theme t) {
        super(t);
    }

    @Override
    public void draw() {
        System.out.print("Triangle drawn with theme ");
        theme.applyTheme();
    }
}
```

Example

```
public class Pentagon extends Shape{

    public Pentagon(Theme t) {
        super(t);
    }

    @Override
    public void draw() {
        System.out.print("Pentagon drawn with theme ");
        theme.applyTheme();
    }
}
```

Example

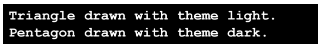
```
public class LightTheme implements Theme{

   public void applyTheme(){
        System.out.println("light.");
   }
}
```

Example

```
public class DarkTheme implements Theme{

   public void applyTheme(){
        System.out.println("dark.");
   }
}
```

Example

```
public class BridgePatternTest {

   public static void main(String[] args) {
        Shape tri = new Triangle(new LightTheme());
        tri.draw();

        Shape pent = new Pentagon(new DarkTheme());
        pent.draw ();
   }

}
```



```
Triangle drawn with theme light.
Pentagon drawn with theme dark.
```