Software Architecture 11
Design Pattern 3

Haopeng Chen

***RE**liable, **IN**telligent and **S**calable Systems Group (**REINS**)*

Shanghai Jiao Tong University

Shanghai, China

http://reins.se.sjtu.edu.cn/~chenhp

e-mail: chen-hp@sjtu.edu.cn

Agenda

- GoF Design Patterns
  - Behavioral Patterns

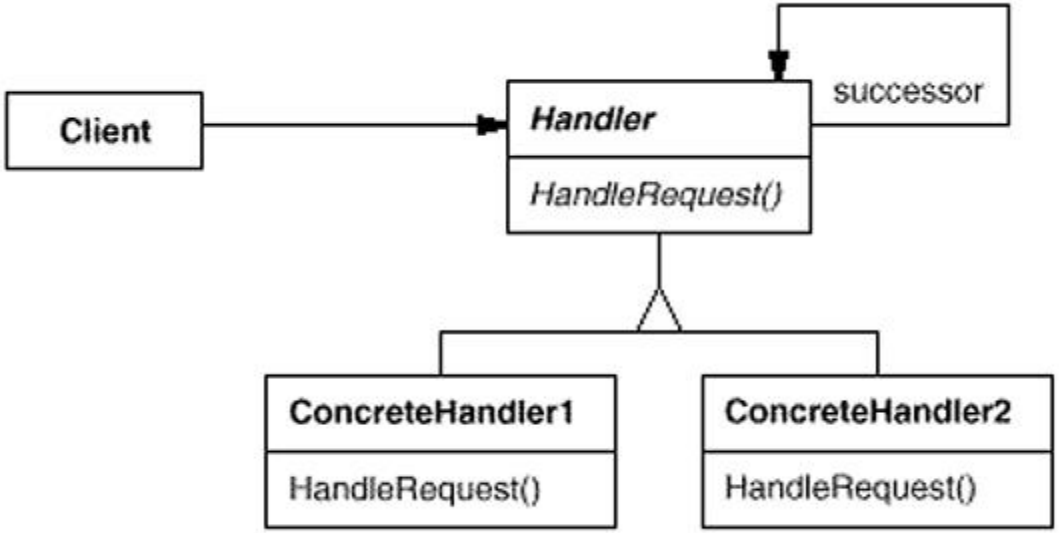
Chain of Responsibility

- Intent
  - Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

- In Proxy pattern
  - We mentioned that a container is composed of multiple interceptors each of which is responsible for one aspect of instance management.
  - All the interceptors are interconnected as a chain of responsibility

Chain of Responsibility

- Structure



Chain of Responsibility

- Applicability

- Use Chain of Responsibility when
  - more than one object may handle a request, and the handler isn't known a priori. The handler should be ascertained automatically.
  - you want to issue a request to one of several objects without specifying the receiver explicitly.
  - the set of objects that can handle a request should be specified dynamically.

Chain of Responsibility

- Consequences

- Chain of Responsibility has the following benefits and liabilities:
  - Reduced coupling.
  - Added flexibility in assigning responsibilities to objects.
  - Receipt isn't guaranteed.

Chain of Responsibility

- Implementation

- Here are implementation issues to consider in Chain of Responsibility:
  - Implementing the successor chain.
  - Connecting successors.
  - Representing requests.
  - Automatic forwarding in Smalltalk.

Chain of Responsibility

TopTitle.java

```
    // the Interface to be implemented by all Classes in the Chain


public interface TopTitle {
   public String getTopTitle();
   public String getAllCategories();
}
```

Chain of Responsibility

DvdCategory.java // the Highest Class in the Chain

```
public class DvdCategory implements TopTitle {
    private String category;
    private String topCategoryTitle;
    public DvdCategory(String category)
      { this.setCategory(category); }
    public void setCategory(String categoryIn)
      {this.category = categoryIn;}
    public String getCategory()
      {return this.category;}
    public String getAllCategories()
      {return getCategory();}
    public void setTopCategoryTitle(String topCategoryTitleIn) {this.topCategoryTitle =
    topCategoryTitleIn;}
    public String getTopCategoryTitle()
      {return this.topCategoryTitle;}
    public String getTopTitle()
      {return this.topCategoryTitle;}
}
```

Chain of Responsibility


```
DvdSubCategory.java // the Middle Class in the Chain
public class DvdSubCategoryimplements TopTitle{
    private String subCategory;
    private String topSubCategoryTitle;
    private DvdCategory parent;
    public DvdSubCategory(DvdCategorydvdCategory, String subCategory)
     {
        this.setSubCategory(subCategory);
        this.parent= dvdCategory;
     }
    public void setSubCategory(String subCategoryIn)
     {this.subCategory= subCategoryIn;}
    public String getSubCategory()
     {return this.subCategory;}
    public void setCategory(String categoryIn)
     {parent.setCategory(categoryIn);}
    public String getCategory()
     {return parent.getCategory();}
    public String getAllCategories()
     {return (getCategory() + "/" + getSubCategory());}
```

Chain of Responsibility


```
public void setTopSubCategoryTitle(String  topSubCategoryTitleIn)
    {this.topSubCategoryTitle  = topSubCategoryTitleIn;}
public String getTopSubCategoryTitle()
    {return this.topSubCategoryTitle;}
public void setTopCategoryTitle(String  topCategoryTitleIn)
    {parent.setTopCategoryTitle(topCategoryTitleIn);}
public String getTopCategoryTitle()
    {return parent.getTopCategoryTitle();}
public String getTopTitle()
  {
    if (null != getTopSubCategoryTitle())
      { return this.getTopSubCategoryTitle();  }
    else
      { System.out.println("no top title in Category/SubCategory  " +
        getAllCategories());
        return parent.getTopTitle(); }
   }
}
```

Chain of Responsibility


```
DvdSubSubCategory.java // the Lowest Class in the Chain
public class DvdSubSubCategoryimplements  TopTitle{
  private String subSubCategory;
  private String topSubSubCategoryTitle;
  private DvdSubCategory  parent;
  public DvdSubSubCategory(DvdSubCategorydvdSubCategory,  String subCategory)
   {
    this.setSubSubCategory(subCategory);
    this.parent=  dvdSubCategory;
   }
  public void setSubSubCategory(String  subSubCategoryIn)
    {this.subSubCategory=  subSubCategoryIn;}
  public String getSubSubCategory()
    {return this.subSubCategory;}
  public void setSubCategory(String subCategoryIn)
    {parent.setSubCategory(subCategoryIn);}
  public String getSubCategory()
    {return parent.getSubCategory();}
  public void setCategory(String categoryIn)
    {parent.setCategory(categoryIn);}
```

Chain of Responsibility


```
public String getCategory()
   {return parent.getCategory();}
public String getAllCategories()
   {return (getCategory() + "/" + getSubCategory() + "/" +
     getSubSubCategory());}
public void setTopSubSubCategoryTitle(String topSubSubCategoryTitleIn)
   {this.topSubSubCategoryTitle = topSubSubCategoryTitleIn;}
public String getTopSubSubCategoryTitle()
   {return this.topSubSubCategoryTitle;}
public void setTopSubCategoryTitle(String topSubCategoryTitleIn)
   {parent.setTopSubCategoryTitle(topSubCategoryTitleIn);}
public String getTopSubCategoryTitle()
   {return parent.getTopSubCategoryTitle();}
public void setTopCategoryTitle(String topCategoryTitleIn)
   {parent.setTopCategoryTitle(topCategoryTitleIn);}
public String getTopCategoryTitle()
   {return parent.getTopCategoryTitle();}
```

Chain of Responsibility


```
    public String getTopTitle()
    {
       if (null != getTopSubSubCategoryTitle())
        { return this.getTopSubSubCategoryTitle(); }
       else
        { System.out.println("no top title in
        Category/SubCategory/SubSubCategory " + getAllCategories());
         return parent.getTopTitle();
        }
    }
}
```

Chain of Responsibility

TestChainOfResponsibility.java: // testing the Chain of Responsibility

```
class TestChainOfResponsibility {
   public static void main(String[] args) {
     String topTitle;
     DvdCategory comedy = new DvdCategory("Comedy");
     comedy.setTopCategoryTitle("Ghost World");
     DvdSubCategory comedyChildrens =
         new DvdSubCategory(comedy, "Childrens");
     DvdSubSubCategory comedyChildrensAquatic =
         new DvdSubSubCategory(comedyChildrens, "Aquatic");
     comedyChildrensAquatic.setTopSubSubCategoryTitle("Sponge  Bob Squarepants");
     System.out.println("");
     System.out.println("Getting top comedy title:");
     topTitle = comedy.getTopTitle();
     System.out.println("The top title for " + comedy.getAllCategories() + " is " + topTitle);
     System.out.println("");
```

```
        System.out.println("Getting top comedy/childrens title:");
        topTitle = comedyChildrens.getTopTitle();
        System.out.println("The top title for " +
              comedyChildrens.getAllCategories() + " is " + topTitle);
        System.out.println("");
        System.out.println("Getting top comedy/childrens/aquatic title:");
        topTitle = comedyChildrensAquatic.getTopTitle();
        System.out.println("The top title for " +
        comedyChildrensAquatic.getAllCategories() + " is " + topTitle);
        }
}
```

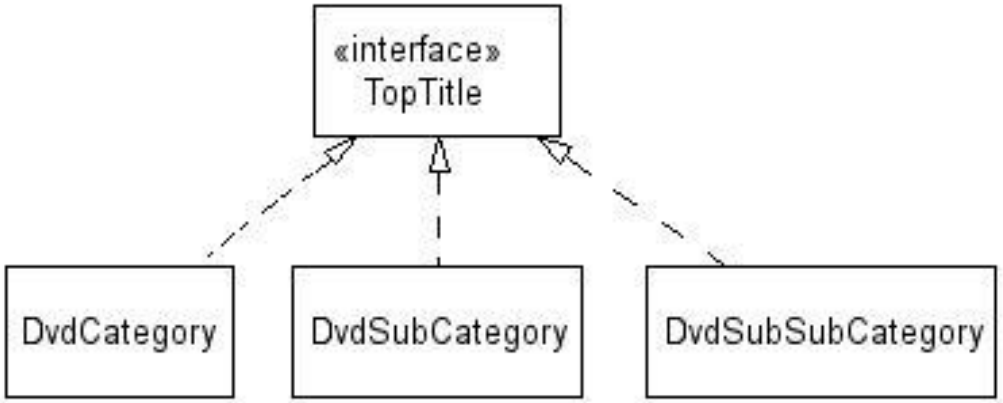
Test Results

Getting top comedy title:
The top title for Comedy is Ghost World

Getting top comedy/childrens title:
no top title in Category/SubCategory Comedy/Childrens
The top title for Comedy/Childrens is Ghost World

Getting top comedy/childrens/aquatic title:
The top title for Comedy/Childrens/Aquatic is Sponge Bob Squarepants

Chain of Responsibility

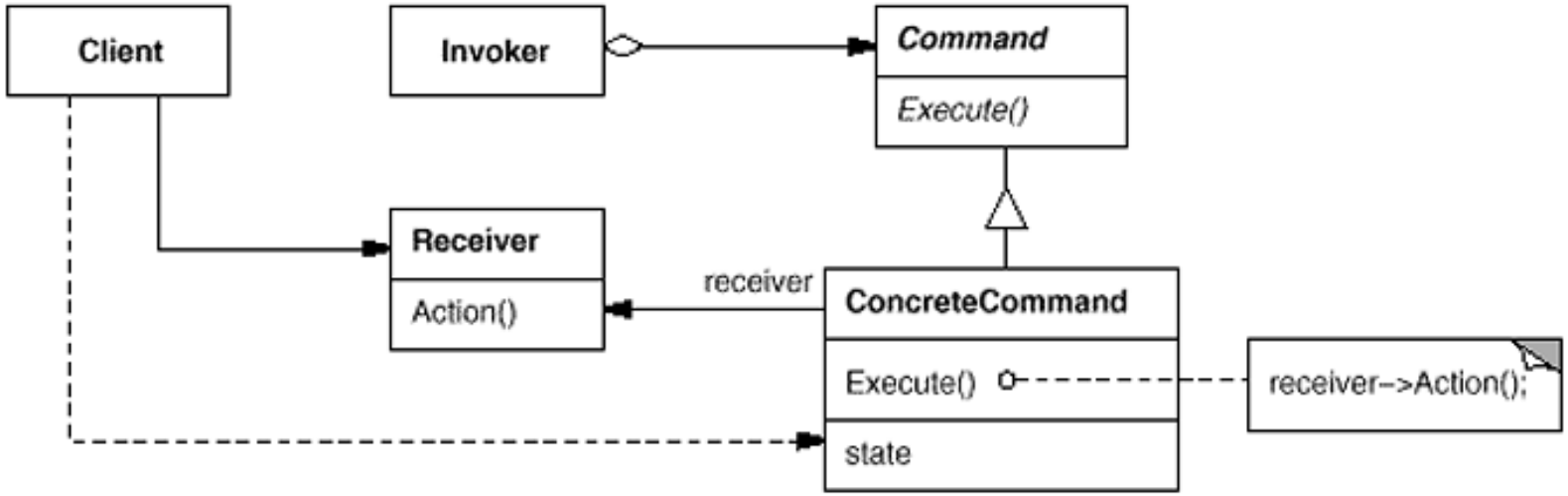


Command

- Intent
  - Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- In 4S application
  - The developers want to encapsulate the code of processing of purchasing in order to decouple it with the one invokes it.
  - Thus, the maintainability of application will be improved.
  - Command pattern can be applied into this scenario.

Command

- Structure



Command

- Applicability

- Use the Command pattern when you want
  - parameterize objects by an action to perform.
  - specify, queue, and execute requests at different times.
  - support undo.
  - support logging changes so that they can be reapplied in case of a system crash.
  - structure a system around high-level operations built on primitives operations.

Command

- Consequences

- The Command pattern has the following consequences:
  - Command decouples the object that invokes the operation from the one that knows how to perform it.
  - Commands are first-class objects. They can be manipulated and extended like any other object.
  - You can assemble commands into a composite command. An example is the MacroCommand class described earlier. In general, composite commands are an instance of the Composite pattern.
  - It's easy to add new Commands, because you don't have to change existing classes.

Command

- Implementation

- Consider the following issues when implementing the Command pattern:
  - How intelligent should a command be?
  - Supporting undo and redo.
  - Avoiding error accumulation in the undo process.
  - Using C++ templates.

Command

DvdName.java

```
    // the Receiver
public class DvdName {

  private String titleName;
  public DvdName(String titleName) { this.setTitleName(titleName); }

  public final void setTitleName(String titleNameIn)
    {this.titleName = titleNameIn;}
  public final String getTitleName() {return this.titleName;}
  public void setNameStarsOn() {this.setTitleName(this.getTitleName().replace(' ','*'));}
  public void setNameStarsOff() {this.setTitleName(this.getTitleName().replace('*',' '));}
  public String toString()
    {return ("DVD: " + this.getTitleName());}
}
```

Command


```
CommandAbstract.java: //the Command
public abstract class CommandAbstract { public abstract void execute(); }
DvdCommandNameStarsOn.java: //one of two Concrete Commands
public class DvdCommandNameStarsOn extends CommandAbstract {
    private DvdName dvdName;
    public DvdCommandNameStarsOn(DvdName dvdNameIn)
      { this.dvdName = dvdNameIn; }
    public void execute() {this.dvdName.setNameStarsOn();}
}
DvdCommandNameStarsOff.java: //two of two Concrete Commands
public class DvdCommandNameStarsOff extends CommandAbstract {
    private DvdName dvdName;
    public DvdCommandNameStarsOff(DvdName dvdNameIn)
       { this.dvdName = dvdNameIn; }
    public void execute() {this.dvdName.setNameStarsOff();}
}
```

Command

TestCommand.java: //testing the Command

```
class TestCommand {
  public static void main(String[] args) {
    DvdName jayAndBob =
        new DvdName("Jay and Silent Bob Strike Back");
    DvdName spongeBob =
        new DvdName("Sponge Bob Squarepants -Nautical Nonsense and
                          Sponge Buddies");
    System.out.println("as first instantiated");
    System.out.println(jayAndBob.toString());
    System.out.println(spongeBob.toString());

    CommandAbstract bobStarsOn = new
          DvdCommandNameStarsOn(jayAndBob);
    CommandAbstract bobStarsOff = new
          DvdCommandNameStarsOff(jayAndBob);
```

Command


```
    CommandAbstract spongeStarsOn = new
        DvdCommandNameStarsOn(spongeBob);
    CommandAbstract spongeStarsOff = new
        DvdCommandNameStarsOff(spongeBob);
    bobStarsOn.execute();
    spongeStarsOn.execute();
    System.out.println(" ");
    System.out.println("stars on");
    System.out.println(jayAndBob.toString());
    System.out.println(spongeBob.toString());

    spongeStarsOff.execute();
    System.out.println(" ");
    System.out.println("sponge stars off");
    System.out.println(jayAndBob.toString());
    System.out.println(spongeBob.toString());
    }
}
```

Command

Test Results

as first instantiated

DVD: Jay and Silent Bob Strike Back

DVD: Sponge Bob Squarepants -Nautical Nonsense and Sponge Buddies

stars on

DVD: Jay*and*Silent*Bob*Strike*Back

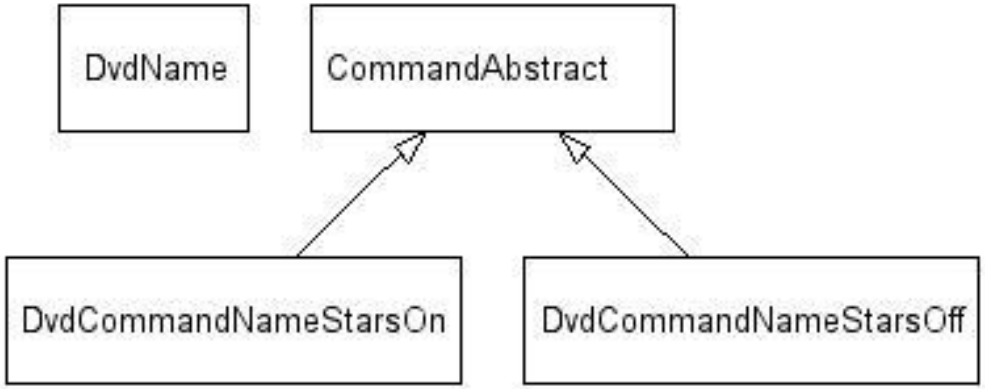
DVD: Sponge*Bob*Squarepants*-*Nautical*Nonsense*and*Sponge*Buddies sponge

stars off

DVD: Jay*and*Silent*Bob*Strike*Back

DVD: Sponge Bob Squarepants -Nautical Nonsense and Sponge Buddies

Command

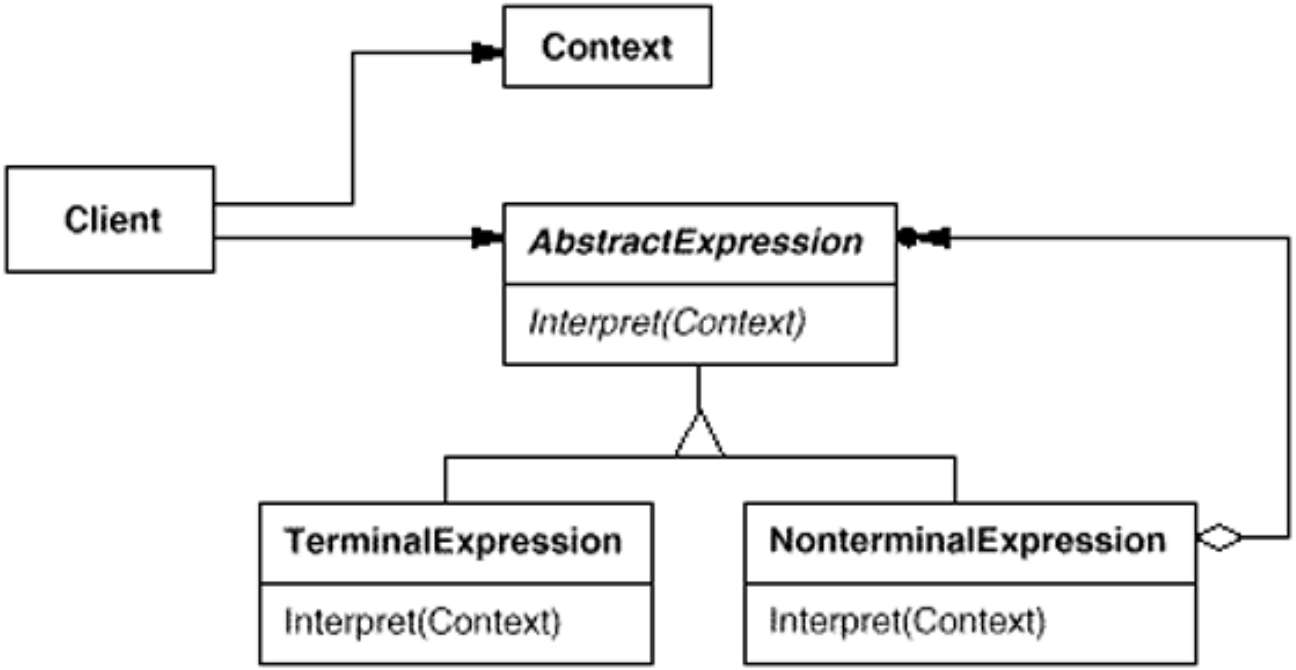

DvdName
CommandAbstract
DvdCommandNameStarsOn
DvdCommandNameStarsOff

Interpreter

- Intent
  - Given a language, define a represention for its grammar along with an interpreter that uses the representation to interpret sentences in the language
- In 4S application
  - The users are allowed to customize the query.
  - The syntax of the query language is simple:
    - ID:[number] R:[string] F:string L:[string]
    - ID is user's id
    - R is region
    - F is first name
    - L is last name
  - We need a interpreter to interpret the query.

Interpreter

- Structure



Interpreter

- Applicability

- Use the Interpreter pattern when there is a language to interpret, and you can represent statements in the language as abstract syntax trees.
- The Interpreter pattern works best when
  - the grammar is simple.
  - efficiency is not a critical concern.

Interpreter

- Consequences

- The Interpreter pattern has the following benefits and liabilities:
  - It's easy to change and extend the grammar.
  - Implementing the grammar is easy, too.
  - Complex grammars are hard to maintain.
  - Adding new ways to interpret expressions.

Interpreter

- Implementation

- The Interpreter and Composite patterns share many implementation issues. The following issues are specific to Interpreter:
  - Creating the abstract syntax tree.
  - Defining the Interpret operation.
  - Sharing terminal symbols with the Flyweight pattern.

