互联网应用开发技术

Web Application Development

第20课
设计模式 – 结构型模式

Episode Twenty

Structural Design Patterns

陈昊鹏

chen-hp@sjtu.edu.cn

Structural Patterns

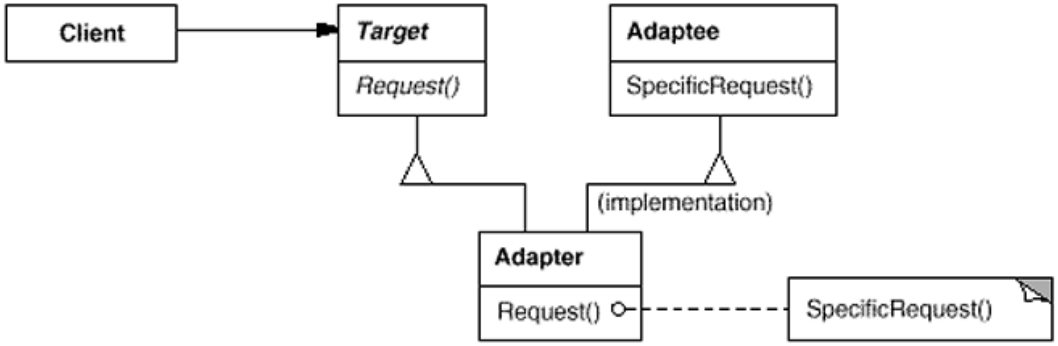
- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

Adapter

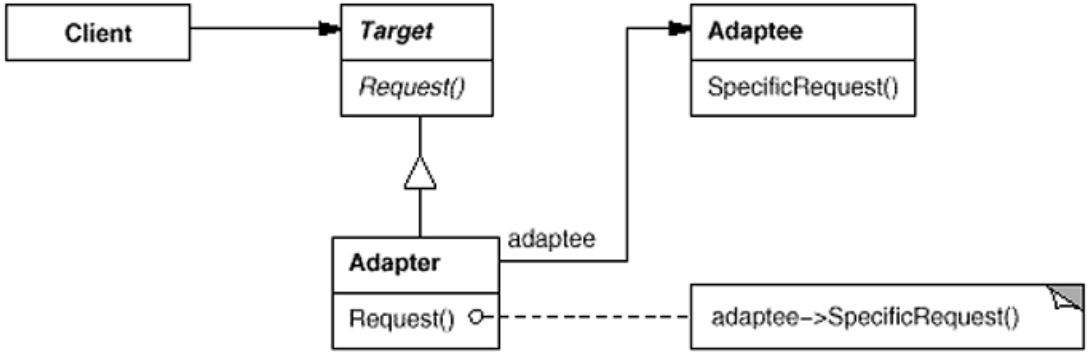
- Intent
  - Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Adapter

- Structure
  - A class adapter



  - An object adapter



Adapter

- Applicability

- Use the Adapter pattern when
  - you want to use an existing class, and its interface does not match the one you need.
  - you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.
  - *(object adapter only)* you need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one.

Adapter

- Consequences

- Class and object adapters have different trade-offs.
- A class adapter
  - adapts Adaptee to Target by committing to a concrete Adapter class. As a consequence, a class adapter won't work when we want to adapt a class *and* all its subclasses.
  - lets Adapter override some of Adaptee's behavior, since Adapter is a subclass of Adaptee.
  - introduces only one object, and no additional pointer indirection is needed to get to the adaptee.
- An object adapter
  - lets a single Adapter work with many Adaptees—that is, the Adaptee itself and all of its subclasses (if any). The Adapter can also add functionality to all Adaptees at once.
  - makes it harder to override Adaptee behavior. It will require subclassing Adaptee and making Adapter refer to the subclass rather than the Adaptee itself.

Adapter

- Implementation

- Although the implementation of Adapter is usually straightforward, here are some issues to keep in mind:
  - Implementing class adapters in C++.
  - Pluggable adapters.
  - Parameterized adapters.

Adapter

TeaBag.java： //the class that adapter will make the adaptee adapt to

```
public class TeaBag {
  boolean teaBagIsSteeped;

  public TeaBag()
    { teaBagIsSteeped = false; }

  public void steepTeaInCup() {
      teaBagIsSteeped = true;
      System.out.println("tea bag is steeping in cup");
  }
}
```

Adapter

LooseLeafTea.java： //the adaptee

```
public class LooseLeafTea {
  boolean teaIsSteeped;

  public LooseLeafTea() { teaIsSteeped = false; }

  public void steepTea() {
    teaIsSteeped = true;
    System.out.println("tea is steeping");
   }
}
```