Interpreter

DvdInterpreterClient.java: //the Client

```
import java.util.StringTokenizer;
public class DvdInterpreterClient {
 DvdInterpreterContext dvdInterpreterContext;
 public DvdInterpreterClient(DvdInterpreterContext dvdInterpreterContext)
  { this.dvdInterpreterContext = dvdInterpreterContext; }
  // expression syntax:
  // show title | actor [for actor | title ]
 public String interpret(String expression) {
    StringBuffer result = new StringBuffer("Query Result: ");
    String currentToken;
    StringTokenizer expressionTokens = new StringTokenizer(expression);
    char mainQuery = ' ';
    char subQuery = ' ';
    boolean forUsed = false;
    String searchString = null;
    boolean searchStarted = false;
    boolean searchEnded = false;
```

Interpreter


```
while (expressionTokens.hasMoreTokens())
    {
        currentToken = expressionTokens.nextToken();
        if (currentToken.equals("show"))
            {continue;} //show in all queries, not really used
        else if (currentToken.equals("title"))
                {
                 if (mainQuery == ' ') {mainQuery = 'T';}
                 else {
                        if ((subQuery == ' ') && (forUsed)) {subQuery = 'T';}
                        }
                }
                else
                 if (currentToken.equals("actor"))  {
                        if (mainQuery == ' ') {mainQuery = 'A';}
                        else {
                                if ((subQuery == ' ') && (forUsed)) {subQuery = 'A';}
                                }
                 }
                  else
```

Interpreter


```
if (currentToken.equals("for"))  {forUsed = true;}
   else
      if ((searchString == null) && (subQuery != ' ') &&
         (currentToken.startsWith("<")))
      {
            searchString = currentToken;
            searchStarted = true;
             if (currentToken.endsWith(">"))  {searchEnded = true;}
       }
       else
          if ((searchStarted)  && (!searchEnded))
             { searchString = searchString + " " + currentToken;
               if (currentToken.endsWith(">"))  {searchEnded = true;}
              }
 }
 if (searchString != null)
 {
   searchString = searchString.substring(1,(searchString.length()  -1));
   //remove <>
 }
```

Interpreter


```
DvdAbstractExpression abstractExpression;
switch (mainQuery) {
   case 'A' : {
       switch (subQuery) {
           case 'T' : {
               abstractExpression  = new DvdActorTitleExpression(searchString);
               break;
           }
           default : {
              abstractExpression  = new DvdActorExpression();
               break;
           }
       }
       break;
 }
 case 'T' : {
     switch (subQuery) {
         case 'A' : {
            abstractExpression  = new DvdTitleActorExpression(searchString);
            break;
         }
```

Interpreter

```
            default : {
                        abstractExpression  = new DvdTitleExpression();
                        break;
                        }
             }
             break;
          }
        default :
          return result.toString();
   }
  result.append(abstractExpression.interpret(dvdInterpreterContext));
  return result.toString();
  }
}
```

Interpreter

DvdInterpreterContext.java: // The Context

```
import java.util.ArrayList;
import java.util.ListIterator;
public class DvdInterpreterContext {
  private ArrayList titles = new ArrayList();
  private ArrayList actors = new ArrayList();
  private ArrayList titlesAndActors = new ArrayList();
  public void addTitle(String title) {titles.add(title);}
  public void addActor(String actor) {actors.add(actor);}
  public void addTitleAndActor(TitleAndActor titleAndActor)
    {titlesAndActors.add(titleAndActor);}
  public ArrayList getAllTitles() {return titles;}
  public ArrayList getAllActors() {return actors;}
  public ArrayList getActorsForTitle(String titleIn)
  {
        ArrayList actorsForTitle = new ArrayList();
        TitleAndActor tempTitleAndActor;
        ListIterator titlesAndActorsIterator = titlesAndActors.listIterator();
        while (titlesAndActorsIterator.hasNext()) {
          tempTitleAndActor = (TitleAndActor)titlesAndActorsIterator.next();
```

```
    if (titleIn.equals(tempTitleAndActor.getTitle()))
      { actorsForTitle.add(tempTitleAndActor.getActor()); }
    }
    return actorsForTitle;
  }
  public ArrayList getTitlesForActor(String actorIn) {
     ArrayList titlesForActor = new ArrayList();
     TitleAndActor tempTitleAndActor;
     ListIterator actorsAndTitlesIterator = titlesAndActors.listIterator();
     while (actorsAndTitlesIterator.hasNext()) {
       tempTitleAndActor = (TitleAndActor)actorsAndTitlesIterator.next();
       if (actorIn.equals(tempTitleAndActor.getActor()))
         { titlesForActor.add(tempTitleAndActor.getTitle()); }
     }
     return titlesForActor;
  }
}
```

Interpreter


```
DvdAbstractExpression.java: //The Abstract Expression
public abstract class DvdAbstractExpression {
  public abstract String interpret(DvdInterpreterContext  dvdInterpreterContext);
}
DvdActorExpression.java: //One Of Four Terminal Expressions
import java.util.ArrayList;
import java.util.ListIterator;
public class DvdActorExpression extends DvdAbstractExpression {
  public String interpret(DvdInterpreterContext  dvdInterpreterContext)
  { ArrayList actors = dvdInterpreterContext.getAllActors();
    ListIterator actorsIterator = actors.listIterator();
    StringBuffer titleBuffer = new StringBuffer("");
    boolean first = true;
    while (actorsIterator.hasNext()) {
        if (!first) {titleBuffer.append(", ");}
        else {first = false;}
        titleBuffer.append((String)actorsIterator.next());
    }
    return titleBuffer.toString();
  }
}
```

Interpreter

DvdActorTitleExpression.java: //Two Of Four Terminal Expressions

```
import java.util.ArrayList;
import java.util.ListIterator;
public class DvdActorTitleExpression extends DvdAbstractExpression {
   String title;
   public DvdActorTitleExpression(String title)
     {this.title = title;}
   public String interpret(DvdInterpreterContext dvdInterpreterContext)
     {
       ArrayList actorsAndTitles = dvdInterpreterContext.getActorsForTitle(title);
       ListIterator actorsAndTitlesIterator = actorsAndTitles.listIterator();
       StringBuffer actorBuffer = new StringBuffer("");
       boolean first = true;
       while (actorsAndTitlesIterator.hasNext()) {
            if (!first) {actorBuffer.append(", ");}
           else {first = false;}
          actorBuffer.append((String)actorsAndTitlesIterator.next());
        }
        return actorBuffer.toString();
     }
}
```

Interpreter

DvdTitleExpression.java: //Three Of Four Terminal Expressions

```
import java.util.ArrayList;
import java.util.ListIterator;
public class DvdTitleExpression extends DvdAbstractExpression {
    public String interpret(DvdInterpreterContext dvdInterpreterContext) {
        ArrayList titles = dvdInterpreterContext.getAllTitles();
        ListIterator titlesIterator = titles.listIterator();
        StringBuffer titleBuffer = new StringBuffer("");
        boolean first = true;
        while (titlesIterator.hasNext())
        {
            if (!first) {titleBuffer.append(", ");}
            else {first = false;}
            titleBuffer.append((String)titlesIterator.next());
        }
        return titleBuffer.toString();
    }
}
```

Interpreter

DvdTitleActorExpression.java: //Four Of Four Terminal Expressions

```
import java.util.ArrayList;
import java.util.ListIterator;
public class DvdTitleActorExpression extends DvdAbstractExpression {
   String title;
   public DvdTitleActorExpression(String title) {this.title = title;}
   public String interpret(DvdInterpreterContext dvdInterpreterContext)
   {
      ArrayList titlesAndActors = dvdInterpreterContext.getTitlesForActor(title);
      ListIterator titlesAndActorsIterator = titlesAndActors.listIterator();
      StringBuffer titleBuffer = new StringBuffer("");
      boolean first = true;
      while (titlesAndActorsIterator.hasNext())
      {
         if (!first) {titleBuffer.append(", ");}
         else {first = false;}
         titleBuffer.append((String)titlesAndActorsIterator.next());
      }
      return titleBuffer.toString();
   }
}
```

Interpreter

TitleAndActor.java: //A Helper Class

```
public class TitleAndActor {
  private String title;
  private String actor;
  public TitleAndActor(String title, String actor)
  {
   this.title = title;
   this.actor = actor;
  }
   public String getTitle() {return this.title;}
   public String getActor() {return this.actor;}
}
```

Interpreter

TestDvdInterpreter.java: //testing the Interpreter

```
class TestDvdInterpreter {
  public static void main(String[] args) {
    DvdInterpreterContext dvdInterpreterContext = new DvdInterpreterContext();
    dvdInterpreterContext.addTitle("Caddy Shack");
    dvdInterpreterContext.addTitle("Training Day");
    dvdInterpreterContext.addTitle("Hamlet");
    dvdInterpreterContext.addActor("Ethan Hawke");
    dvdInterpreterContext.addActor("Denzel Washington");
    dvdInterpreterContext.addTitleAndActor(new TitleAndActor("Hamlet", "Ethan Hawke"));
    dvdInterpreterContext.addTitleAndActor(new TitleAndActor("Training Day", "Ethan
     Hawke"));
    dvdInterpreterContext.addTitleAndActor(new TitleAndActor("Caddy Shack", "Ethan
     Hawke"));
    dvdInterpreterContext.addTitleAndActor(new TitleAndActor("Training Day", "Denzel
     Washington"));
    DvdInterpreterClient dvdInterpreterClient = new
          DvdInterpreterClient(dvdInterpreterContext);
```

Interpreter


```
    System.out.println("interpreting show actor: " +
            dvdInterpreterClient.interpret("show actor")); System.out.println("interpreting
     show actor for title : " +
            dvdInterpreterClient.interpret("show actor for title "));
     System.out.println("interpreting show title: " +
            dvdInterpreterClient.interpret("show title"));
     System.out.println("interpreting show title for actor : " +
            dvdInterpreterClient.interpret("show title for actor "));
   }
}
```

Test Results

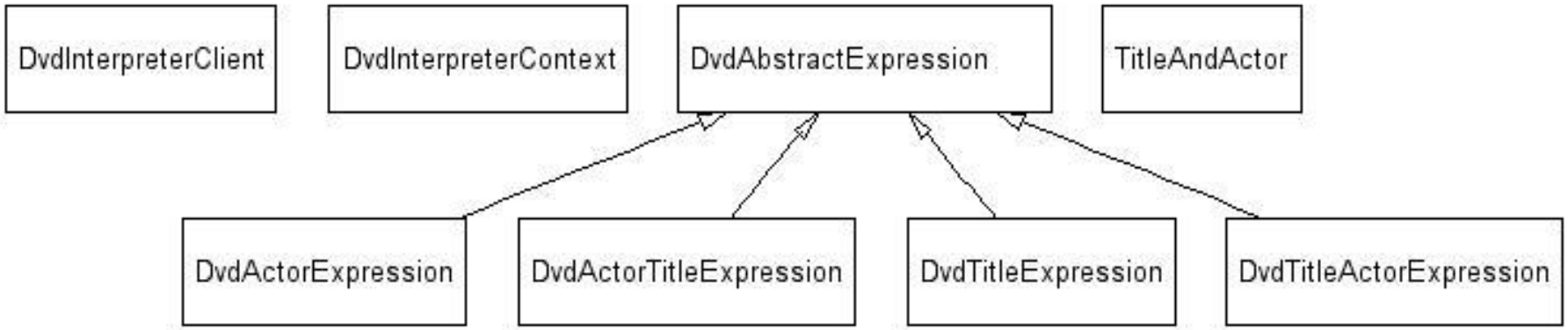
interpreting show actor:Query Result: Ethan Hawke, Denzel Washington

interpreting show actor for title : Query Result: Ethan Hawke, Denzel Washington

interpreting show title: Query Result: Caddy Shack, Training Day, Hamlet

interpreting show title for actor : Query Result: Hamlet, Training Day, Caddy Shack

Interpreter



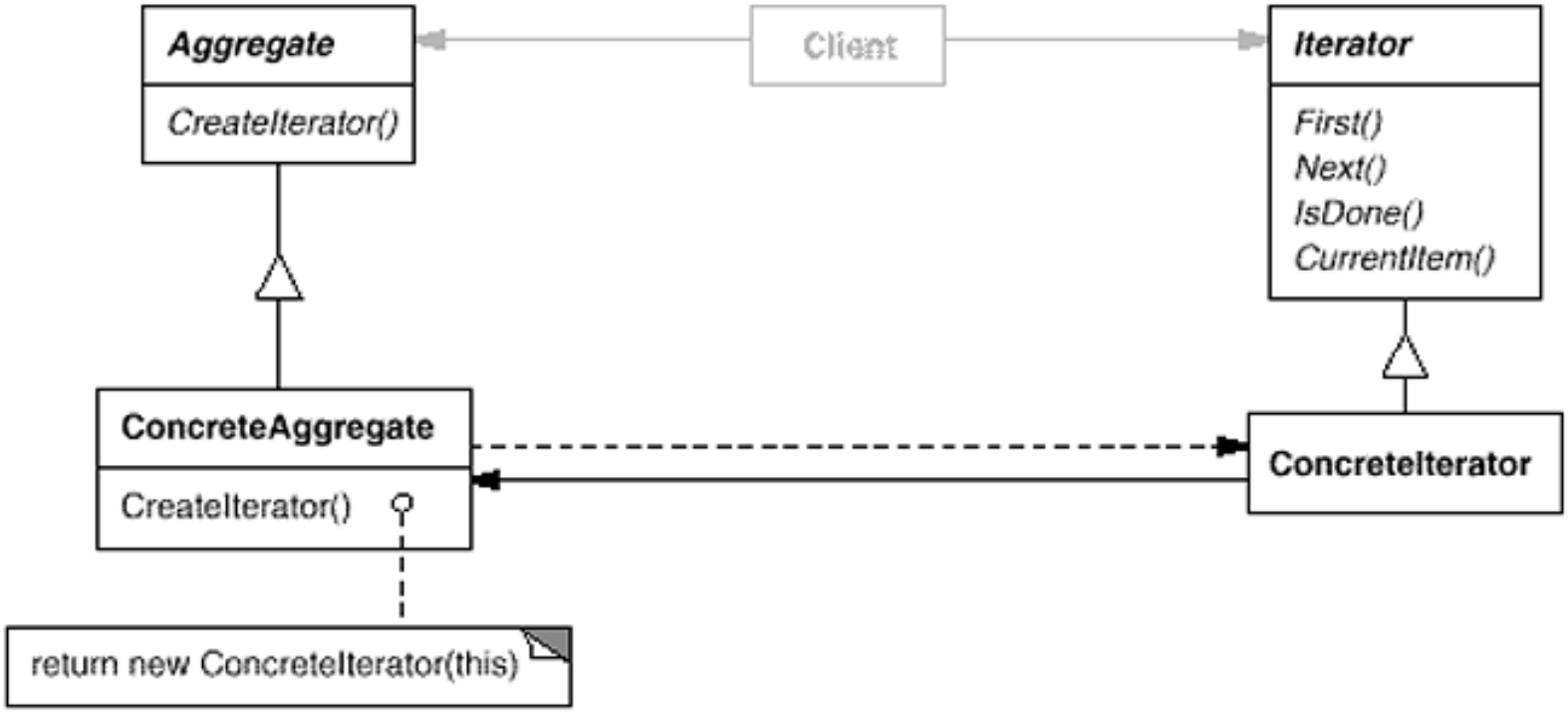
Iterator

- Intent
  - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

- In 4S application
  - Users often query the information about cars and customers.
  - Different collection types are used for such information
    - Set is for information about customers since it can remove duplicated elements.
    - HashLinkedList is for information about cars since it facilitates the query by types.
  - We need a universal way to access the information.
  - Consequently, iterator is used.

Iterator

- Structure



Iterator

- Applicability

- Use the Iterator pattern
  - to access an aggregate object's contents without exposing its internal representation.
  - to support multiple traversals of aggregate objects.
  - to provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration).

Iterator

- Consequences

- The Iterator pattern has three important consequences:
  - It supports variations in the traversal of an aggregate.
  - Iterators simplify the Aggregate interface.
  - More than one traversal can be pending on an aggregate.

Iterator

- Implementation

- Iterator has many implementation variants and alternatives.
  - Who controls the iteration?
  - Who defines the traversal algorithm?
  - How robust is the iterator?
  - Additional Iterator operations.
  - Using polymorphic iterators in C++.
  - Iterators may have privileged access.
  - Iterators for composites.
  - Null iterators.

Iterator

- DvdListIterator.java

```
// the Iterator Interface
public interface DvdListIterator
{
   public void first();
   public void next();
   public boolean isDone();
   public String currentItem();
}
```

Iterator

- DvdList.java

```
// the Concrete Aggregate (with a Concrete Iterator inner class)
public class DvdList
{
    private String[] titles;
    private int titleCount;
    private int arraySize;

    public DvdList()
    {
      titles = new String[3];
      titleCount = 0;
      arraySize = 3;
    }

    public int count() {return titleCount;}
```

Iterator


```
public void append(String titleIn)
{
  if (titleCount>= arraySize)
  {
    String[] tempArray= new String[arraySize];
    for (inti = 0; i < arraySize; i++) {tempArray[i] = titles[i];}
    titles = null;
    arraySize= arraySize+ 3;
    titles = new String[arraySize];
    for (inti = 0; i < (arraySize-3); i++) {titles[i] = tempArray[i];}
  }
  titles[titleCount++] = titleIn;
}
```

Iterator


```
public void delete(String titleIn)
{    boolean found = false;
     for (int i = 0; i < (titleCount -1); i++)
    { if (found == false)
      {
        if (titles[i].equals(titleIn))
        {
           found = true;
           titles[i] = titles[i + 1];
        }
      }
      else
      {
        if (i < (titleCount -1))
        {
            titles[i] = titles[i + 1];
        } else {titles[i] = null;}
      }
    }
    if (found == true) {--titleCount;}
}
public DvdListIterator createIterator() {return new InnerIterator();}
```

Iterator


```
private class InnerIterator implements DvdListIterator
{   private int currentPosition = 0;
    private InnerIterator() {}

    public void first() {currentPosition = 0;}
    public void next()
    {
       if (currentPosition < (titleCount))
          { ++currentPosition; }
    }
    public boolean isDone()
    {
       if (currentPosition >= (titleCount))
         {return true;}
      else {return false;}
    }
    public String currentItem()
     {return titles[currentPosition];}
   }
}
```

Iterator

- TestDvdIterator.java: // testing the Iterator

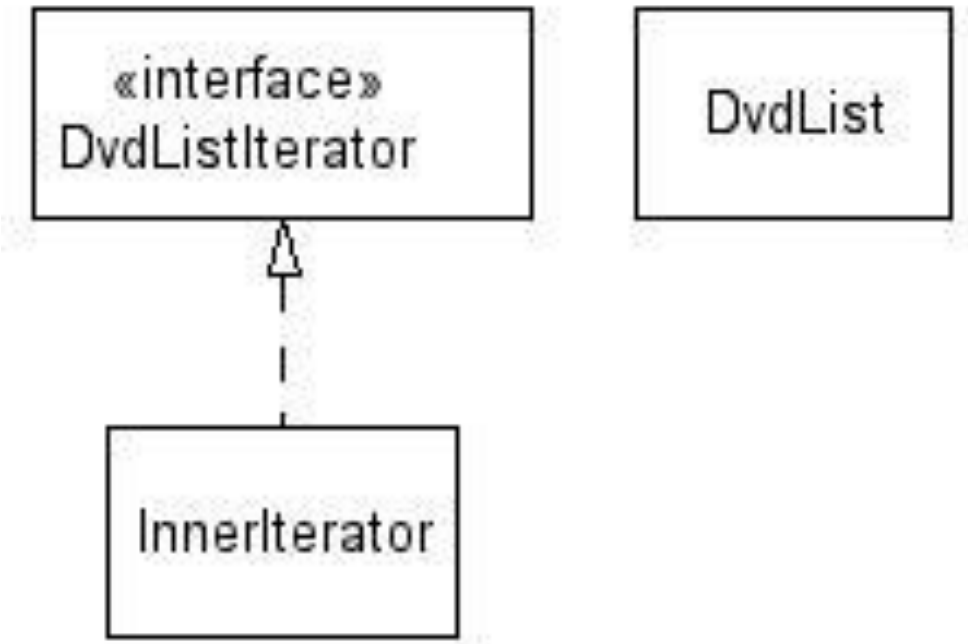
```
class TestDvdIterator {
   public static void main(String[] args)
   { DvdList fiveShakespeareMovies = new DvdList();
     fiveShakespeareMovies.append("10 Things I Hate About You");
     fiveShakespeareMovies.append("Shakespeare In Love");
     fiveShakespeareMovies.append("O (2001)");
     fiveShakespeareMovies.append("American Pie 2");
     fiveShakespeareMovies.append("Scotland, PA.");
     fiveShakespeareMovies.append("Hamlet (2000)");
     DvdListIterator fiveShakespeareIterator = fiveShakespeareMovies.createIterator();
     while (!fiveShakespeareIterator.isDone())
     { System.out.println(fiveShakespeareIterator.currentItem());
       fiveShakespeareIterator.next();
     }
     fiveShakespeareMovies.delete("American Pie 2");
     System.out.println(" ");
     fiveShakespeareIterator.first();
     while (!fiveShakespeareIterator.isDone())
      { System.out.println(fiveShakespeareIterator.currentItem());
        fiveShakespeareIterator.next();
      }
   }
}
```

Iterator

- Test Results

10 Things I Hate About You
Shakespeare In Love
O (2001)
American Pie 2
Scotland, PA.
Hamlet (2000)
10 Things I Hate About You
Shakespeare In Love
O (2001)
Scotland, PA.
Hamlet (2000)

Iterator



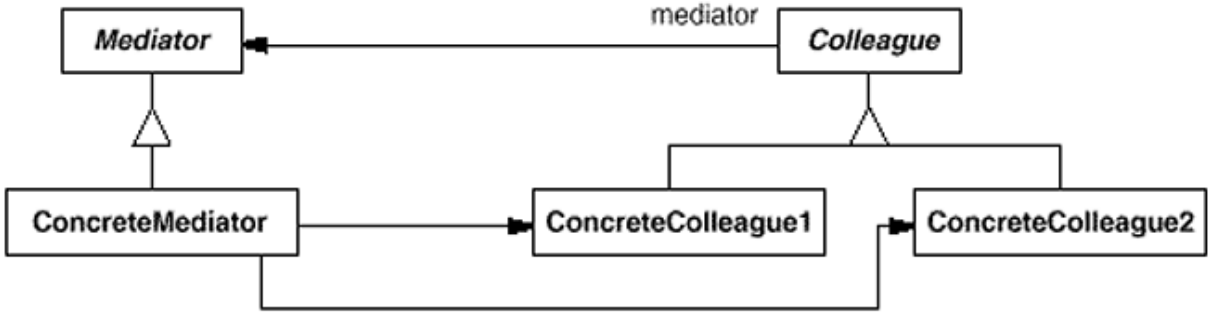
Mediator

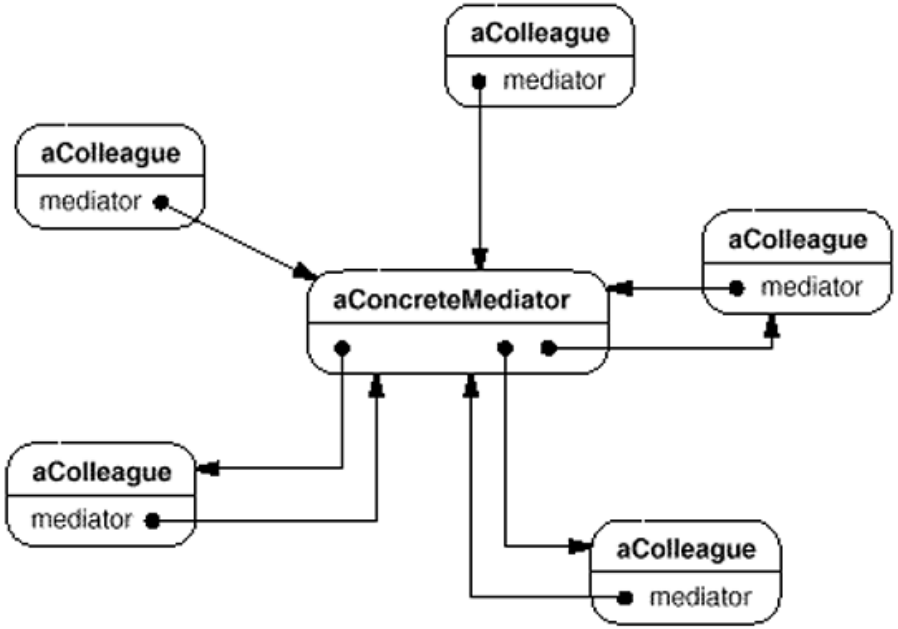
- Intent
  - Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

- In 4S application
  - In order to
    - Decouple of customers and retailers, and
    - Speed up the processing of orders.
  - The developers introduce a mediator into the application.

Mediator

- Structure





Mediator

- Applicability

- Use the Mediator pattern when
  - a set of objects communicate in well-defined but complex ways. The resulting interdependencies are unstructured and difficult to understand.
  - reusing an object is difficult because it refers to and communicates with many other objects.
  - a behavior that's distributed between several classes should be customizable without a lot of subclassing.

Mediator

- Consequences

- The Mediator pattern has the following benefits and drawbacks:
  - It limits subclassing.
  - It decouples colleagues.
  - It simplifies object protocols.
  - It abstracts how objects cooperate.
  - It centralizes control.

Mediator

- Implementation

- The following implementation issues are relevant to the Mediator pattern:
  - Omitting the abstract Mediator class.
  - Colleague-Mediator communication.

Mediator

- DvdList.java

```
// the Abstract Colleague or Mediatee
public abstract class DvdTitle
{
    private String title;
    public void setTitle(String titleIn)
        {this.title = titleIn;}
    public String getTitle()
        {return this.title;}
}
```

Mediator

- DvdLowercaseTitle.java

```
// One of Two Concrete Colleagues or Mediatees
public class DvdLowercaseTitle extends DvdTitle
{
   private String star;
   private String LowercaseTitle;
   private DvdMediator dvdMediator;

   public DvdLowercaseTitle(String title, DvdMediator dvdMediator)
   {
      this.setTitle(title);
      resetTitle();
      this.dvdMediator = dvdMediator;
      this.dvdMediator.setLowercase(this);
   }
   public DvdLowercaseTitle(DvdTitle dvdTitle, DvdMediator dvdMediator)
   {
      this(dvdTitle.getTitle(), dvdMediator);
   }
```

Mediator


```
public void resetTitle()
{
    this.setLowercaseTitle(this.getTitle().toLowerCase());
}
public void resetTitle(String title)
{
    this.setTitle(title);
    this.resetTitle();
}
public void setSuperTitleLowercase()
{
    this.setTitle(this.getLowercaseTitle());
    dvdMediator.changeTitle(this);
}
public String getLowercaseTitle()
  {return LowercaseTitle;}
private void setLowercaseTitle(String LowercaseTitle)
  { this.LowercaseTitle = LowercaseTitle; }
}
```

Mediator

- DvdUpcaseTitle.java

```
// Two of Two Concrete Colleagues or Mediatees
public class DvdUpcaseTitle extends DvdTitle
{
    private String star;
    private String upcaseTitle;
    private DvdMediator dvdMediator;

    public DvdUpcaseTitle(String title, DvdMediator dvdMediator)
    {
        this.setTitle(title);
        resetTitle();
        this.dvdMediator = dvdMediator;
        this.dvdMediator.setUpcase(this);
    }
   public DvdUpcaseTitle(DvdTitle dvdTitle, DvdMediator dvdMediator)
   {
      this(dvdTitle.getTitle(), dvdMediator);
   }
```

Mediator


```
public void resetTitle()
{
    this.setUpcaseTitle(this.getTitle().toUpperCase());
}
public void resetTitle(String title)
{
    this.setTitle(title);
    this.resetTitle();
}
public void setSuperTitleUpcase()
{
    this.setTitle(this.getUpcaseTitle());
    dvdMediator.changeTitle(this);
}
public String getUpcaseTitle()
    {return upcaseTitle;}
private void setUpcaseTitle(String  upcaseTitle)
    { this.upcaseTitle  = upcaseTitle; }
}
```

Mediator

- DvdMediator.java

```
// The Mediator
public class DvdMediator
{
    private String star;
    private DvdUpcaseTitle dvdUpcaseTitle;
    private DvdLowercaseTitle dvdLowercaseTitle;

    public void setUpcase(DvdUpcaseTitle dvdUpcaseTitle)
      { this.dvdUpcaseTitle = dvdUpcaseTitle; }
    public void setLowercase(DvdLowercaseTitle dvdLowercaseTitle)
      { this.dvdLowercaseTitle = dvdLowercaseTitle; }

    public void changeTitle(DvdUpcaseTitle dvdUpcaseTitle)
      { this.dvdLowercaseTitle.resetTitle(dvdUpcaseTitle.getTitle()); }
    public void changeTitle(DvdLowercaseTitle dvdLowercaseTitle)
      { this.dvdUpcaseTitle.resetTitle(dvdLowercaseTitle.getTitle()); }
}
```

Mediator

- TestDvdMediator.java

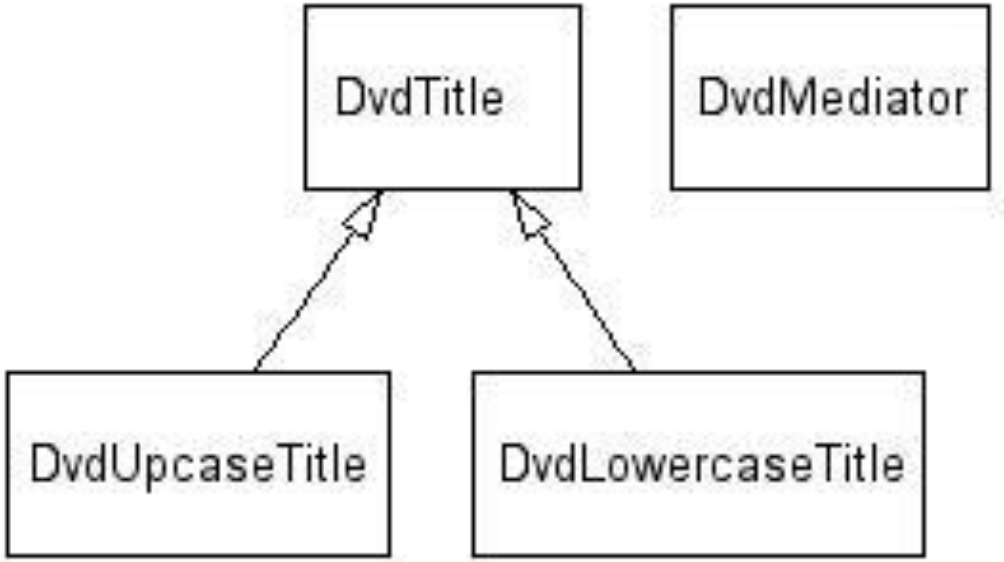
```
// testing the Mediator
class TestDvdMediator
{
public static void main(String[] args)
{
 DvdMediator dvdMediator = new DvdMediator();
 DvdLowercaseTitle dvdLower = new DvdLowercaseTitle("Mulholland  Dr.",
dvdMediator);
 DvdUpcaseTitle dvdUp = new DvdUpcaseTitle(dvdLower, dvdMediator);

 System.out.println("Lowercase LC title :" + dvdLower.getLowercaseTitle());
 System.out.println("Lowercase super title :" + dvdLower.getTitle());
 System.out.println("Upcase UC title :" + dvdUp.getUpcaseTitle());
 System.out.println("Upcase super title :" + dvdUp.getTitle());

 dvdLower.setSuperTitleLowercase();
 System.out.println(" ");
```

```
        System.out.println("After Super  set to LC");
        System.out.println("Lowercase  LC title :" + dvdLower.getLowercaseTitle());
        System.out.println("Lowercase  super title :" + dvdLower.getTitle());
        System.out.println("Upcase  UC title :" + dvdUp.getUpcaseTitle());
        System.out.println("Upcase  super title :" + dvdUp.getTitle());
    }
}
```

- Test Results

```
Lowercase LC title :mulholland dr.
Lowercase super title :Mulholland Dr.
Upcase UC title :MULHOLLAND DR.
Upcase super title :Mulholland Dr.


After Super  set to LC
Lowercase LC title :mulholland dr.
Lowercase super title :mulholland dr.
Upcase UC title :MULHOLLAND DR.
Upcase super title :mulholland dr.
```

Mediator

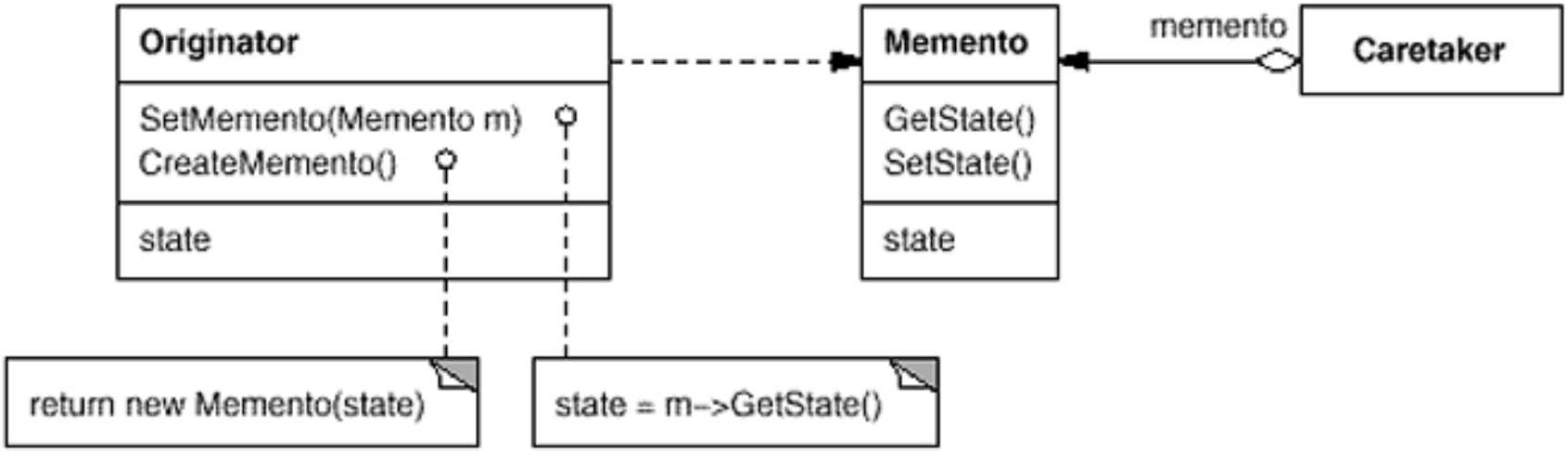

DvdTitle
DvdMediator
DvdUpcaseTitle
DvdLowercaseTitle

Memento

- Intent
  - Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
- In 4S application
  - We want to remove the abnormal prices of retails by restoring the previous prices.
  - Thus, memento can be used to backup the normal prices.

Memento

- Structure



Memento

- Applicability

- Use the Memento pattern when
  - a snapshot of (some portion of) an object's state must be saved so that it can be restored to that state later, and
  - a direct interface to obtaining the state would expose implementation details and break the object's encapsulation.

Memento

- Consequences

- Memento pattern has several consequences:
  - Preserving encapsulation boundaries.
  - It simplifies Originator.
  - Using mementos might be expensive.
  - Defining narrow and wide interfaces.
  - Hidden costs in caring for mementos.

Memento

- Implementation

- Here are two issues to consider when implementing the Memento pattern:
  - Language support.
  - Storing incremental changes.

Memento

- DvdDetails.java -the Originator (the class to be backed up) Contains the inner class DvdMemento -the Memento

```
import java.util.ArrayList;
import java.util.ListIterator;
public class DvdDetails //the originator
{
    private String titleName;
    private ArrayList stars;
    private char encodingRegion;
    public DvdDetails(String titleName, ArrayList stars, char encodingRegion)
    {
       this.setTitleName(titleName);
       this.setStars(stars);
       this.setEncodingRegion(encodingRegion);
    }
    private void setTitleName(String titleNameIn)
      { this.titleName = titleNameIn; }
    private String getTitleName()
      {return this.titleName;}
```

Memento


```
private void setStars(ArrayList starsIn)
   {this.stars = starsIn;}
public void addStar(String starIn)
   {stars.add(starIn);}
private ArrayList getStars()
   {return this.stars;}
private static String getStarsString(ArrayList starsIn)
{
      int count = 0;
      StringBuffer sb = new StringBuffer();
      ListIterator starsIterator = starsIn.listIterator();
      while (starsIterator.hasNext())
      {
         if (count++ > 0)
            {sb.append(", ");}
         sb.append((String)starsIterator.next());
      }
      return sb.toString();
}
```

Memento


```
private void setEncodingRegion(char  encodingRegionIn)
  { this.encodingRegion  = encodingRegionIn; }
private char getEncodingRegion()
  {return this.encodingRegion;}
public String formatDvdDetails()
  { return ("DVD: " + this.getTitleName() + ", starring: " + getStarsString(getStars())  + ",
       encoding region: " + this.getEncodingRegion());  }

//sets current state to what DvdMemento has
public void setDvdMemento(DvdMemento  dvdMementoIn)
  { dvdMementoIn.getState(); }

//save current state of DvdDetails in a DvdMemento
public DvdMemento createDvdMemento()
{
    DvdMemento mementoToReturn = new DvdMemento();
    mementoToReturn.setState();
    return mementoToReturn;
}
```

Memento


```
//an inner class for the memento
class DvdMemento
{
    private String mementoTitleName;
    private ArrayList mementoStars;
    private char mementoEncodingRegion;
    //sets DvdMementoData to DvdDetails
    public void setState()
    {
        //Because String are immutable we can just set the
        // DvdMemento Strings to = the DvdDetail Strings.
        mementoTitleName = getTitleName();
        mementoEncodingRegion = getEncodingRegion();
        //However, ArrayLists are not immutable,
        // so we need to instantiate a new ArrayList.
        mementoStars = new ArrayList(getStars());
    }
```

Memento


```
    //resets DvdDetails to DvdMementoData
    public void getState()
    {
       setTitleName(mementoTitleName);
       setStars(mementoStars);
       setEncodingRegion(mementoEncodingRegion);
    }
    //only useful for testing
    public String showMemento()
    {
       return ("DVD: " + mementoTitleName + ", starring: " +
     getStarsString(mementoStars) + ", encoding region: " + mementoEncodingRegion);
    }
  }
}
```

Memento

- TestDvdMemento.java
  testing the Memento contains a Caretaker object

```
import java.util.ArrayList;
import java.util.ArrayList;
public class TestDvdMemento
{
   public static void main(String[] args)
   {
      DvdDetails.DvdMemento dvdMementoCaretaker;
      ArrayList stars = new ArrayList();
      stars.add(new String("Guy Pearce"));
      DvdDetails dvdDetails = new DvdDetails("Memento", stars, '1');
      dvdMementoCaretaker = dvdDetails.createDvdMemento();
      System.out.println("as first instantiated");
      System.out.println(dvdDetails.formatDvdDetails());
      System.out.println("");

     //oops -Cappuccino on the keyboard!
      dvdDetails.addStar("edskdzkvdfb");
```

```
    System.out.println("after star added incorrectly");
    System.out.println(dvdDetails.formatDvdDetails());

    System.out.println("");
    System.out.println("the memento");
    //show the memento
    System.out.println(dvdMementoCaretaker.showMemento());

    System.out.println("");
    //back off changes
    dvdDetails.setDvdMemento(dvdMementoCaretaker);
    System.out.println("after DvdMemento state is restored to DvdDetails");
    System.out.println(dvdDetails.formatDvdDetails());
  }
}
```

Memento

- Test Results

as first instantiated

DVD: Memento, starring: Guy Pearce, encoding region: 1

after title set incorrectly

DVD: Memento, starring: Guy Pearce, edskdzkvdfb, encoding region: 1

the memento

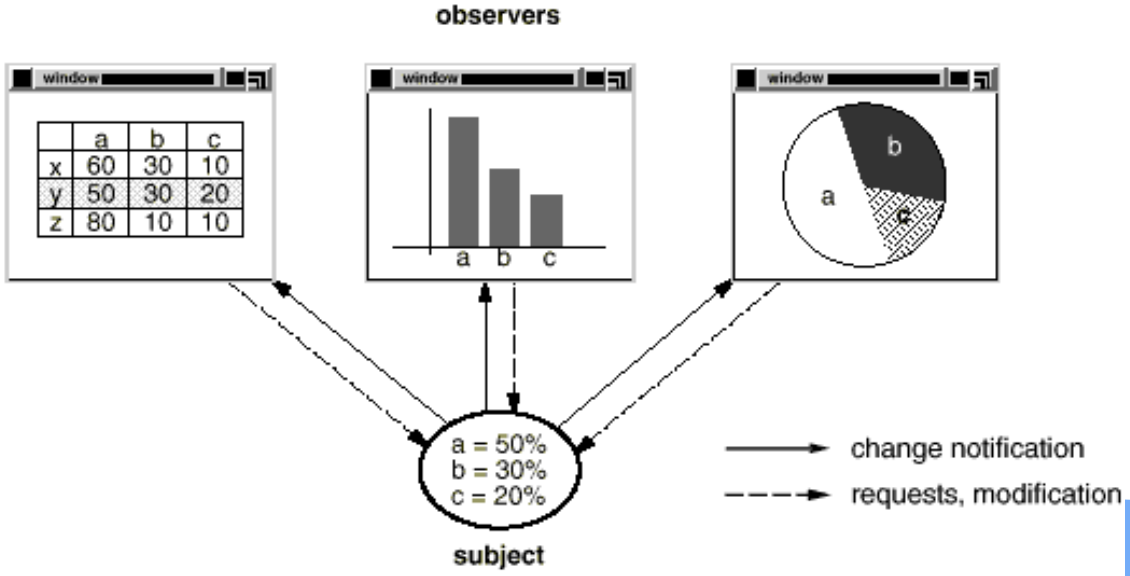
DVD: Memento, starring: Guy Pearce, encoding region: 1

after DvdMemento state is restored to DvdDetails

DVD: Memento, starring: Guy Pearce, encoding region: 1

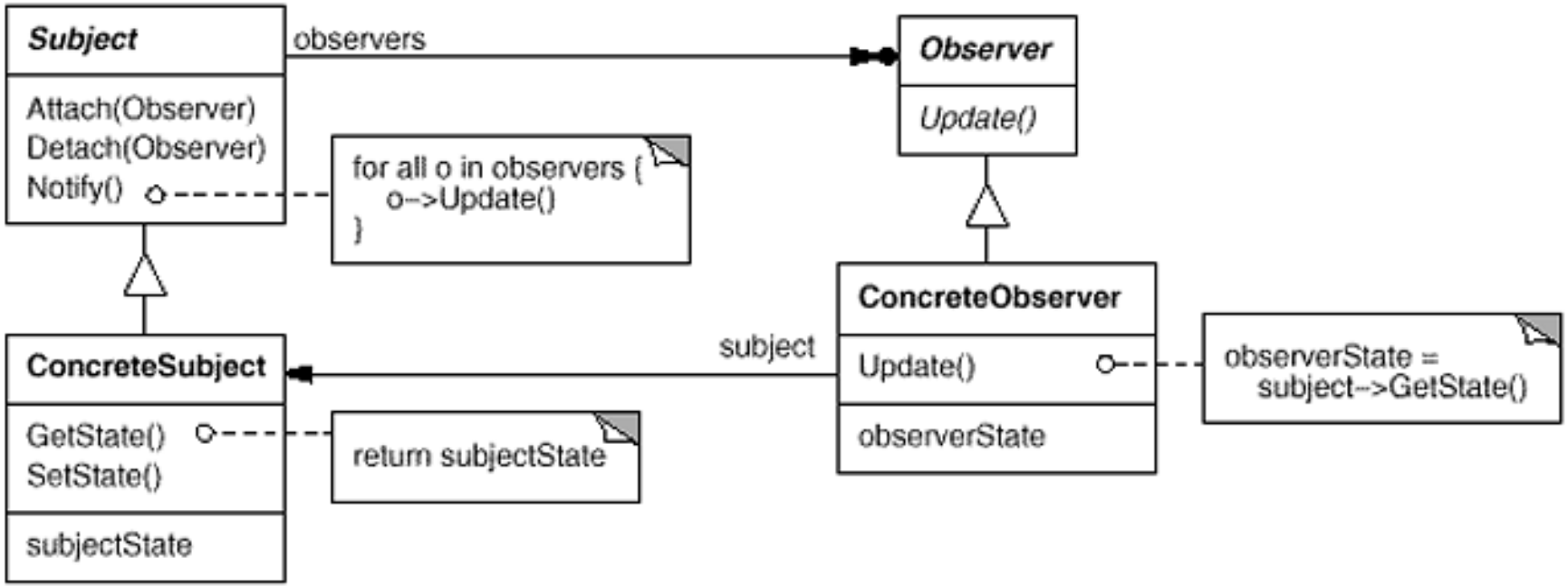
Observer

- Intent
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- In 4S application
  - A table and two charts are presented on the client-ends to show the price of cars.
  - They need to reflect the price up-to-date.



Observer

- Structure



Observer

- Applicability

- Use the Observer pattern in any of the following situations:
  - When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
  - When a change to one object requires changing others, and you don't know how many objects need to be changed.
  - When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

Observer

- Consequences

- The Observer pattern lets you vary subjects and observers independently.
  - You can reuse subjects without reusing their observers, and vice versa. It lets you add observers without modifying the subject or other observers.

- Further benefits and liabilities of the Observer pattern include the following:
  - Abstract coupling between Subject and Observer.
  - Support for broadcast communication.
  - Unexpected updates.

Observer

- Implementation

- Several issues related to the implementation of the dependency mechanism are discussed in this section.
  - Mapping subjects to their observers.
  - Observing more than one subject.
  - Who triggers the update?
  - Dangling references to deleted subjects.
  - Making sure Subject state is self-consistent before notification.
  - Avoiding observer-specific update protocols: the push and pull models.
  - Specifying modifications of interest explicitly.
  - Encapsulating complex update semantics.
  - Combining the Subject and Observer classes.

Observer

- DvdReleaseByCategory.java
  the subject (the class which is observed)

```
import java.util.ArrayList;
import java.util.ListIterator;
public class DvdReleaseByCategory
{
    String categoryName;
    ArrayList subscriberList = new ArrayList();
    ArrayList dvdReleaseList = new ArrayList();

    public DvdReleaseByCategory(String categoryNameIn)
      { categoryName = categoryNameIn; }
    public String getCategoryName()
      {return this.categoryName;}
    public boolean addSubscriber(DvdSubscriber dvdSubscriber)
      { return subscriberList.add(dvdSubscriber); }
```

Observer


```
public boolean removeSubscriber(DvdSubscriber  dvdSubscriber)
 {
    ListIterator listIterator = subscriberList.listIterator();
    while (listIterator.hasNext())
    {
        if (dvdSubscriber  == (DvdSubscriber)(listIterator.next()))
        {
            listIterator.remove();
            return true;
        }
    }
    return false;
}

public void newDvdRelease(DvdRelease dvdRelease)
{
   dvdReleaseList.add(dvdRelease);
   notifySubscribersOfNewDvd(dvdRelease);
}
```

Observer


```
public void updateDvd(DvdRelease dvdRelease)
{
    boolean dvdUpdated = false;
    DvdRelease tempDvdRelease;
    ListIterator listIterator = dvdReleaseList.listIterator();
    while (listIterator.hasNext())
    {
       tempDvdRelease = (DvdRelease)listIterator.next();
       if (dvdRelease.getSerialNumber().equals(tempDvdRelease.getSerialNumber()))
       {
          listIterator.remove();
          listIterator.add(dvdRelease);
          dvdUpdated = true;
          break;
       }
    }
    if (dvdUpdated == true) { notifySubscribersOfUpdate(dvdRelease);  }
    else { this.newDvdRelease(dvdRelease); }
}
```

Observer


```
    private void notifySubscribersOfNewDvd(DvdRelease  dvdRelease)
    {
       ListIterator listIterator = subscriberList.listIterator();
       while (listIterator.hasNext())
       {
         ((DvdSubscriber)(listIterator.next())).newDvdRelease(dvdRelease,
         this.getCategoryName());
       }
    }
    private void notifySubscribersOfUpdate(DvdRelease  dvdRelease)
    {
       ListIterator listIterator = subscriberList.listIterator();
       while (listIterator.hasNext())
       {
         ((DvdSubscriber)(listIterator.next())).updateDvdRelease(dvdRelease,
               this.getCategoryName() );
       }
    }
}
```

Observer

- DvdSubscriber.java
  the observer

```
public class DvdSubscriber
{
   private String subscriberName;
   public DvdSubscriber(String subscriberNameIn)
      { this.subscriberName = subscriberNameIn; }
   public String getSubscriberName()
      { return this.subscriberName; }
   public void newDvdRelease(DvdRelease newDvdRelease, String subscriptionListName)
   {
      System.out.println("");
      System.out.println("Hello " + this.getSubscriberName() + ", subscriber to the " +
         subscriptionListName + " DVD release list.");
      System.out.println("The new Dvd " + newDvdRelease.getDvdName() + " will be
         released on " + newDvdRelease.getDvdReleaseMonth() + "/" +
         newDvdRelease.getDvdReleaseDay() + "/" +
         newDvdRelease.getDvdReleaseYear() + ".");
}
```

Observer


```
 public void updateDvdRelease(DvdRelease newDvdRelease, String subscriptionListName)
 {
     System.out.println("");
     System.out.println("Hello " + this.getSubscriberName() + ", subscriber to the " +
          subscriptionListName + " DVD release list.");
     System.out.println("The following DVDs release has been revised: " +
          newDvdRelease.getDvdName() + " will be released on " +
          newDvdRelease.getDvdReleaseMonth() + "/" +
          newDvdRelease.getDvdReleaseDay() + "/" +
          newDvdRelease.getDvdReleaseYear() + ".");
 }
}
```

Observer

- DvdRelease.java
  a helper class

```
public class DvdRelease
{
    private String serialNumber;
    private String dvdName;
    private int dvdReleaseYear;
    private int dvdReleaseMonth;
    private int dvdReleaseDay;

    public DvdRelease(String serialNumber, String dvdName, int dvdReleaseYear, int
      dvdReleaseMonth, int dvdReleaseDay)
    {
       setSerialNumber(serialNumber);
       setDvdName(dvdName);
       setDvdReleaseYear(dvdReleaseYear);
       setDvdReleaseMonth(dvdReleaseMonth);
       setDvdReleaseDay(dvdReleaseDay);
}
```

Observer


```
public void updateDvdRelease(String  serialNumber,  String dvdName, int dvdReleaseYear, int
   dvdReleaseMonth,  int dvdReleaseDay)
{
     setSerialNumber(serialNumber);
     setDvdName(dvdName);
     setDvdReleaseYear(dvdReleaseYear);
     setDvdReleaseMonth(dvdReleaseMonth);
     setDvdReleaseDay(dvdReleaseDay);
}
public void updateDvdReleaseDate(int  dvdReleaseYear, int dvdReleaseMonth,  int dvdReleaseDay)
{
    setDvdReleaseYear(dvdReleaseYear);
    setDvdReleaseMonth(dvdReleaseMonth);
    setDvdReleaseDay(dvdReleaseDay);
}
public void setSerialNumber(String  serialNumberIn)
  { this.serialNumber  = serialNumberIn;  }
public String getSerialNumber()
  { return this.serialNumber;  }
```

Observer


```
  public void setDvdName(String dvdNameIn)
     {this.dvdName = dvdNameIn;}
  public String getDvdName()
     {return this.dvdName;}
  public void setDvdReleaseYear(int dvdReleaseYearIn)
     { this.dvdReleaseYear = dvdReleaseYearIn; }
  public int getDvdReleaseYear()
     {return this.dvdReleaseYear;}
  public void setDvdReleaseMonth(int dvdReleaseMonthIn)
     { this.dvdReleaseMonth = dvdReleaseMonthIn; }
  public int getDvdReleaseMonth()
     { return this.dvdReleaseMonth; }
  public void setDvdReleaseDay(int dvdReleaseDayIn)
     { this.dvdReleaseDay = dvdReleaseDayIn; }
  public int getDvdReleaseDay()
     {return this.dvdReleaseDay;}
}
```

Observer

- TestDvdObserver.java
  testing the observer

```
class TestDvdObserver
{
    public static void main(String[] args)
    {
        DvdReleaseByCategory btvs = new DvdReleaseByCategory("Buffy the Vampire Slayer");
        DvdReleaseByCategory simpsons = new DvdReleaseByCategory("The Simpsons");
        DvdReleaseByCategory sopranos = new DvdReleaseByCategory("The Sopranos");
        DvdReleaseByCategory xfiles = new DvdReleaseByCategory("The X-Files");

        DvdSubscriber jsopra = new DvdSubscriber("Junior  Soprano");
        DvdSubscriber msimps = new DvdSubscriber("Maggie  Simpson");
        DvdSubscriber rgiles = new DvdSubscriber("Rupert  Giles");
        DvdSubscriber smulde = new DvdSubscriber("Samantha  Mulder");
        DvdSubscriber wrosen = new DvdSubscriber("Willow Rosenberg");

        btvs.addSubscriber(rgiles);
        btvs.addSubscriber(wrosen);
```

Observer


```
simpsons.addSubscriber(msimps);
sopranos.addSubscriber(jsopra);
xfiles.addSubscriber(smulde);
xfiles.addSubscriber(wrosen);

DvdRelease btvsS2 = new DvdRelease("DVDFOXBTVSS20", "Buffy The Vampire
    Slayer Season 2", 2002, 06, 11);
DvdRelease simpS2 = new DvdRelease("DVDFOXSIMPSO2", "The Simpsons
    Season 2", 2002, 07, 9);
DvdRelease soprS2 = new DvdRelease("DVDHBOSOPRAS2", "The Sopranos
    Season 2", 2001, 11, 6);
DvdRelease xfilS5 = new DvdRelease("DVDFOXXFILES5", "The X-Files Season
    5", 2002, 04, 1);

btvs.newDvdRelease(btvsS2);
simpsons.newDvdRelease(simpS2);
sopranos.newDvdRelease(soprS2);
xfiles.newDvdRelease(xfilS5);
```

```
        xfiles.removeSubscriber(wrosen);
        xfilS5.updateDvdReleaseDate(2002, 5, 14);
        xfiles.updateDvd(xfilS5);
    }
}
```

- Test Results

```
Hello Rupert Giles, subscriber to the Buffy the Vampire Slayer DVD release list.
The new Dvd Buffy The Vampire Slayer Season 2 will be released on 6/11/2002.

Hello Willow Rosenberg, subscriber to the Buffy the Vampire Slayer DVD release list.
The new Dvd Buffy The Vampire Slayer Season 2 will be released on 6/11/2002.

Hello Maggie Simpson, subscriber to the The Simpsons DVD release list.
The new Dvd The Simpsons Season 2 will be released on 7/9/2002.

Hello Junior Soprano, subscriber to the The Sopranos DVD release list.
The new Dvd The Sopranos Season 2 will be released on 11/6/2001.
```

Observer

Hello Samantha Mulder, subscriber to the The X-Files DVD release list.
The new Dvd The X-Files Season 5 will be released on 4/1/2002.