Adapter

TeaBall.java： //the adapter

```
public class TeaBall extends TeaBag {
  LooseLeafTea looseLeafTea;

  public TeaBall(LooseLeafTea looseLeafTeaIn) {
    looseLeafTea = looseLeafTeaIn;
    teaBagIsSteeped = looseLeafTea.teaIsSteeped;
  }

  public void steepTeaInCup() {
     looseLeafTea.steepTea();
     teaBagIsSteeped = true;
  }
}
```

Adapter

TeaCup.java： //the class that accepts class TeaBag in it's steepTeaBag() method, and so is being adapted for

```
public class TeaCup {
  public void steepTeaBag(TeaBag teaBag)
   { teaBag.steepTeaInCup(); }
}
```

Adapter

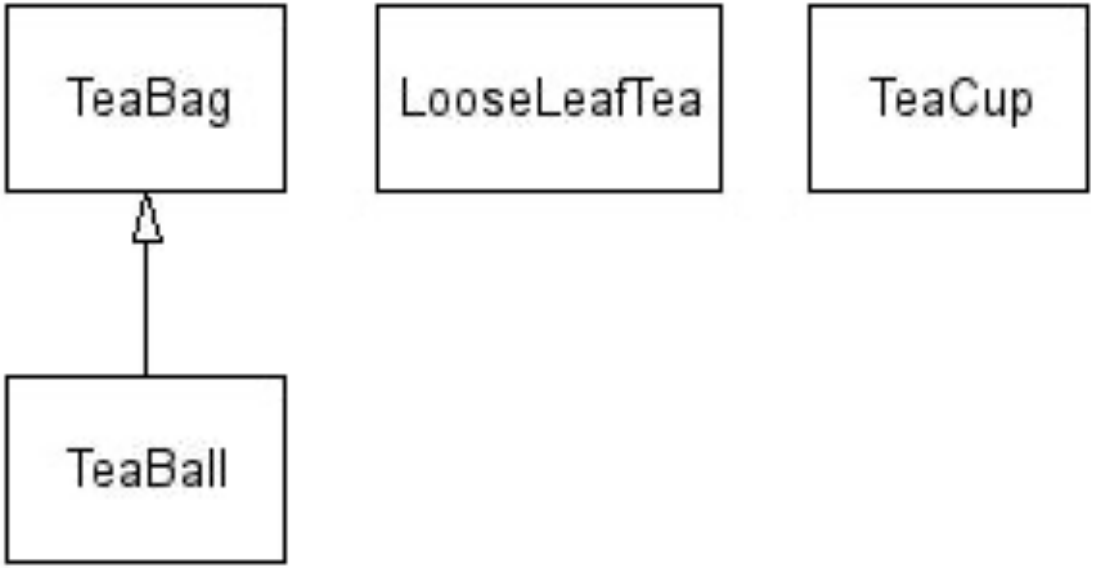
TestTeaBagAdaptation.java： //testing the adapter

```
class TestTeaBagAdaptation {
  public static void main(String[] args) {
    TeaCup teaCup = new TeaCup();

    System.out.println("Steeping tea bag");
    TeaBag teaBag = new TeaBag();
    teaCup.steepTeaBag(teaBag);

    System.out.println("Steeping loose leaf tea");
    LooseLeafTea looseLeafTea = new LooseLeafTea();
    TeaBall teaBall = new TeaBall(looseLeafTea);
    teaCup.steepTeaBag(teaBall);
  }
}
```

Adapter

Test Results

Steeping tea bag
tea bag is steeping in cup
Steeping loose leaf tea
tea is steeping