Hello Willow Rosenberg, subscriber to the The X-Files DVD release list.
The new Dvd The X-Files Season 5 will be released on 4/1/2002.

Hello Samantha Mulder, subscriber to the The X-Files DVD release list.
The following DVDs release has been revised: The X-Files Season 5 will be released on 5/14/2002.

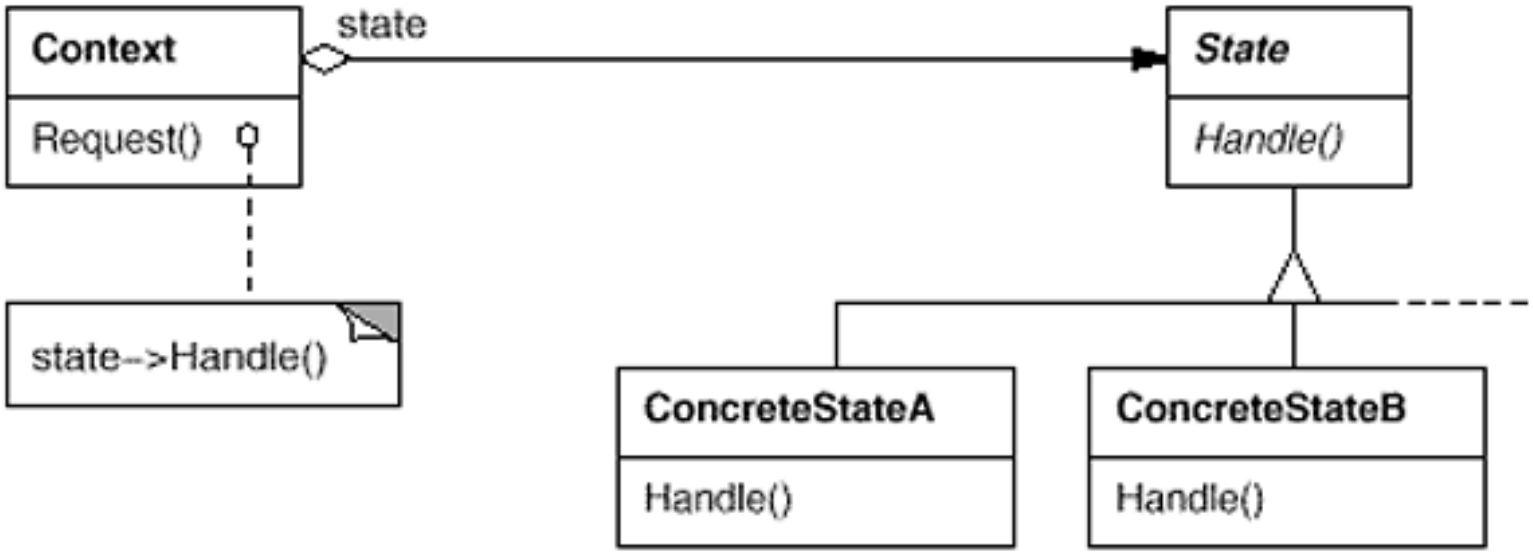
State

- Intent
  - Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

- In 4S application
  - All the clients communicate with server via a mediator.
  - If the queue of mediator is full, no request of client can be pushed into it.
  - While if the queue is not full, new request will be pushed into it.
  - Thus, for clients, the state of mediator determines the behavior of server.
  - State pattern is suitable for such scenario.

State

- Structure



- Applicability

- Use the State pattern in either of the following cases:
  - An object's behavior depends on its state, and it must change its behavior at run-time depending on that state.
  - Operations have large, multipart conditional statements that depend on the object's state.
    - This state is usually represented by one or more enumerated constants.
    - Often, several operations will contain this same conditional structure.
    - State pattern puts each branch of the conditional in a separate class.
    - This lets you treat the object's state as an object in its own right that can vary independently from other objects.

- Consequences

- The State pattern has the following consequences:
  - It localizes state-specific behavior and partitions behavior for different states.
  - It makes state transitions explicit.
  - State objects can be shared.

- Implementation

- The State pattern raises a variety of implementation issues:
  - Who defines the state transitions?
  - A table-based alternative.
  - Creating and destroying State objects.
  - Using dynamic inheritance.

State

- DvdStateContext.java
  the Context

```
public class DvdStateContext
{
   private DvdStateName dvdStateName;
   public DvdStateContext()
   {
     setDvdStateName(new DvdStateNameStars()); //start with stars
   }
   public void setDvdStateName(DvdStateName dvdStateNameIn)
     { this.dvdStateName = dvdStateNameIn; }
   public void showName(String nameIn)
     { this.dvdStateName.showName(this, nameIn); }
}
```

State

- DvdStateName.java
  the State interface

```
public interface DvdStateName
{
public void showName(DvdStateContext dvdStateContext, String nameIn);
}
```

- DvdStateNameExclaim.java
  one of two Concrete States

```
public class DvdStateNameExclaim implements DvdStateName
{
    public DvdStateNameExclaim() {}
    public void showName(DvdStateContext dvdStateContext, String nameIn)
    {
        System.out.println(nameIn.replace(' ','!'));
        //show exclaim only once, switch back to stars
       dvdStateContext.setDvdStateName(new DvdStateNameStars());
    }
}
```

- DvdStateNameStars.java
  two of two Concrete States

```
public class DvdStateNameStars implements DvdStateName
{
   int starCount;
   public DvdStateNameStars() { starCount = 0; }
   public void showName(DvdStateContext dvdStateContext, String nameIn)
   {
       System.out.println(nameIn.replace(' ','*'));
       // show stars twice, switch to exclamation point
      if (++starCount > 1)
      {
          dvdStateContext.setDvdStateName(new DvdStateNameExclaim());
      }
   }
}
```

State

- TestState.java
  testing the State

```
class TestState
{
   public static void main(String[]  args)
   {
      DvdStateContext stateContext = new DvdStateContext();
      stateContext.showName(  "Sponge  Bob Squarepants  –Nautical Nonsense  and Sponge  Buddies");
      stateContext.showName("Jay  and Silent  Bob Strike  Back");
      stateContext.showName("Buffy  The Vampire  Slayer Season  2");
      stateContext.showName("The  Sopranos Season  2");
   }
}
```

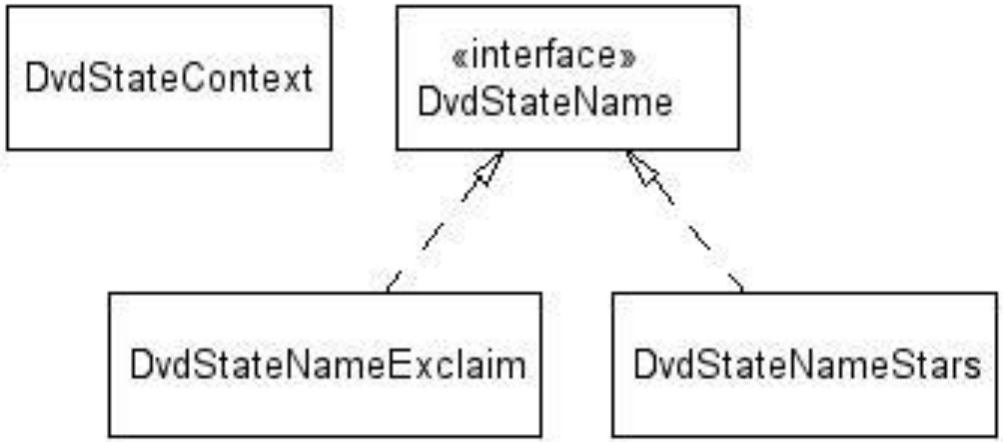
- Test Results

Sponge*Bob*Squarepants*-*Nautical*Nonsense*and*Sponge*Buddies

Jay*and*Silent*Bob*Strike*Back

Buffy!The!Vampire!Slayer!Season!2

The*Sopranos*Season*2



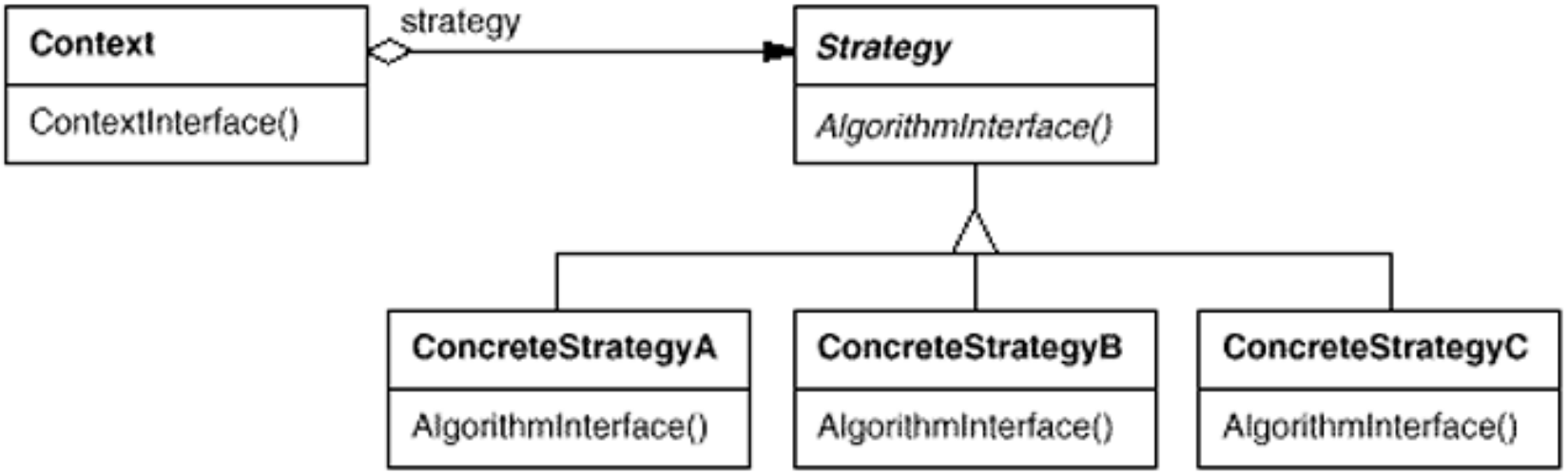
Strategy

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

- In 4S application
  - The customers are classified as Gold, Silver and Bronze levels.
  - For different customers, the processing of their order is also different.
  - To achieve this design goal, Strategy pattern is used.

Strategy

- Structure



Strategy

- Applicability

- Use the Strategy pattern when
  - many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
  - you need different variants of an algorithm.
  - an algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.
  - a class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.

Strategy

- Consequences

- The Strategy pattern has the following benefits and drawbacks:
  - Families of related algorithms.
  - An alternative to subclassing.
  - Strategies eliminate conditional statements.
  - A choice of implementations.
  - Clients must be aware of different Strategies.
  - Communication overhead between Strategy and Context.
  - Increased number of objects.

Strategy

- Implementation

- Consider the following implementation issues:
  - Defining the Strategy and Context interfaces.
  - Strategies as template parameters.
  - Making Strategy objects optional.

Strategy

DvdNameStrategy.java

the abstract strategy

```
public abstract class DvdNameStrategy {
   public abstract String formatDvdName(String dvdName, char charIn);
}
```

Strategy

DvdNameAllCapStrategy.java: one of three concrete strategies

```
public class DvdNameAllCapStrategy extends DvdNameStrategy {
  public String formatDvdName(String dvdName, char charIn) {
    return dvdName.toUpperCase();
  }
}
```

DvdNameReplaceSpacesStrategy.java: two of three concrete strategies

```
public class DvdNameReplaceSpacesStrategy extends DvdNameStrategy
{
  public String formatDvdName(String dvdName, char charIn) {
    return dvdName.replace(' ', charIn);
  }
}
```

Strategy

DvdNameTheAtEndStrategy.java: three of three concrete strategies

```
public class DvdNameTheAtEndStrategy extends DvdNameStrategy {
  public String formatDvdName(String dvdName, char charIn) {
    if (dvdName.startsWith("The "))
        { return new String(dvdName.substring(4, (dvdName.length())) + ",
        The"); }
    if (dvdName.startsWith("THE "))
       { return new String(dvdName.substring(4, (dvdName.length())) + ",
         THE"); }
    if (dvdName.startsWith("the "))
       { return new String(dvdName.substring(4, (dvdName.length())) + ",
         the"); }
    return dvdName;
  }
}
```

Strategy

DvdNameContext.java: the context

```
public class DvdNameContext {
   private DvdNameStrategy dvdNameStrategy;
   public DvdNameContext(char strategyTypeIn) {
      switch (strategyTypeIn) {
        case 'C' :
            this.dvdNameStrategy = new DvdNameAllCapStrategy();
            break;
        case 'E' :
            this.dvdNameStrategy = new DvdNameTheAtEndStrategy();
            break;
        case 'S' :
            this.dvdNameStrategy = new DvdNameReplaceSpacesStrategy();
            break;
         default :
            this.dvdNameStrategy = new DvdNameTheAtEndStrategy();
   }
}
```

```
 public String[] formatDvdNames(String[] namesIn)
  { return this.formatDvdNames(namesIn, ' '); }
 public String[] formatDvdNames(String[] namesIn, char replacementIn)
 {
  String[] namesOut = new String[namesIn.length];
  for (int i = 0; i < namesIn.length; i++) {
  namesOut[i] = dvdNameStrategy.formatDvdName(namesIn[i],
       replacementIn);
  System.out.println("Dvd name before formatting: " + namesIn[i]);
  System.out.println("Dvd name after formatting: " + namesOut[i]);
  System.out.println("============================");
 }
 return namesOut;
}
}
```

Strategy

TestDvdStrategy.java:testing the strategy

```
class TestDvdStrategy {
  public static void main(String[] args) {
    DvdNameContext allCapContext = new DvdNameContext('C');
    DvdNameContext theEndContext = new DvdNameContext('E');
    DvdNameContext spacesContext = new DvdNameContext('S');
    String dvdNames[] = new String[3];
    dvdNames[0] = "Jay and Silent Bob Strike Back";
    dvdNames[1] = "The Fast and the Furious";
    dvdNames[2] = "The Others";
    char replaceChar = '*';
    System.out.println("Testing formatting with all caps");
    String[] dvdCapNames = allCapContext.formatDvdNames(dvdNames);
    System.out.println(" ");
    System.out.println("Testing formatting with beginning the at end");
    String[] dvdEndNames = theEndContext.formatDvdNames(dvdNames);
    System.out.println(" ");
    System.out.println("Testing formatting with all spaces replaced with " + replaceChar);
    String[] dvdSpcNames = spacesContext.formatDvdNames(dvdNames, replaceChar);
  }
}
```

Strategy

Test Results

Testing formatting with all caps

Dvd name before formatting: Jay and Silent Bob Strike Back

Dvd name after formatting: JAY AND SILENT BOB STRIKE BACK

==========

Dvd name before formatting: The Fast and the Furious

Dvd name after formatting: THE FAST AND THE FURIOUS

==========

Dvd name before formatting: The Others

Dvd name after formatting: THE OTHERS

==========

Testing formatting with beginning the at end

Dvd name before formatting: Jay and Silent Bob Strike Back

Dvd name after formatting: Jay and Silent Bob Strike Back

==========

Dvd name before formatting: The Fast and the Furious

Dvd name after formatting: Fast and the Furious, The

==========

Dvd name before formatting: The Others

Dvd name after formatting: Others, The

==========

Testing formatting with all spaces replaced with *

Dvd name before formatting: Jay and Silent Bob Strike Back

Dvd name after formatting: Jay*and*Silent*Bob*Strike*Back

==========

Dvd name before formatting: The Fast and the Furious

Dvd name after formatting: The*Fast*and*the*Furious

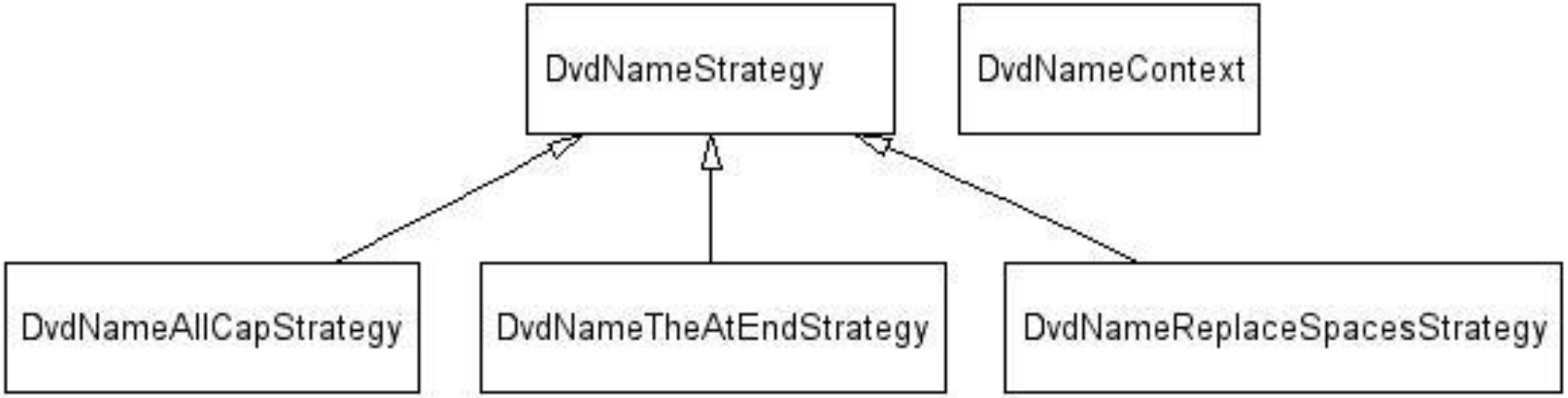
==========

Dvd name before formatting: The Others

Dvd name after formatting: The*Others

==========

Strategy


DvdNameStrategy
DvdNameContext
DvdNameAllCapStrategy
DvdNameTheAtEndStrategy
DvdNameReplaceSpacesStrategy

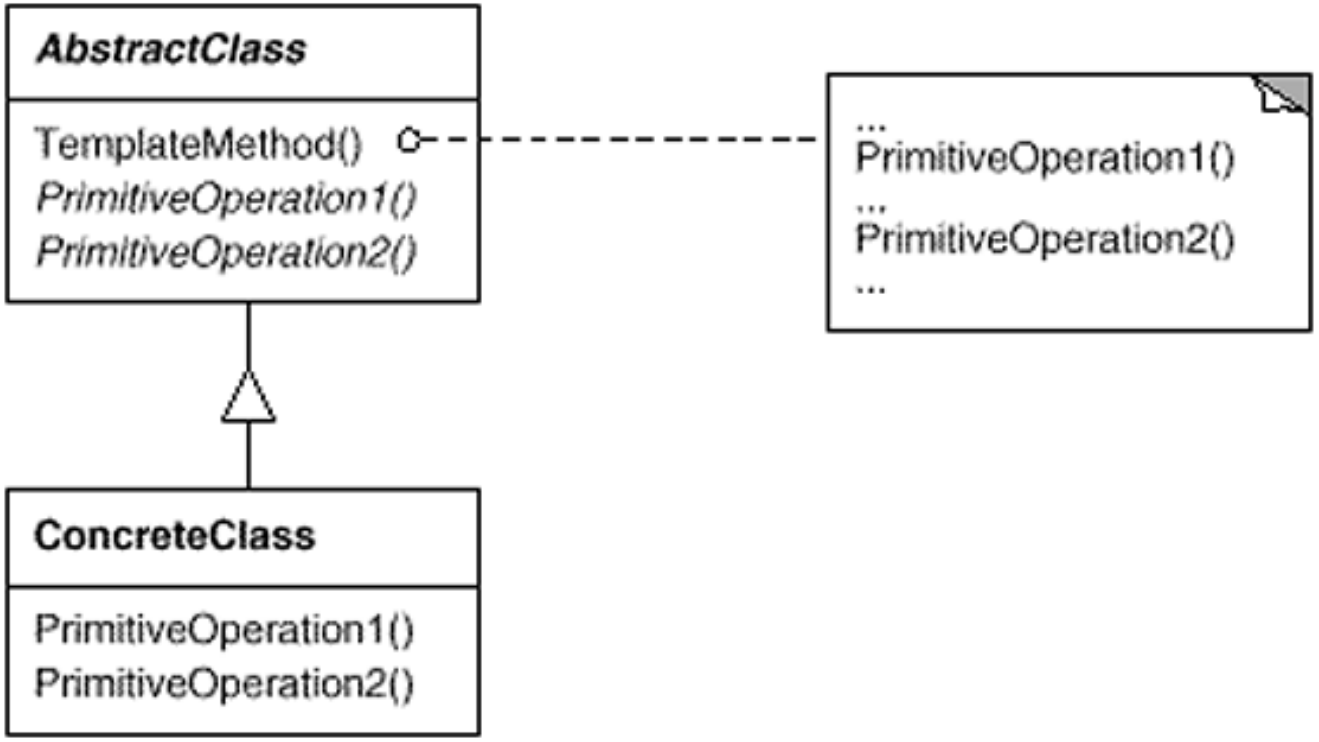
Template Method

- Intent
  - Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

- In 4S application
  - We use Strategy pattern to process the orders of different customer.
  - If all the strategies look almost same except for several steps, we can use Template Method pattern to establish the processing flow.
  - Thus, the specific implementation of a strategy just needs to implement such steps but not the whole flow.

Template Method

- Structure



Template Method

- Applicability

- The Template Method pattern should be used
  - to implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary.
  - when common behavior among subclasses should be factored and localized in a common class to avoid code duplication.
  - to control subclasses extensions. You can define a template method that calls "hook" operations at specific points, thereby permitting extensions only at those points.

Template Method

- Consequences

- Template methods are a fundamental technique for code reuse.
  - They are particularly important in class libraries, because they are the means for factoring out common behavior in library classes.
- Template methods lead to an inverted control structure
  - that's sometimes referred to as "the Hollywood principle," that is, "Don't call us, we'll call you". This refers to how a parent class calls the operations of a subclass and not the other way around.

Template Method

- Implementation

- Three implementation issues are worth noting:
  - Access control.
  - Minimizing primitive operations.
  - Naming conventions.

Template Method

TitleInfo.java：the abstract Template

```
public abstract class TitleInfo {
    private String titleName;
    //the "template method" -calls the concrete class methods, is not overridden
    public final String ProcessTitleInfo() {
    StringBuffer titleInfo = new StringBuffer();
    titleInfo.append(this.getTitleBlurb());
    titleInfo.append(this.getDvdEncodingRegionInfo());
    return titleInfo.toString();
    }
    //the following 2 methods are "concrete abstract class methods"
    public final void setTitleName(String titleNameIn)
        {this.titleName = titleNameIn;}
    public final String getTitleName()
        {return this.titleName;}
```

Template Method


```
    //this is a "primitive operation", and must be overridden in the concretetemplates
    public abstract String getTitleBlurb();
    //this is a "hook operation", which may be overridden,
    //hook operations usually do nothing if not overridden
    public String getDvdEncodingRegionInfo()
        {return " ";}
}
```

Template Method

DvdTitleInfo.java：one of three concrete templates

```
public class DvdTitleInfo extends TitleInfo {
    private String star;
    private char encodingRegion;
    public DvdTitleInfo(String titleName, String star, char encodingRegion)
    {
        this.setTitleName(titleName);
        this.setStar(star);
        this.setEncodingRegion(encodingRegion);
    }
    public void setStar(String starIn) {this.star = starIn;}
    public String getStar() {return this.star;}
    public void setEncodingRegion(char encodingRegionIn)
        {this.encodingRegion = encodingRegionIn;}
    public char getEncodingRegion()
        {return this.encodingRegion;}
    public String getTitleBlurb()
        { return ("DVD: " + this.getTitleName() + ", starring " + this.getStar()); }
    public String getDvdEncodingRegionInfo()
        { return (", encoding region: " + this.getEncodingRegion()); }
}
```

Template Method

BookTitleInfo.java

two of three concrete templates

```
public class BookTitleInfo extends TitleInfo {
  private String author;
  public BookTitleInfo(String titleName, String author)
  {
      this.setTitleName(titleName);
      this.setAuthor(author);
  }
  public void setAuthor(String authorIn)
      {this.author = authorIn;}
  public String getAuthor()
      {return this.author;}
  public String getTitleBlurb()
      { return ("Book title: " + this.getTitleName() + ", Author: " +
        this.getAuthor()); }
}
```

Template Method

GameTitleInfo.java

three of three concrete templates

```
public class GameTitleInfo extends TitleInfo {
  public GameTitleInfo(String titleName)
    { this.setTitleName(titleName); }
  public String getTitleBlurb()
    { return ("Game: " + this.getTitleName()); }
}
```

Template Method

TestTitleInfoTemplate.java

testing the Template

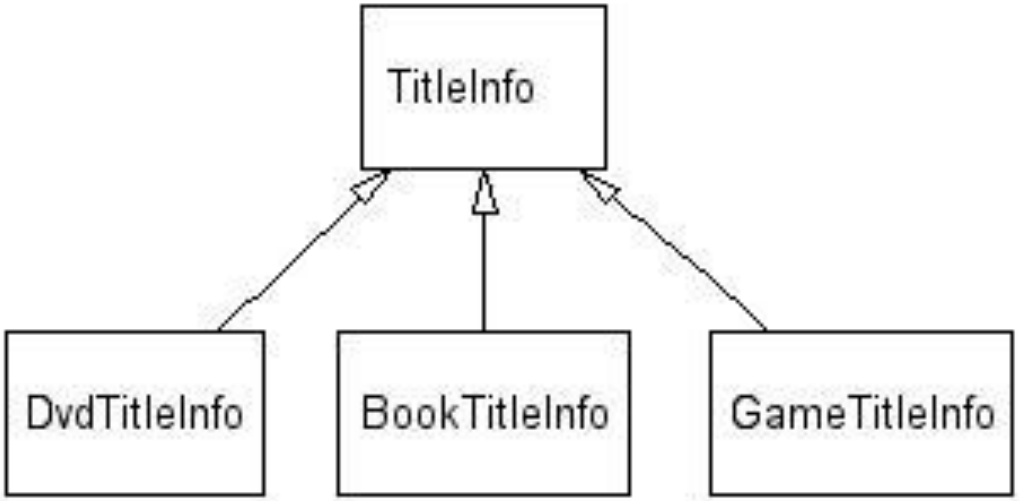
```
class TestTitleInfoTemplate {
  public static void main(String[] args) {
  TitleInfo bladeRunner = new DvdTitleInfo("Blade Runner", "Harrison Ford", '1');
  TitleInfo electricSheep = new BookTitleInfo("Do Androids Dream of Electric
      Sheep?", "Phillip K. Dick");
  TitleInfo sheepRaider = new GameTitleInfo("Sheep Raider");
  System.out.println(" ");
  System.out.println("Testing bladeRunner " + bladeRunner.ProcessTitleInfo());
  System.out.println("Testing electricSheep " + electricSheep.ProcessTitleInfo());
  System.out.println("Testing sheepRaider " + sheepRaider.ProcessTitleInfo());
 }
}
```

Test Results

Testing bladeRunner DVD: Blade Runner, starring Harrison Ford, encoding region: 1

Testing electricSheep Book: Do Androids Dream of Electric Sheep?, Author: Phillip K. Dick

Testing sheepRaider Game: Sheep Raider



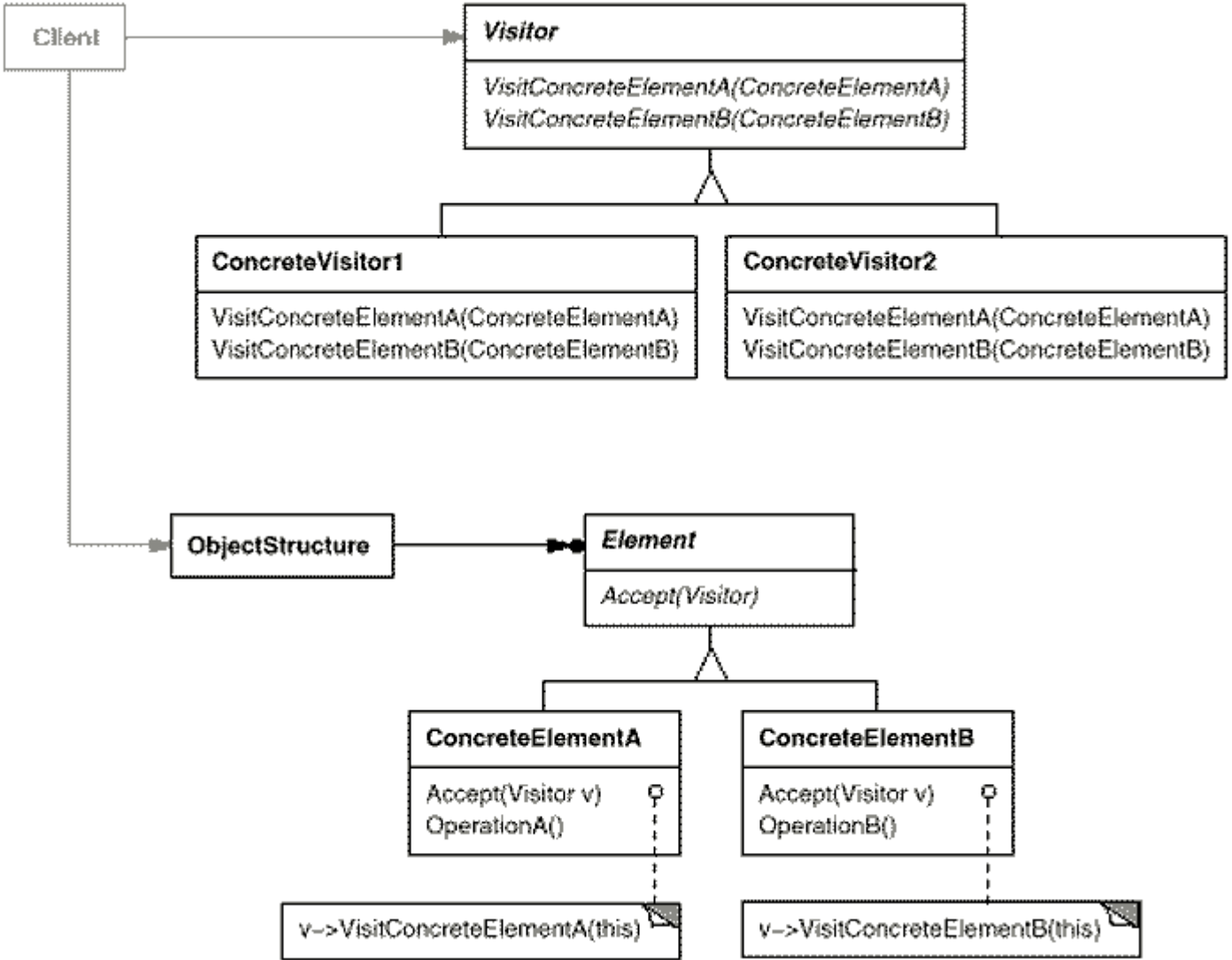
Visitor

- Intent
  - Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

- In 4S application
  - Since the customers are classified into Gold, Silver and Bronze levels.The customer of different level can view different information of cars.
    - For example, the Gold customers can see the 3D view of car, while the Silver customers can see the 2D view of car, and Bronze customers can see the thumb view of car.
  - Visitor Pattern can be applied into this case.

Visitor

- Structure



Visitor

- Applicability

- Use the Visitor pattern when
  - an object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes.
  - many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations.
  - the classes defining the object structure rarely change, but you often want to define new operations over the structure.

- Consequences

- Some of the benefits and liabilities of the Visitor pattern are as follows:
  - Visitor makes adding new operations easy.
  - A visitor gathers related operations and separates unrelated ones.
  - Adding new ConcreteElement classes is hard.
  - Visiting across class hierarchies.
  - Accumulating state.
  - Breaking encapsulation.

Visitor

- Implementation

- Each object structure will have an associated Visitor class.
- Here are two other implementation issues that arise when you apply the Visitor pattern:
  - Double dispatch.
  - Who is responsible for traversing the object structure?

Visitor

- TitleBlurbVisitor.java
  the abstract Visitor

```
public abstract class TitleBlurbVisitor
{
   String titleBlurb;
   public void setTitleBlurb(String blurbIn)
     {this.titleBlurb = blurbIn;}
   public String getTitleBlurb()
     {return this.titleBlurb;}
   public abstract void visit(BookInfo bookInfo);
   public abstract void visit(DvdInfo dvdInfo);
   public abstract void visit(GameInfo gameInfo);
}
```

Visitor

- TitleLongBlurbVisitor.java
  one of two concrete Visitors

```
public class TitleLongBlurbVisitor extends TitleBlurbVisitor
{
   public void visit(BookInfo bookInfo)
   {
      this.setTitleBlurb("LB-Book: " + bookInfo.getTitleName() + ", Author: " +
         bookInfo.getAuthor());
   }
   public void visit(DvdInfo dvdInfo)
   {
      this.setTitleBlurb("LB-DVD: " + dvdInfo.getTitleName() + ", starring " +
         dvdInfo.getStar() + ", encoding region: " + dvdInfo.getEncodingRegion());
   }
   public void visit(GameInfo gameInfo)
     { this.setTitleBlurb("LB-Game: " + gameInfo.getTitleName()); }
}
```

Visitor

- TitleShortBlurbVisitor.java
  two of two concrete Visitors

```
public class TitleShortBlurbVisitor extends TitleBlurbVisitor
{
   public void visit(BookInfo bookInfo)
   {
       this.setTitleBlurb("SB-Book: " + bookInfo.getTitleName());
   }
   public void visit(DvdInfo dvdInfo)
   {
      this.setTitleBlurb("SB-DVD: " + dvdInfo.getTitleName());
   }
   public void visit(GameInfo gameInfo)
  {
      this.setTitleBlurb("SB-Game: " + gameInfo.getTitleName());
  }
}
```

- AbstractTitleInfo.java

  the abstract Visitee

```
public abstract class AbstractTitleInfo
{
    private String titleName;
    public final void setTitleName(String titleNameIn)
        { this.titleName = titleNameIn; }
    public final String getTitleName()
        { return this.titleName; }
    public abstract void accept(TitleBlurbVisitor titleBlurbVisitor);
}
```

Visitor

- BookInfo.java
  one of three concrete Visitees

```
public class BookInfo extends AbstractTitleInfo
{
    private String author;
    public BookInfo(String titleName, String author)
    {
        this.setTitleName(titleName);
        this.setAuthor(author);
    }
    public void setAuthor(String authorIn)
        { this.author = authorIn; }
    public String getAuthor()
        {return this.author;}
    public void accept(TitleBlurbVisitor titleBlurbVisitor)
        { titleBlurbVisitor.visit(this); }
}
```

- DvdInfo.java：two of three concrete Visitees

```
public class DvdInfo extends AbstractTitleInfo
{
   private String star;
   private char encodingRegion;
   public DvdInfo(String titleName, String star, char encodingRegion)
   {
        this.setTitleName(titleName);
        this.setStar(star);
        this.setEncodingRegion(encodingRegion);
   }
   public void setStar(String starIn)
       {this.star = starIn;}
   public String getStar()
       {return this.star;}
   public void setEncodingRegion(char encodingRegionIn)
       { this.encodingRegion = encodingRegionIn; }
   public char getEncodingRegion()
       {return this.encodingRegion;}
   public void accept(TitleBlurbVisitor titleBlurbVisitor)
       { titleBlurbVisitor.visit(this); }
}
```

Visitor

- GameInfo.java

  three of three concrete Visitees

```
public class GameInfo extends AbstractTitleInfo
{
    public GameInfo(String titleName)
      { this.setTitleName(titleName); }
    public void accept(TitleBlurbVisitor titleBlurbVisitor)
      { titleBlurbVisitor.visit(this); }
}
```

Visitor

- TestTitleVisitor.java testing the Visitor

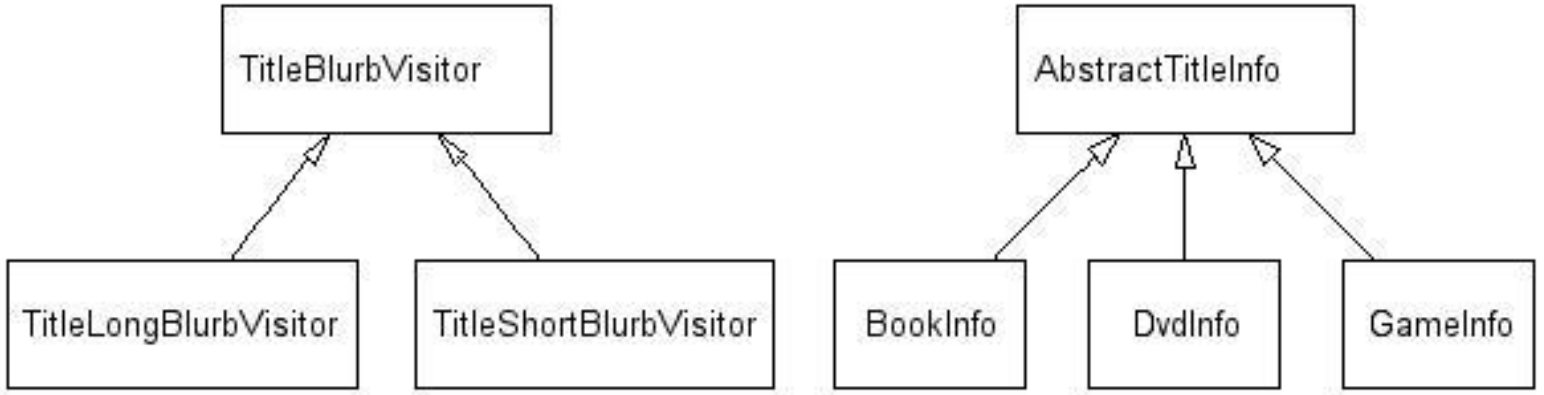
```
class TestTitleVisitor
{
    public static void main(String[] args)
    {
        AbstractTitleInfo bladeRunner = new DvdInfo("Blade Runner", "Harrison Ford", '1');
        AbstractTitleInfo electricSheep = new BookInfo("Do Androids Dream of Electric Sheep?",
            "Phillip K. Dick");
        AbstractTitleInfo sheepRaider = new GameInfo("Sheep Raider");

        TitleBlurbVisitor titleLongBlurbVisitor = new TitleLongBlurbVisitor();

        System.out.println("Long Blurbs:");
        bladeRunner.accept(titleLongBlurbVisitor);
        System.out.println("Testing bladeRunner " + titleLongBlurbVisitor.getTitleBlurb());
        electricSheep.accept(titleLongBlurbVisitor);
        System.out.println("Testing electricSheep " + titleLongBlurbVisitor.getTitleBlurb());
        sheepRaider.accept(titleLongBlurbVisitor);
        System.out.println("Testing sheepRaider " + titleLongBlurbVisitor.getTitleBlurb());
```

```
    TitleBlurbVisitor titleShortBlurbVisitor = new TitleShortBlurbVisitor();
    System.out.println("Short Blurbs:");
    bladeRunner.accept(titleShortBlurbVisitor);
    System.out.println("Testing bladeRunner " + titleShortBlurbVisitor.getTitleBlurb());
    electricSheep.accept(titleLongBlurbVisitor);
    System.out.println("Testing electricSheep " + titleShortBlurbVisitor.getTitleBlurb());
            sheepRaider.accept(titleShortBlurbVisitor);
    System.out.println("Testing sheepRaider " + titleShortBlurbVisitor.getTitleBlurb());
  }
}
```

- Test Results

```
Long Blurbs:
Testing bladeRunner LB-DVD: Blade Runner, starring Harrison Ford, encoding region: 1
Testing electricSheep LB-Book: Do Androids Dream of Electric Sheep?, Author: Phillip K. Dick
Testing sheepRaider LB-Game: Sheep Raider
Short Blurbs:
Testing bladeRunner SB-DVD: Blade Runner
Testing electricSheep SB-DVD: Blade Runner
Testing sheepRaider SB-Game: Sheep Raider
```

Visitor


TitleBlurbVisitor
TitleLongBlurbVisitor
TitleShortBlurbVisitor
AbstractTitleInfo
BookInfo
DvdInfo
GameInfo

Design Patterns Categories

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy Flyweight	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Creational Patterns

- Abstract Factory
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- Builder
  - Separate the construction of a complex object from its representation so that the same construction process can create different representations.
- Factory Method
  - Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- Prototype
  - Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- Singleton
  - Ensure a class only has one instance, and provide a global point of access to it.

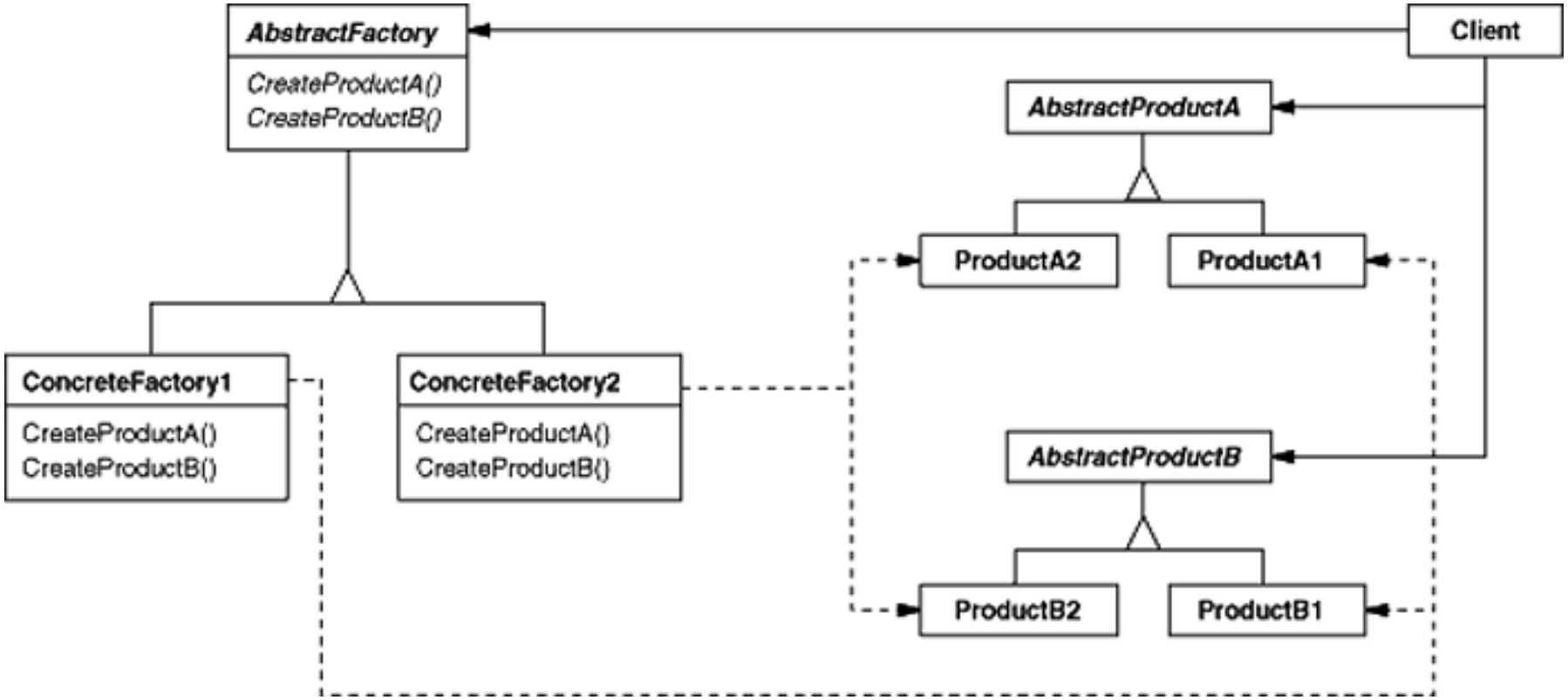
Abstract Factory

- Intent
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

- In 4S application
  - To guarantee the maintainability
    - The users are categorized into GoldUser and SilverUser.
    - GoldBid and SilverBidare related with GoldUser and SilverUser respectively.
  - When create a user object, its bid object should be created at the same time since they are independent on each other.
  - If we want to create different <user, bid> pairs in unified way, then we can use Abstract Factory pattern.

Abstract Factory

- Structure



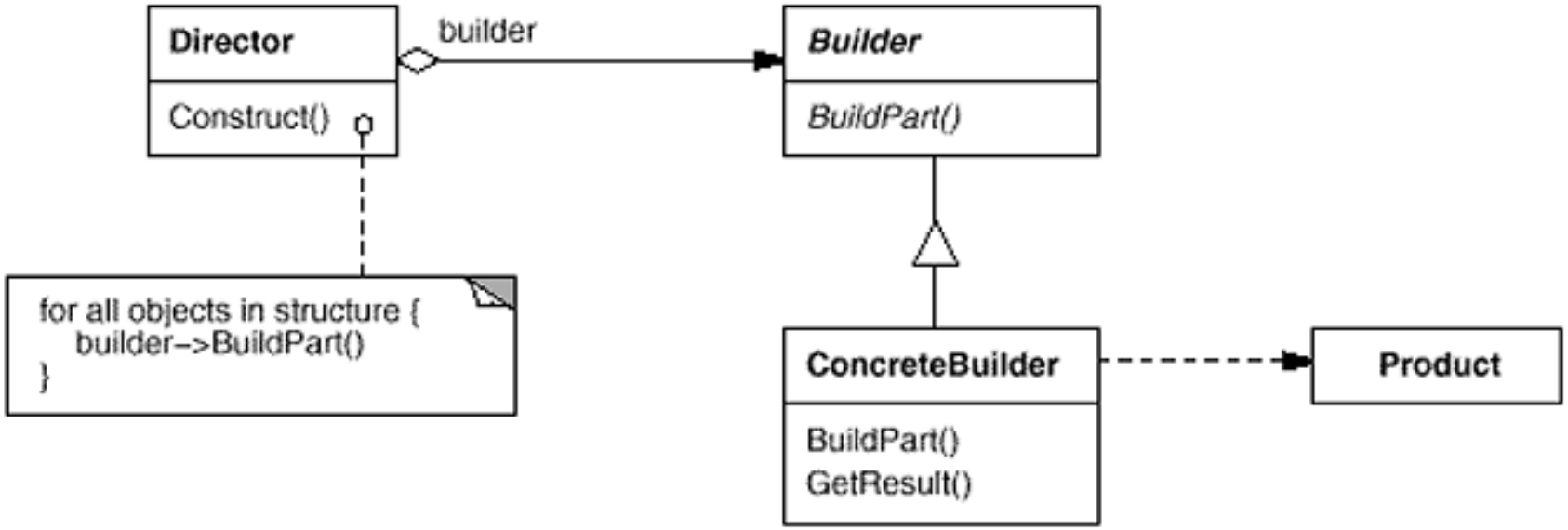
Builder

- Intent
  - Separate the construction of a complex object from its representation so that the same construction process can create different representations.