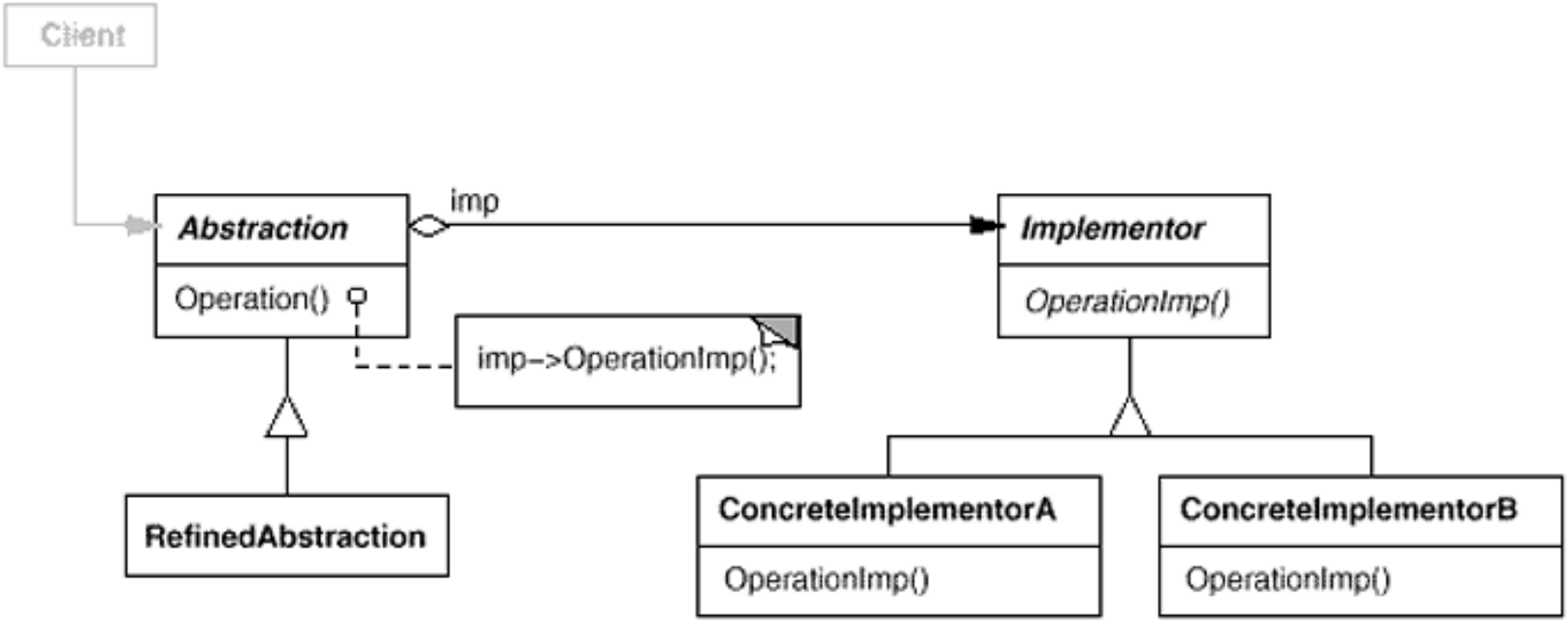


Bridge

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

Bridge

- Structure



Bridge

- Applicability
- Use the Bridge pattern when
  - you want to avoid a permanent binding between an abstraction and its implementation.
  - both the abstractions and their implementations should be extensible by subclassing.
  - changes in the implementation of an abstraction should have no impact on clients; that is, their code should not have to be recompiled.
  - indicates the need for splitting an object into two parts.
  - you want to share an implementation among multiple objects, and this fact should be hidden from the client.

Bridge

- Consequences

- The Bridge pattern has the following consequences:
  - Decoupling interface and implementation
  - Improved extensibility.
  - Hiding implementation details from clients

Bridge

- Implementation

- Consider the following implementation issues when applying the Bridge pattern:
  - Only one Implementor.
  - Creating the right Implementor object.
  - Sharing implementors.
  - Using multiple inheritance.

Bridge

Soda.java: //the Abstract Base Class

```
public abstract class Soda {
  SodaImp sodaImp;
  public void setSodaImp() {
    this.sodaImp = SodaImpSingleton.getTheSodaImp();
  }
  public SodaImp getSodaImp() {return this.sodaImp;}
  public abstract void pourSoda();
}
```

Bridge

MediumSoda.java: //one of two classes extending the Abstract

```
public class MediumSoda extends Soda {
  public MediumSoda() {setSodaImp();}
  public void pourSoda() {
    SodaImp sodaImp = this.getSodaImp();
    for (int i = 0; i < 2; i++) {
      System.out.print("...glug...");
      sodaImp.pourSodaImp();
    }
    System.out.println(" ");
  }
}
```

Bridge

SuperSizeSoda.java: //two of two classes extending the Abstract

```
public class SuperSizeSoda extends Soda {
  public SuperSizeSoda() {setSodaImp();}
  public void pourSoda() {
    SodaImp sodaImp = this.getSodaImp();
    for (int i = 0; i < 5; i++) {
      System.out.print("...glug...");
      sodaImp.pourSodaImp();
    }
    System.out.println(" ");
  }
}
```

Bridge

SodaImp.java: //the Implementation Base Class

```
public abstract class SodaImp {
    public abstract void pourSodaImp();
}
```

CherrySodaImp.java
//one of three classes extending the Implementation Base Class

```
public class CherrySodaImp extends SodaImp {
    CherrySodaImp() {}
    public void pourSodaImp() {
        System.out.println("Yummy Cherry Soda!");
    }
 }
```

Bridge

GrapeSodaImp.java

```
    //two of three classes extending the Implementation Base Class
public class GrapeSodaImp extends SodaImp {
   GrapeSodaImp() {}
   public void pourSodaImp() {
      System.out.println("Delicious Grape Soda!");
   }
}
```

OrangeSodaImp.java

```
    //three of three classes extending the Implementation Base Class
public class OrangeSodaImp extends SodaImp {
   OrangeSodaImp() {}
   public void pourSodaImp() {
      System.out.println("Citrusy Orange Soda!");
   }
}
```