- In 4S application
  - We design a UserPack class which is a combination of User and Bid classes. A UserPack has a reference to a User and a reference to a Bid
  - When we create a UserPack object, we need to create the User and Bid objects it referred.
  - Builder pattern makes the creation of UserPack independent of the creation of User and Bid

Builder

- Structure



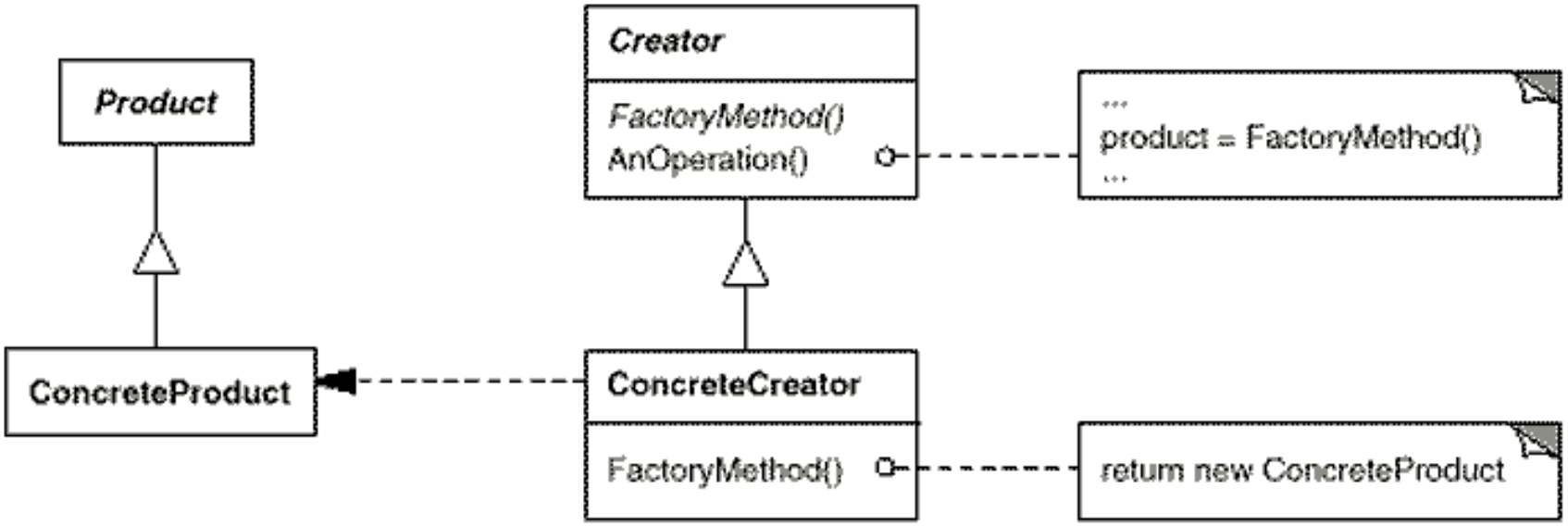
Factory Method

- Intent
  - Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

- In 4S application
  - The users are categorized into several types, such as GoldUser and SilverUser. Each of them has an associated pricing object.
  - We can create a user object and its associated pricing object together.
  - How to implement such logic?
  - For example, GoldUser and SilverUser has almost same logic for creation except for few points. As a result, how to implement the CreateUser() in the two specific factories?
  - The answer is Factory Method.

Factory Method

- Structure



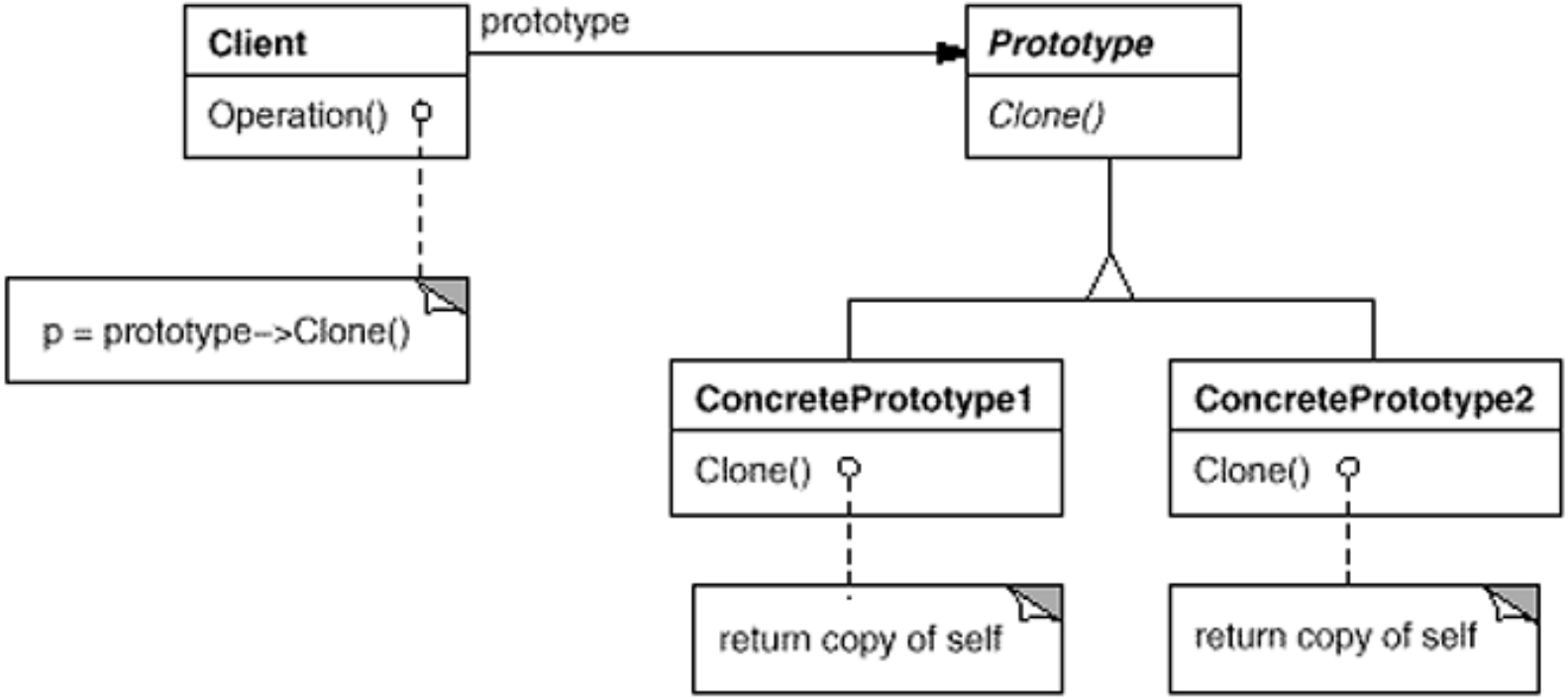
Prototype

- Intent
  - Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

- In 4S application
  - We design a Token class for workflow control.
  - A certain number of Token objects are created and pooled.
  - A client needs to obtain a Token object from the pool in order to be served by the application.
  - All the Token objects are almost same except for a bool member which indicates whether it is hold by a client.
  - Prototype pattern could be applied to instantiate Token object.

Prototype

- Structure



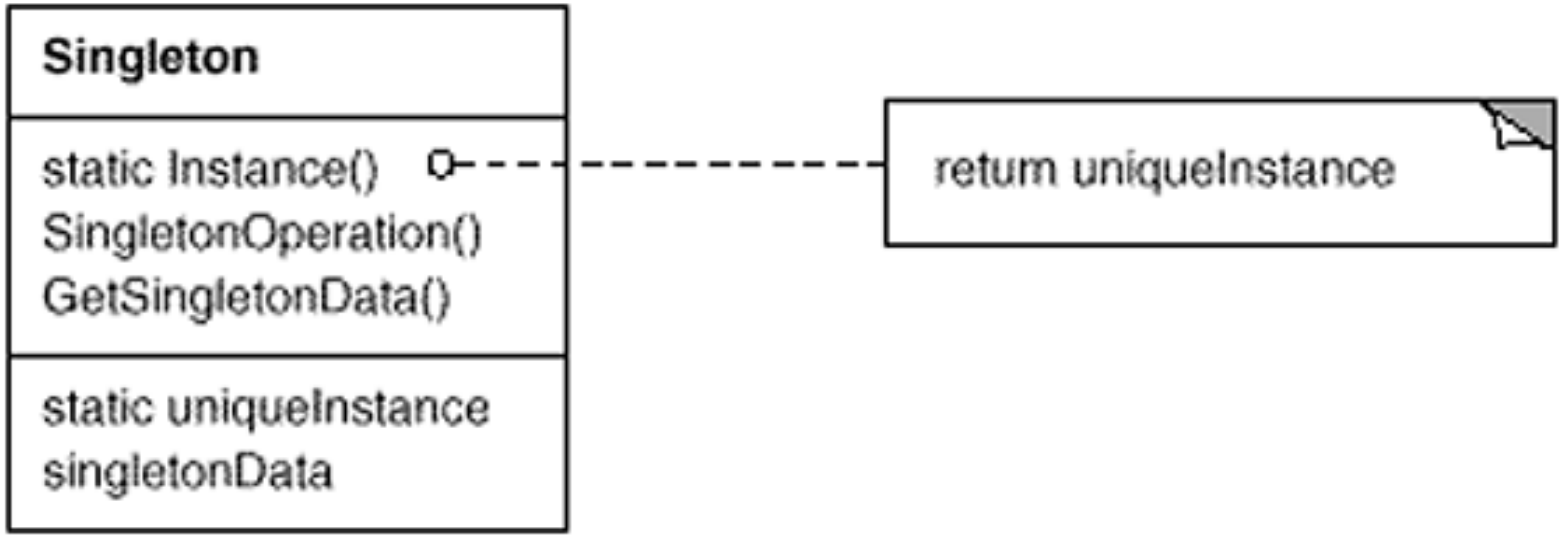
Singleton

- Intent
  - Ensure a class only has one instance, and provide a global point of access to it.

- In 4S application
  - In general, a Factory is a heavy-weight object since it is responsible to create objects.
  - As a result, creation and removal of a Factory is costly.
  - We want to reduce the number of such objects.
  - Singleton is what we want.

Singleton

- Structure



Structural Patterns

- Adapter
  - Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- Bridge
  - Decouple an abstraction from its implementation so that the two can vary independently.
- Composite
  - Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- Decorator
  - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- Façade
  - Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- Flyweight
  - Use sharing to support large numbers of fine-grained objects efficiently.
- Proxy
  - Provide a surrogate or placeholder for another object to control access to it.

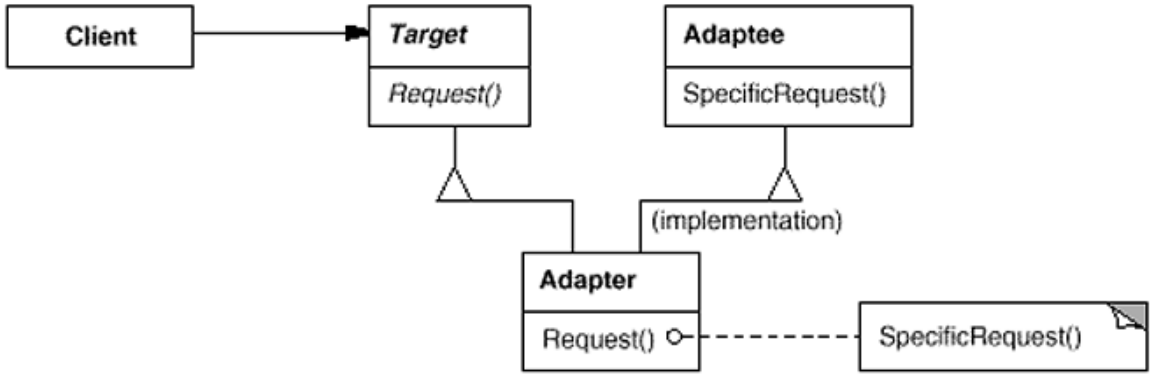
Adapter

- Intent
  - Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

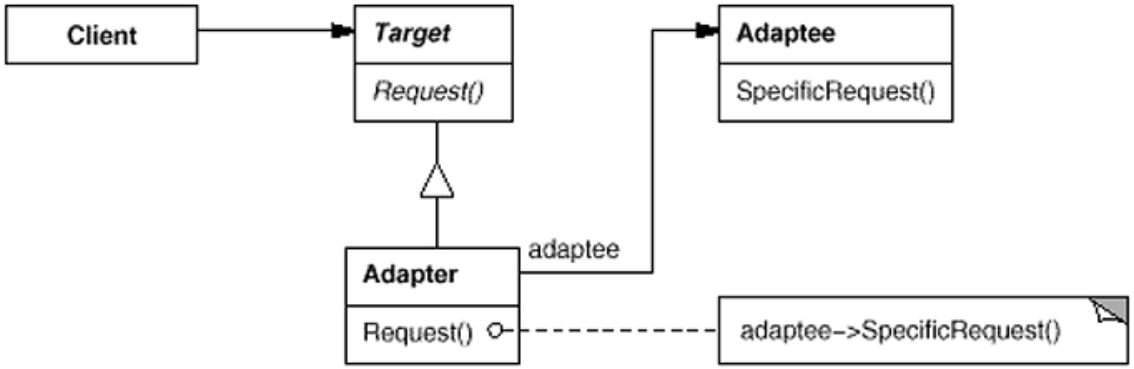
- In 4S application
  - The developers design to replace the AAA service with a new more powerful one.
  - However, the API of new one is different with the existing one.
  - To Avoid modification of code, the Adapter pattern is adopted.

Adapter

- Structure
  - A class adapter uses multiple inheritance to adapt one interface to another:



  - An object adapter relies on object composition:

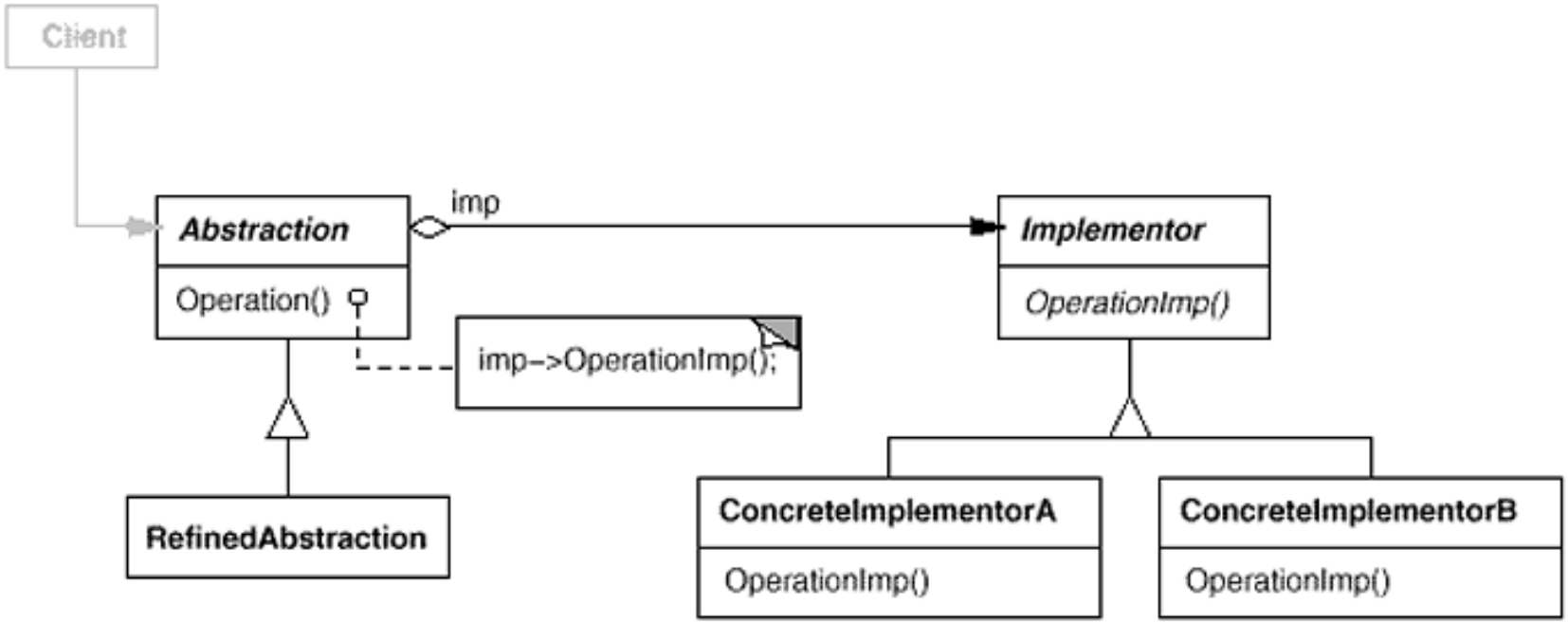


Bridge

- Intent
  - Decouple an abstraction from its implementation so that the two can vary independently.

- In 4S application
  - Beside Gold and Silver, the categories of users would be extended to include Bronze and Diamond.
  - On the other hand, beside email, the contact information of a user would be extended to include mobile phone, post address and account of SNS.
  - Thus, User class has two independent evolving attributes: Type and Contact.
  - Bridge pattern can be applied in this scenario.

- Structure


Client
Abstraction
Operation()
imp
imp->OperationImp();
Implementor
OperationImp()
RefinedAbstraction
ConcreteImplementorA
OperationImp()
ConcreteImplementorB
OperationImp()

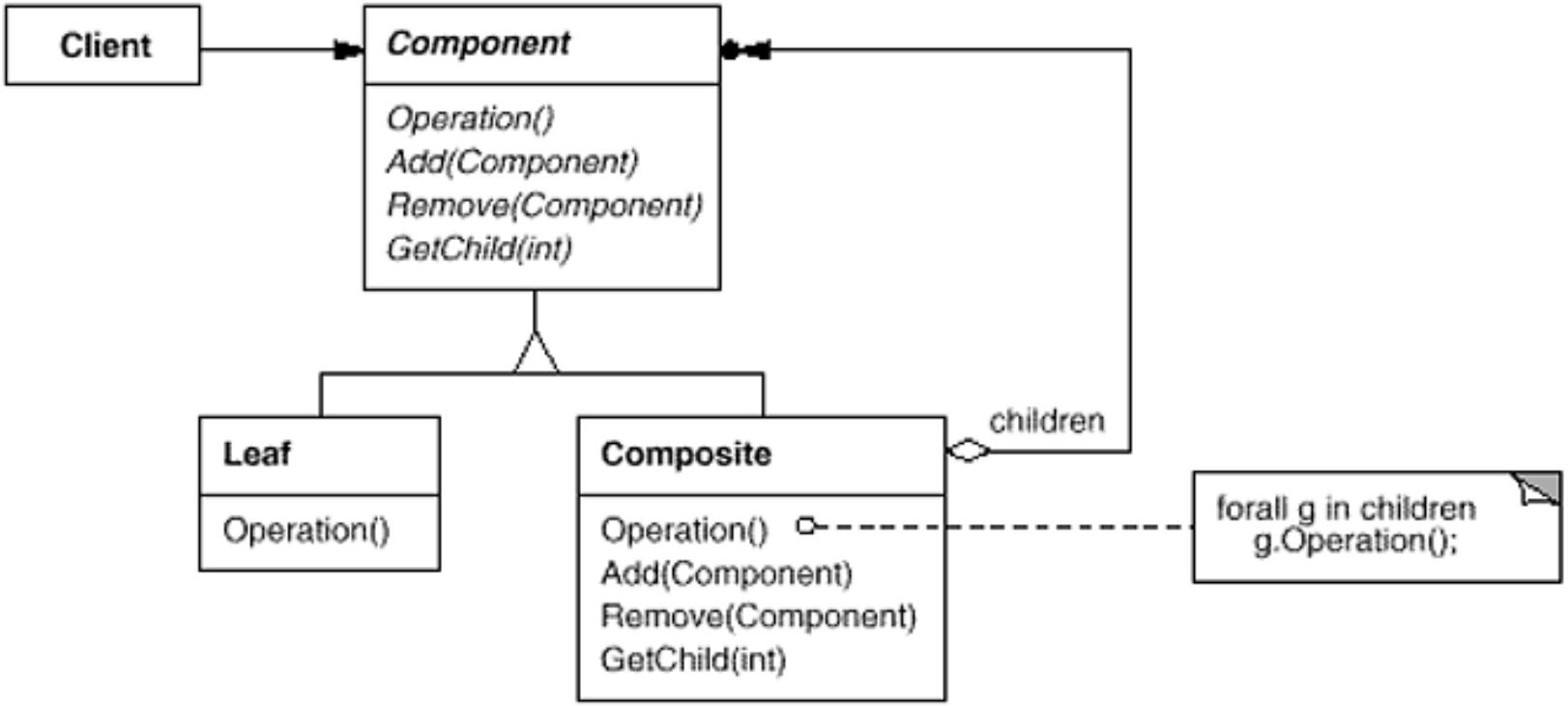
Composite

- Intent
  - Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

- In 4S application
  - The remarks of cars are organized into a tree.
  - Users can post remarks and replies.
  - Developers want to design a unified way to process a specific remark or the subtree of a remark and all its replies.
  - Composite pattern is suitable for this scenario

Composite

- Structure



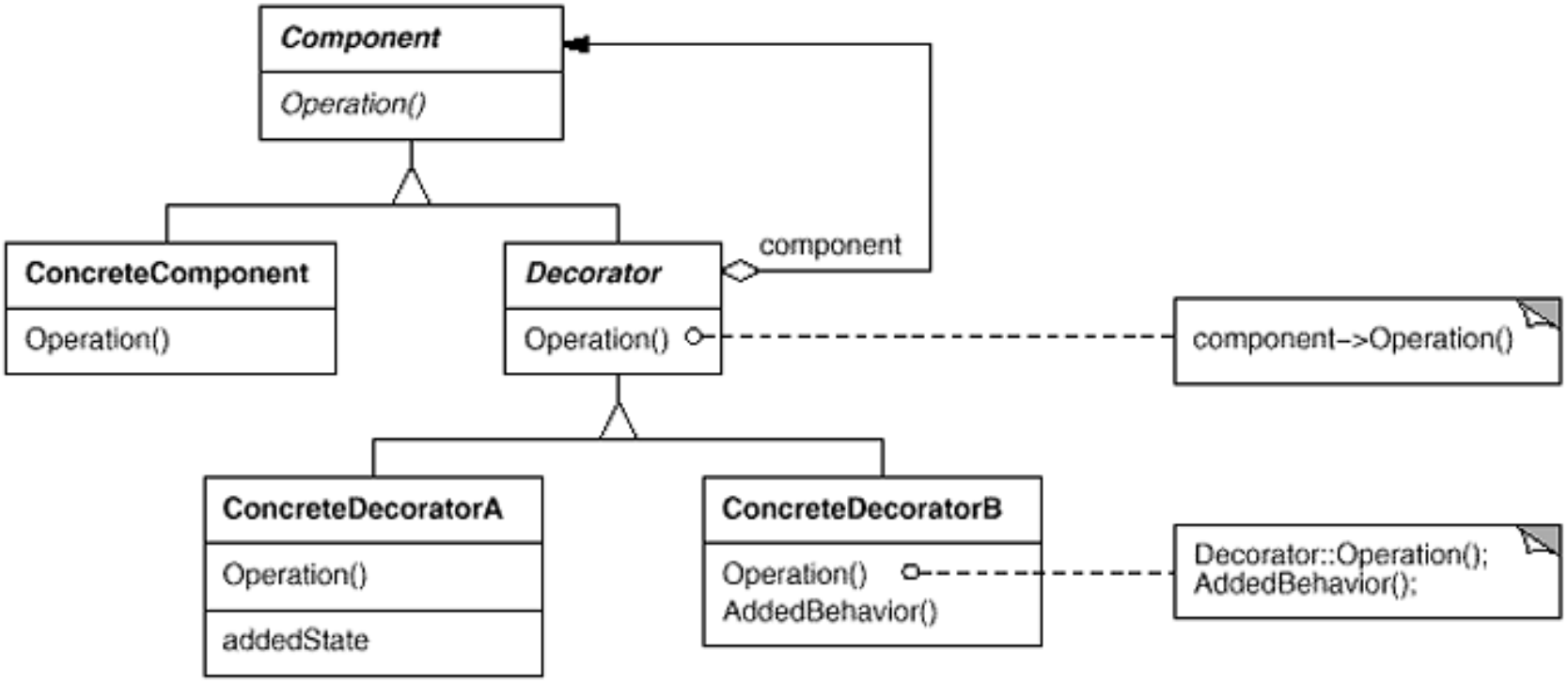
Decorator

- Intent
  - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality

- In 4S application
  - The 4S store will give free gifts to customers from time to time in order to promote the sale.
    - For example, toys are given to customers during the Christmas Holidays, and shopping cards are sent to customers during the Spring Festival
  - However, the gifts are not parts of cars. They are dynamically added to the cars during a special period.
  - So, the cars shouldn't be subclassed into various gift-cars, otherwise, the number of classes will be increased.
  - Decorator can resolve such problem.

Decorator

- Structure



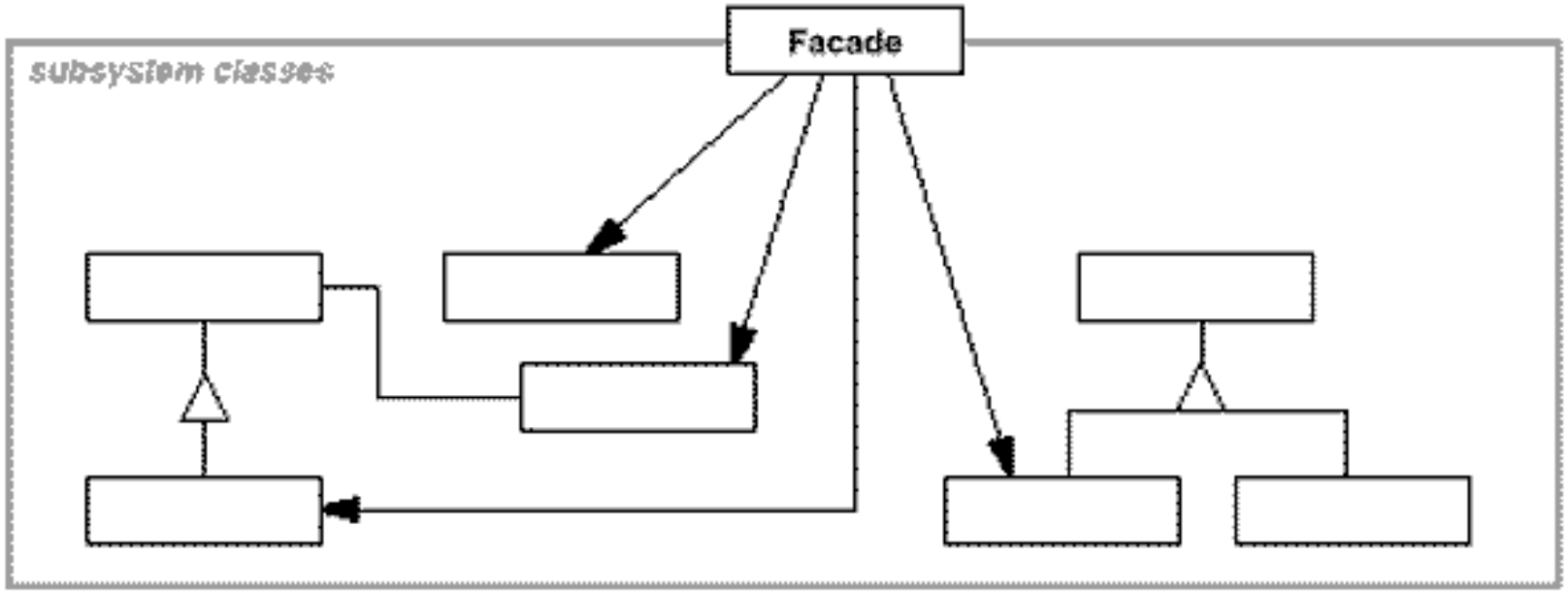
Façade

- Intent
  - Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

- In 4S application
  - Suppose there are three methods need to be invoked for processing the orders:
    - The first one is to authenticate the user with the username and password
    - The second one is to submit the order with the type of car
    - The third one is to retrieve the price and gifts with the type of car
  - In order to decouple the client code from the code for processing orders, the façade pattern is applied

Façade

- Structure



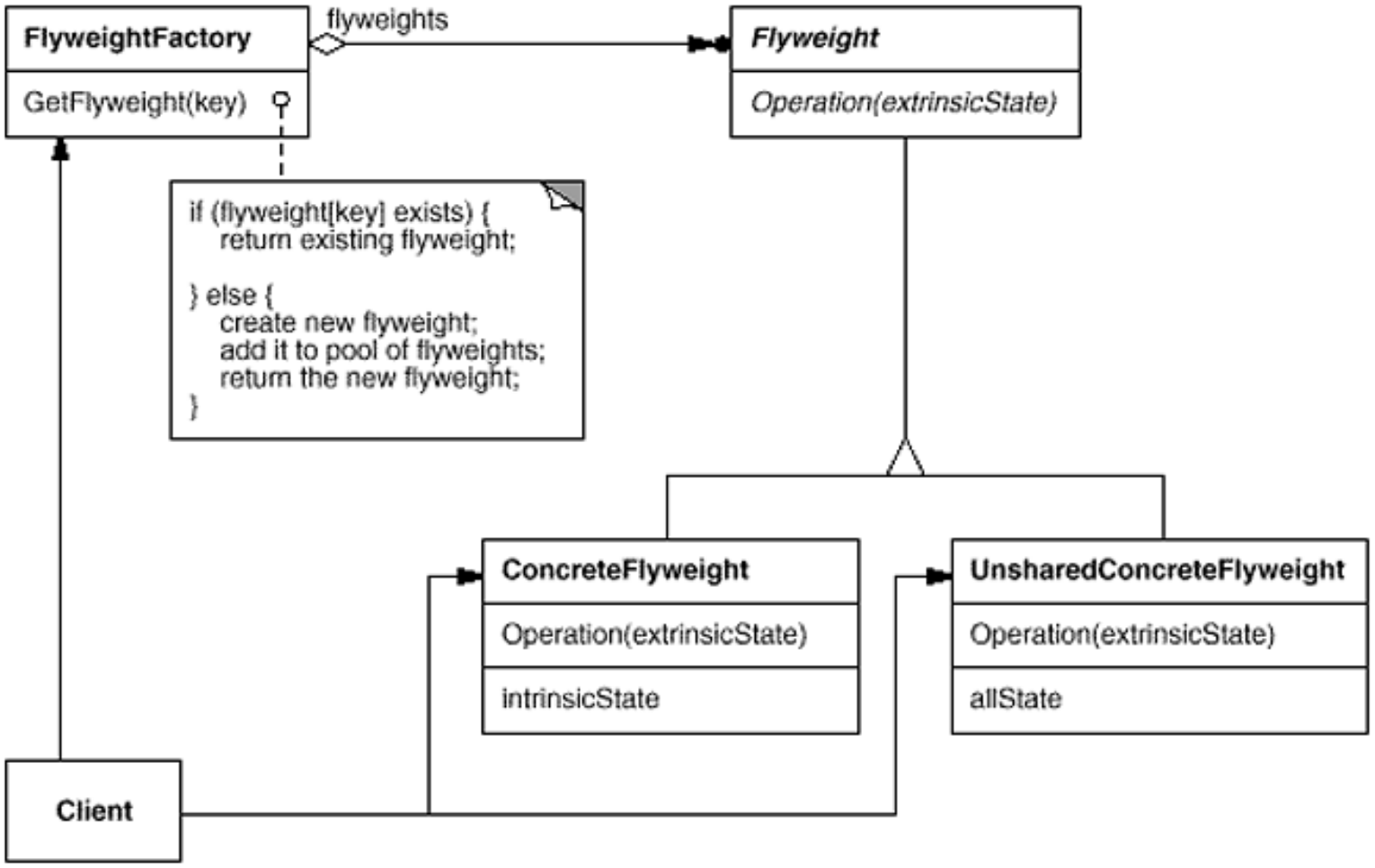
Flyweight

- Intent
  - Use sharing to support large numbers of fine-grained objects efficiently.

- In 4S application
  - We can create only 2 instances of Token in the example of Factory pattern of which the flag attributes are true and false respectively.
  - Thus, the 2 instances can serve thousands of clients by sharing the instances.
  - Flyweight pattern is a feasible way to share the same instance of Token class among multiple clients.

Flyweight

- Structure

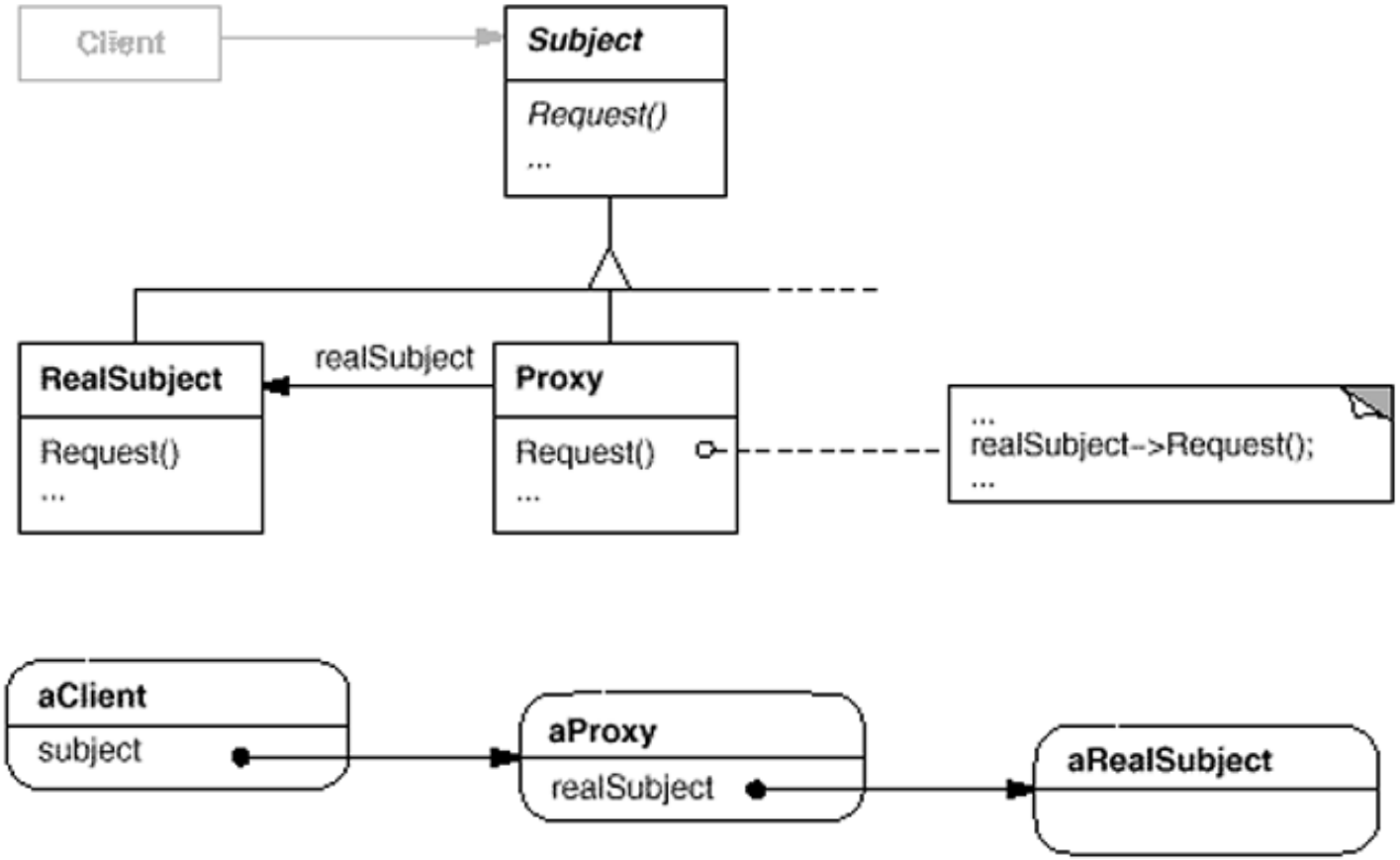


Proxy

- Intent
  - Provide a surrogate or placeholder for another object to control access to it.

- In 4S application
  - The application is deployed into Jboss
  - Jboss provide containers for EJB and Servlet
  - Container is composed of a set of proxies to provide various middleware services, such as transaction management, messaging, security control, the lifecycle management of objects, and so on.

- Structure


Client
Subject
Request()
...
RealSubject
Request()
...
realSubject
Proxy
Request()
...
...
realSubject->Request();
...
aClient
subject
aProxy
realSubject
aRealSubject

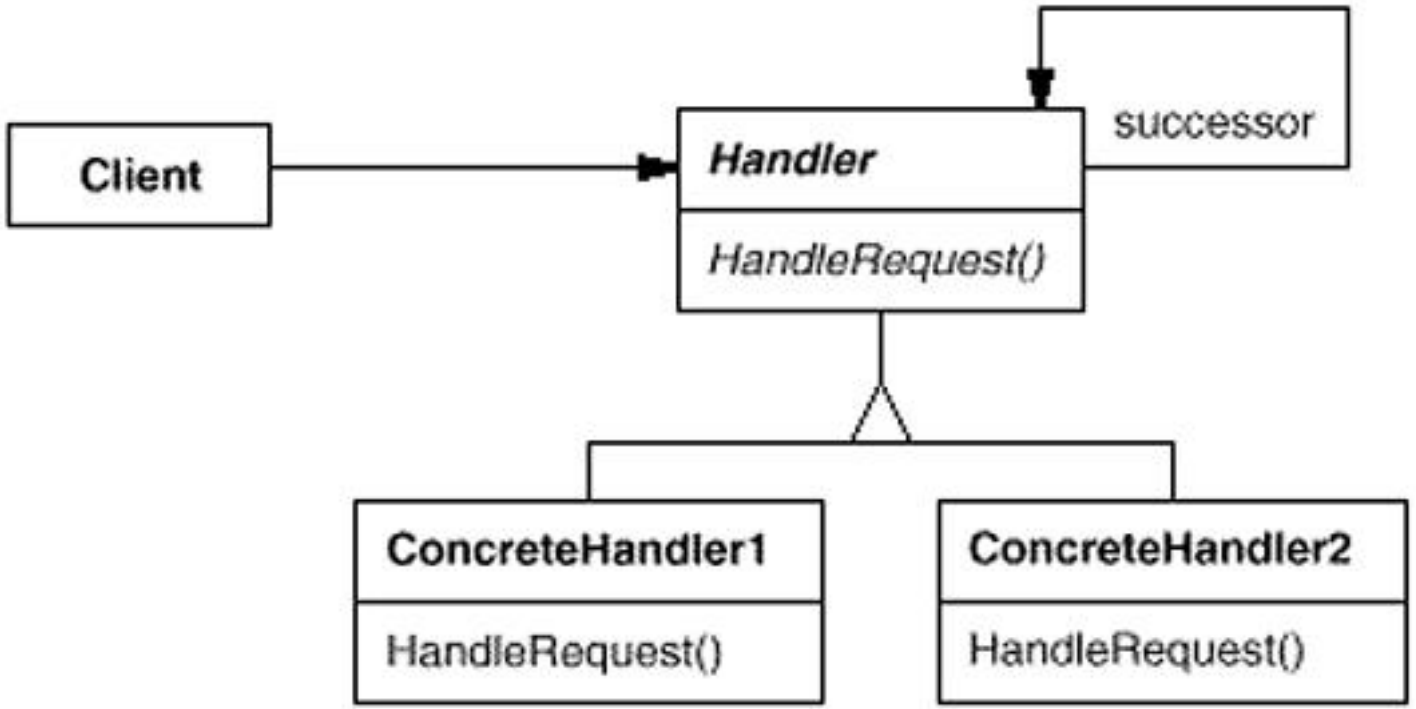
Chain of Responsibility

- Intent
  - Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

- In 4S application
  - We mentioned that a container is composed of multiple interceptors each of which is responsible for one aspect of instance management.
  - All the interceptors are interconnected as a chain of responsibility

Chain of Responsibility

- Structure



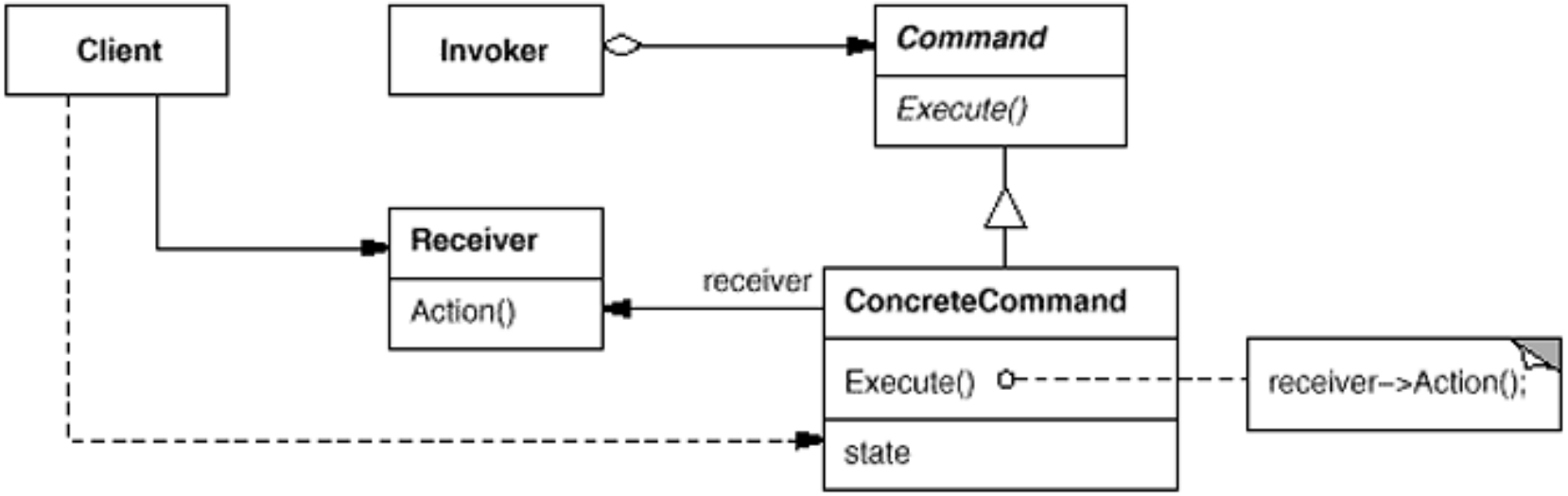
Command

- Intent
  - Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

- In 4S application
  - The developers want to encapsulate the code of processing of purchasing in order to decouple it with the one invokes it.
  - Thus, the maintainability of application will be improved.
  - Command pattern can be applied into this scenario.

Command

- Structure



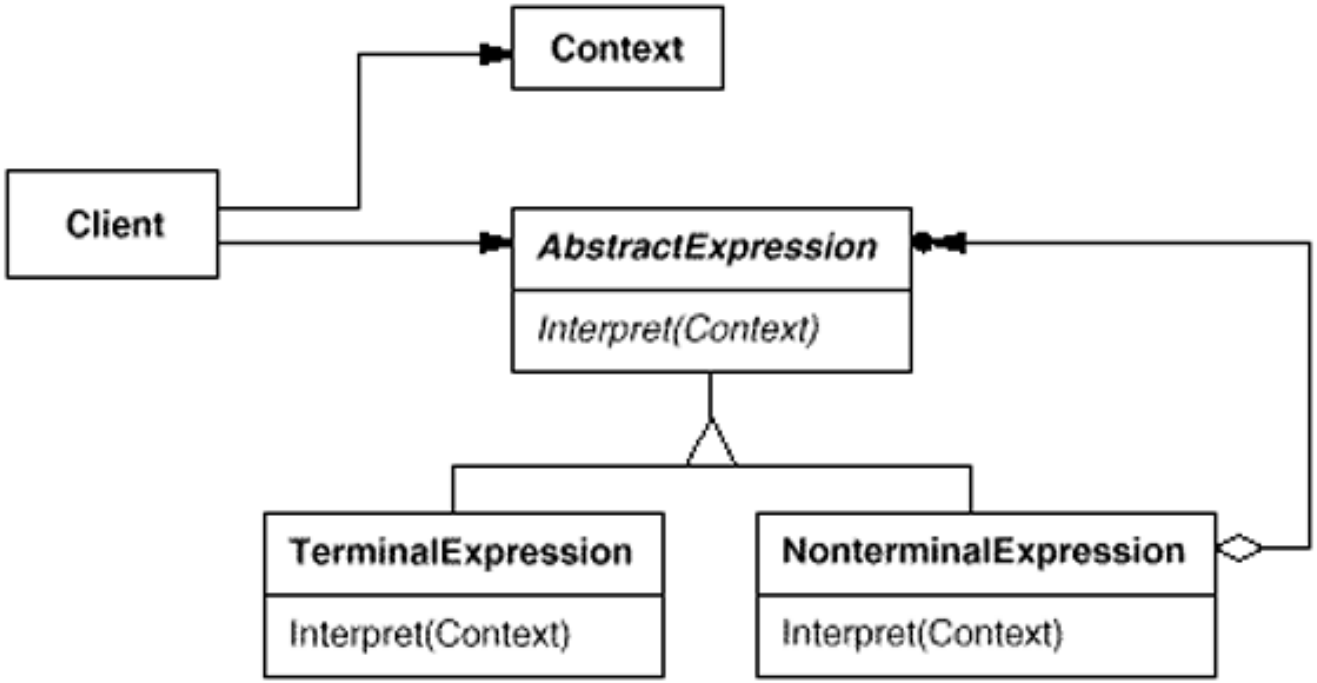
Interpreter

- Intent
  - Given a language, define a represention for its grammar along with an interpreter that uses the representation to interpret sentences in the language

- In 4S application
  - The users are allowed to customize the query
  - The syntax of the query language is simple:
    - ID:[number] R:[string] F:string L:[string]
    - ID is user's id
    - R is region
    - F is first name
    - L is last name
  - We need a interpreter to interpret the query.

Interpreter

- Structure

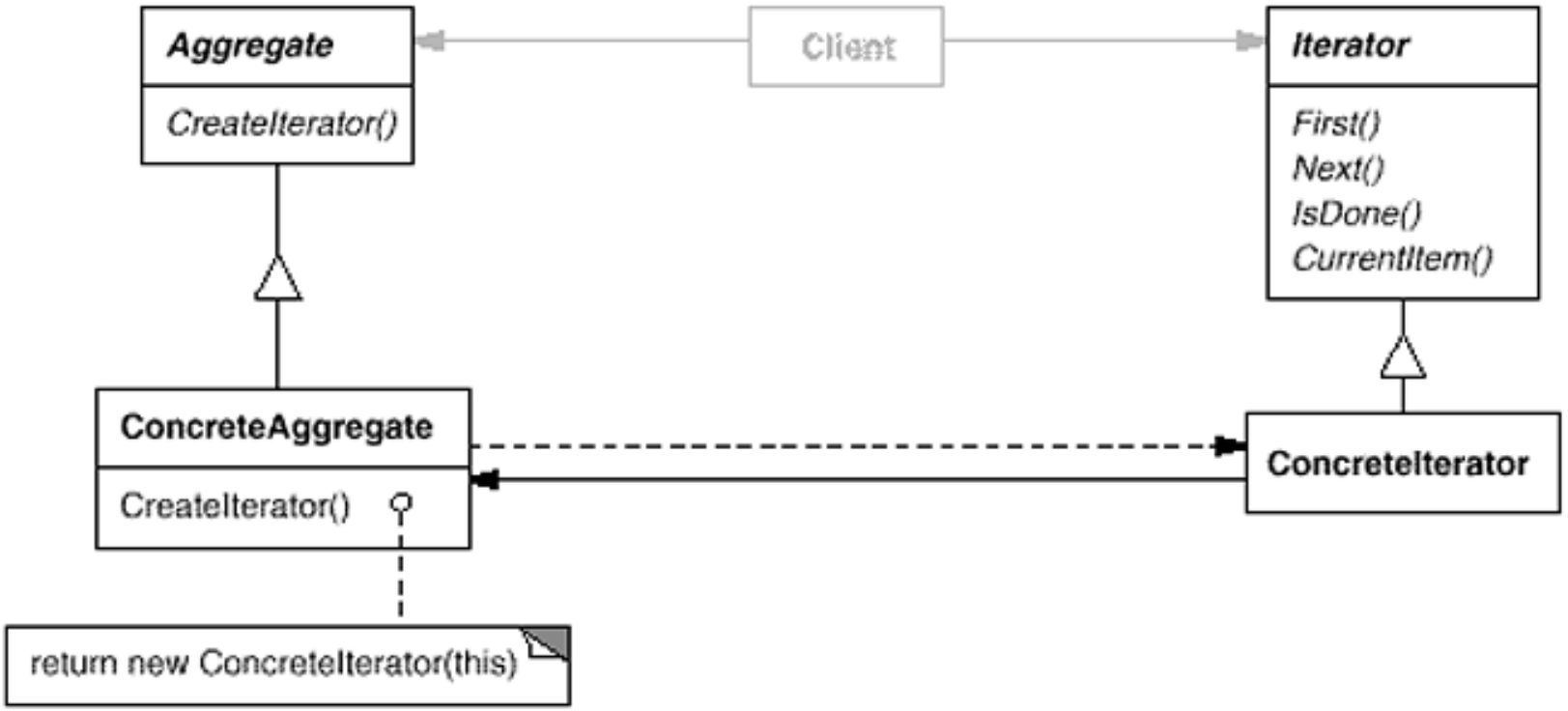


Iterator

- Intent
  - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

- In 4S application
  - Users often query the information about cars and customers.
  - Different collection types are used for such information
    - Set is for information about customers since it can remove duplicated elements.
    - HashLinkedList is for information about cars since it facilitates the query by types.
  - We need a universal way to access the information.
  - Consequently, iterator is used.

- Structure


Aggregate
CreateIterator()
Client
Iterator
First()
Next()
IsDone()
CurrentItem()
ConcreteAggregate
CreateIterator()
ConcreteIterator
return new ConcreteIterator(this)

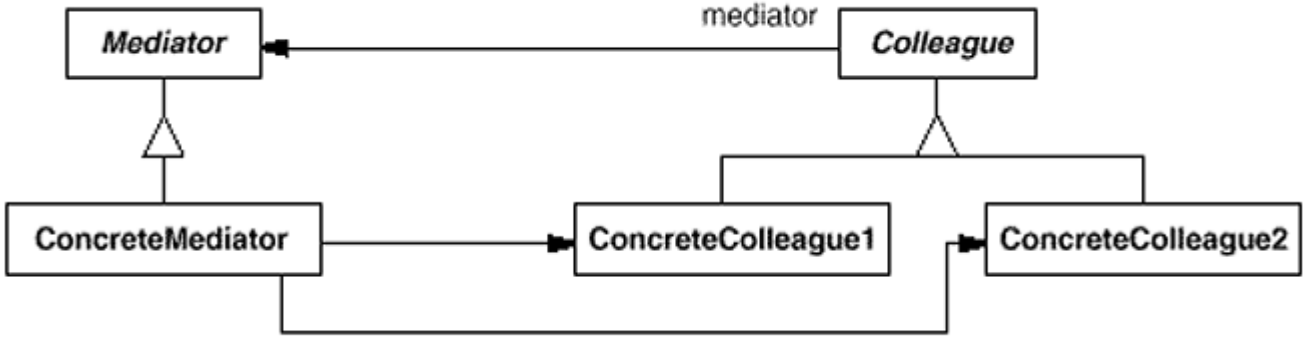
Mediator

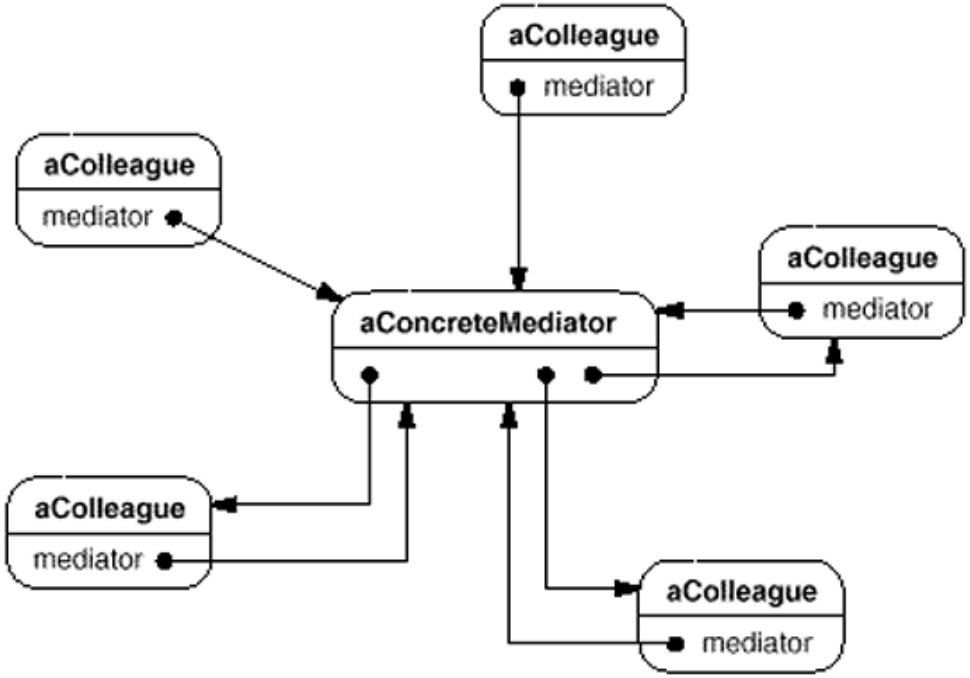
- Intent
  - Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

- In 4S application
  - In order to
    - Decouple of customers and retailers, and
    - Speed up the processing of orders.
  - The developers introduce a mediator into the application.

Mediator

- Structure



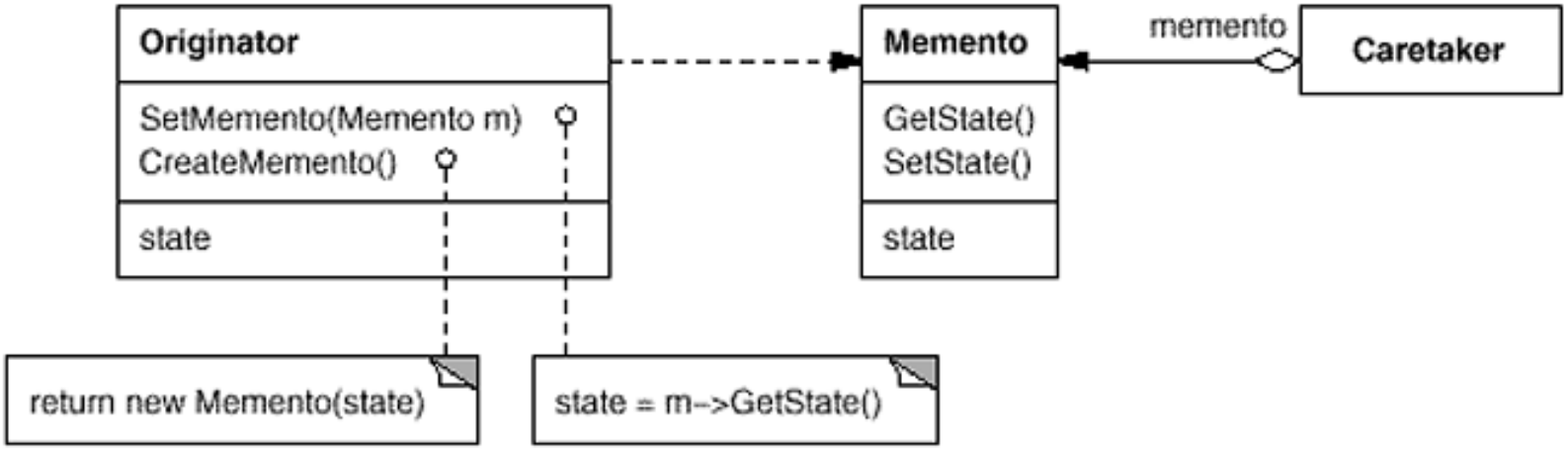


Memento

- Intent
  - Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
- In 4S application
  - We want to remove the abnormal prices of retails by restoring the previous prices.
  - Thus, memento can be used to backup the normal prices.

Memento

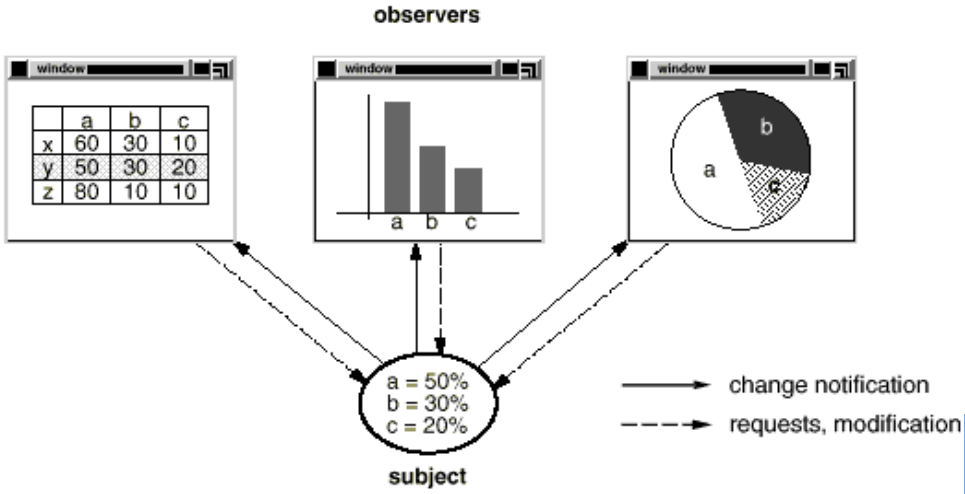
- Structure



Observer

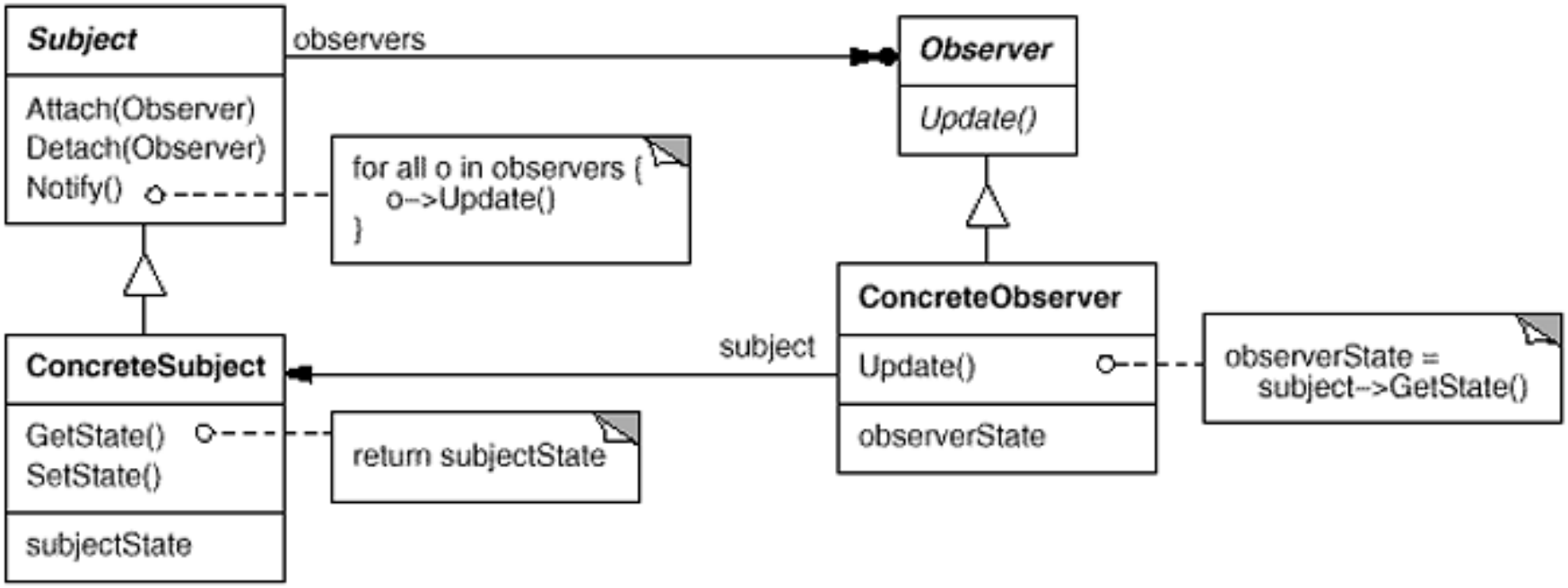
- Intent
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

- In 4S application
  - A table and two charts are presented on the client-ends to show the price of cars.
  - They need to reflect the price up-to-date.



Observer

- Structure



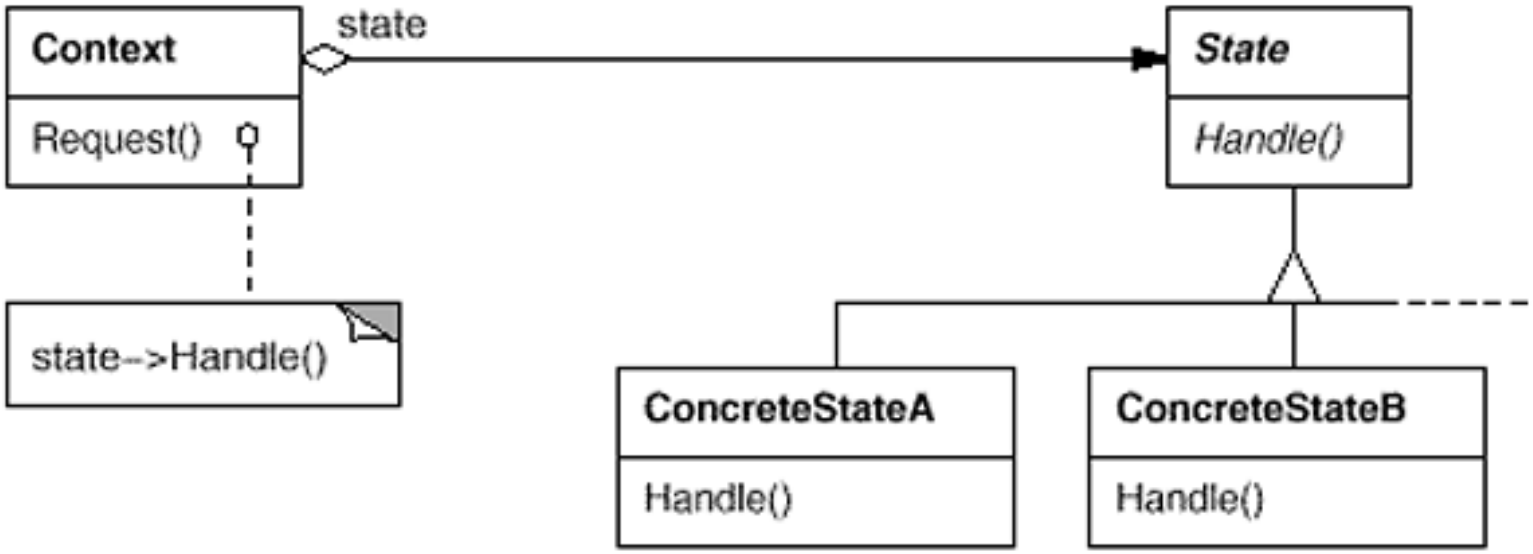
State

- Intent
  - Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

- In 4S application
  - All the clients communicate with server via a mediator.
  - If the queue of mediator is full, no request of client can be pushed into it.
  - While if the queue is not full, new request will be pushed into it.
  - Thus, for clients, the state of mediator determines the behavior of server.
  - State pattern is suitable for such scenario.

State

- Structure



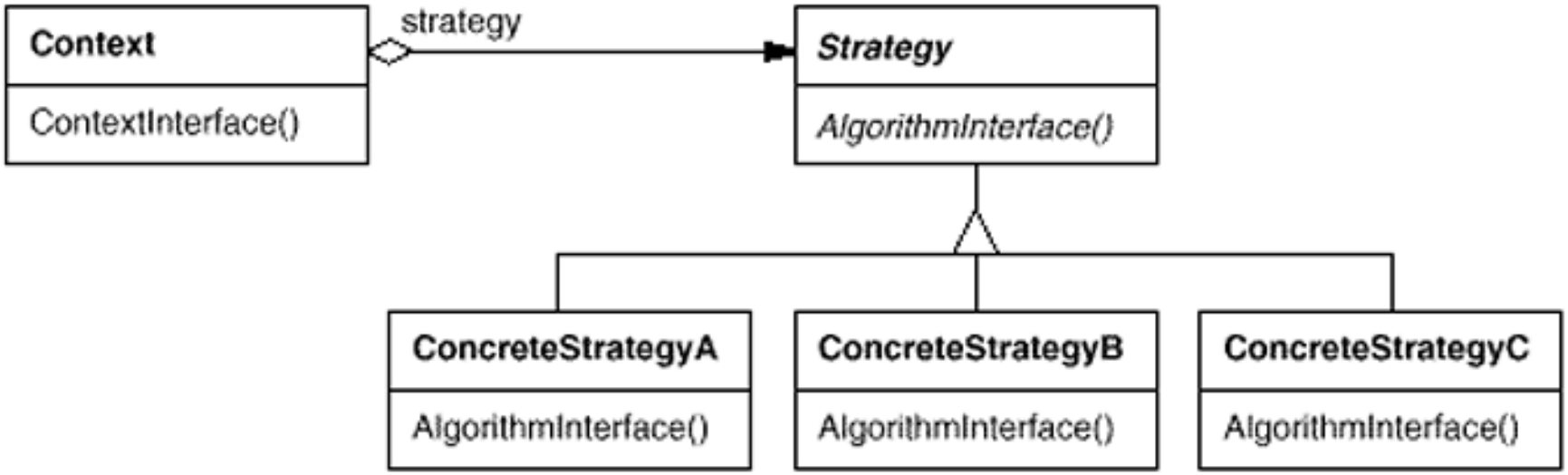
Strategy

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

- In 4S application
  - The customers are classified as Gold, Silver and Bronze levels.
  - For different customers, the processing of their order is also different.
  - To achieve this design goal, Strategy pattern is used.

Strategy

- Structure



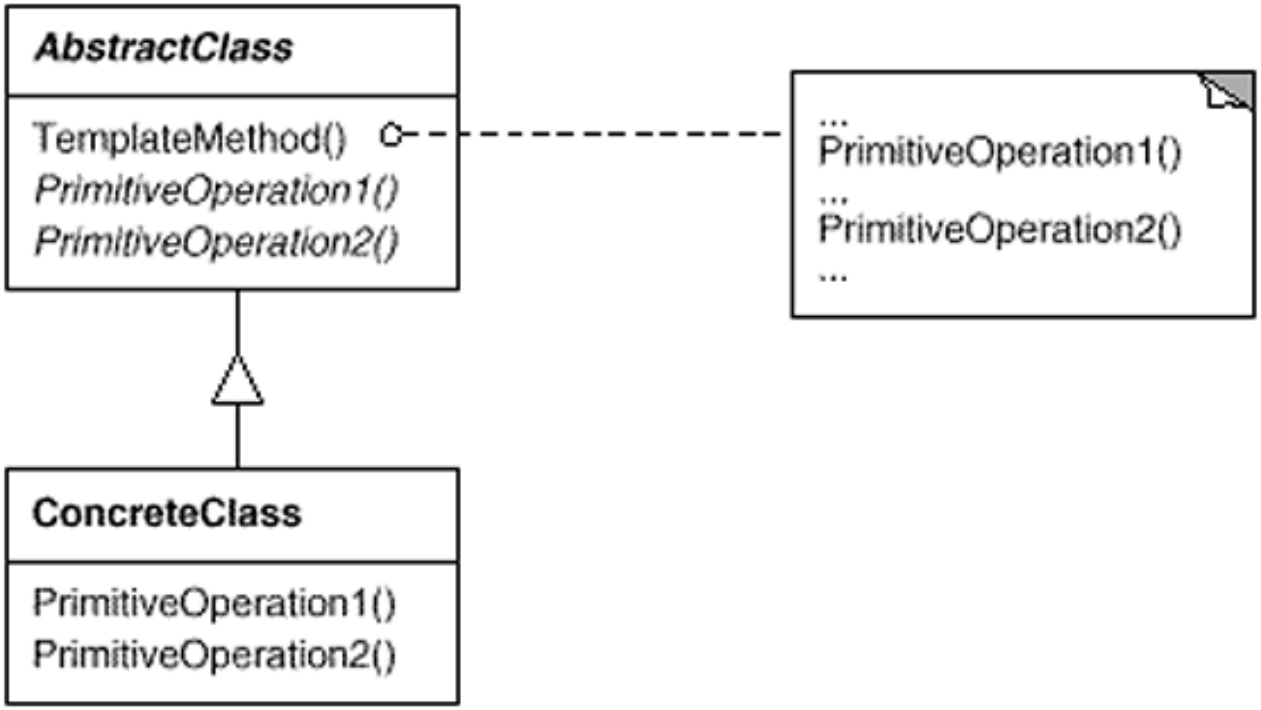
Template Method

- Intent
  - Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

- In 4S application
  - We use Strategy pattern to process the orders of different customer.
  - If all the strategies look almost same except for several steps, we can use Template Method pattern to establish the processing flow.
  - Thus, the specific implementation of a strategy just needs to implement such steps but not the whole flow.

Template Method

- Structure



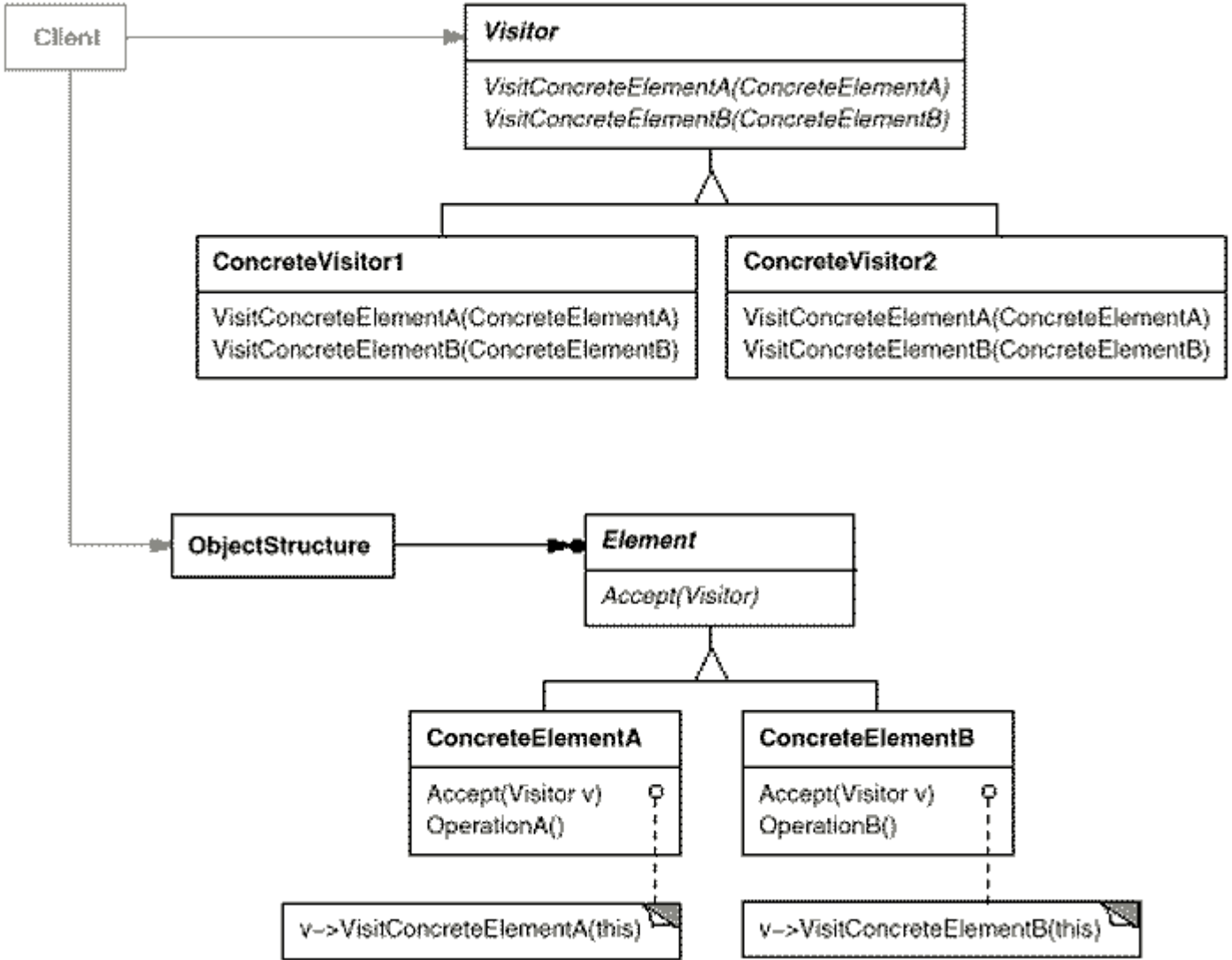
Visitor

- Intent
  - Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

- In 4S application
  - Since the customers are classified into Gold, Silver and Bronze levels. The customer of different level can view different information of cars.
    - For example, the Gold customers can see the 3D view of car, while the Silver customers can see the 2D view of car, and Bronze customers can see the thumb view of car.
  - Visitor Pattern can be applied into this case.

Visitor

- Structure



Project

- Requirements
  - To address what behavioral patterns are applied in your project, including
  - You need to describe how to combine necessary patterns in your website.

References

1. Design Patterns, elements of reusable object-oriented software
   — By Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

Thank You!