Bridge

SodaImpSingleton.java

```
    //a Singleton to hold the current SodaImp

public class SodaImpSingleton {
  private static SodaImp sodaImp;
  public SodaImpSingleton(SodaImp sodaImpIn)
      {this.sodaImp = sodaImpIn;}
  public static SodaImp getTheSodaImp()
      { return sodaImp; }
}
```

Bridge

TestBridge.java: //testing the Bridge

```
class TestBridge {
  public static void testCherryPlatform() {
    SodaImpSingleton sodaImpSingleton = new SodaImpSingleton(new CherrySodaImp());
    System.out.println("testing medium soda on the cherry platform");
    MediumSoda mediumSoda = new MediumSoda();
    mediumSoda.pourSoda();
    System.out.println("testing super size soda on the cherry platform");
    SuperSizeSoda superSizeSoda = new SuperSizeSoda();
    superSizeSoda.pourSoda();
  }

  public static void testGrapePlatform() {
    SodaImpSingleton sodaImpSingleton = new SodaImpSingleton(new GrapeSodaImp());
    System.out.println("testing medium soda on the grape platform");
    MediumSoda mediumSoda = new MediumSoda();
    mediumSoda.pourSoda();
    System.out.println("testing super size soda on the grape platform");
    SuperSizeSoda superSizeSoda = new SuperSizeSoda();
    superSizeSoda.pourSoda();
  }
```

Bridge


```
  public static void testOrangePlatform() {
    SodaImpSingleton sodaImpSingleton = new SodaImpSingleton(new
     OrangeSodaImp());
    System.out.println("testing medium soda on the orange platform");
    MediumSoda mediumSoda = new MediumSoda();
    mediumSoda.pourSoda();
    System.out.println("testing super size soda on the orange platform");
    SuperSizeSoda superSizeSoda = new SuperSizeSoda();
    superSizeSoda.pourSoda();
  }

  public static void main(String[] args) {
     testCherryPlatform();
     testGrapePlatform();
     testOrangePlatform();
  }
}
```

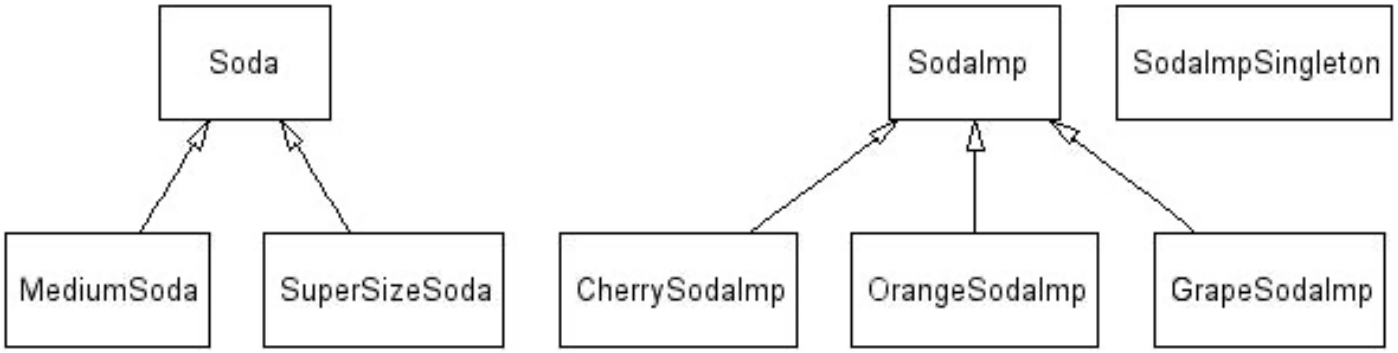
Bridge

Test Results
testing medium soda on the cherry platform
...glug...Yummy Cherry Soda!
...glug...Yummy Cherry Soda!
testing super size soda on the cherry platform
...glug...Yummy Cherry Soda!
...glug...Yummy Cherry Soda!
...glug...Yummy Cherry Soda!
...glug...Yummy Cherry Soda!
...glug...Yummy Cherry Soda!
testing medium soda on the grape platform
...glug...Delicious Grape Soda!
...glug...Delicious Grape Soda!
testing super size soda on the grape platform
...glug...Delicious Grape Soda!
...glug...Delicious Grape Soda!
...glug...Delicious Grape Soda!
...glug...Delicious Grape Soda!
...glug...Delicious Grape Soda!

Bridge

testing medium soda on the orange platform
...glug...Citrusy Orange Soda!
...glug...Citrusy Orange Soda!
testing super size soda on the orange platform
...glug...Citrusy Orange Soda!
...glug...Citrusy Orange Soda!
...glug...Citrusy Orange Soda!
...glug...Citrusy Orange Soda!
...glug...Citrusy Orange Soda!

Bridge

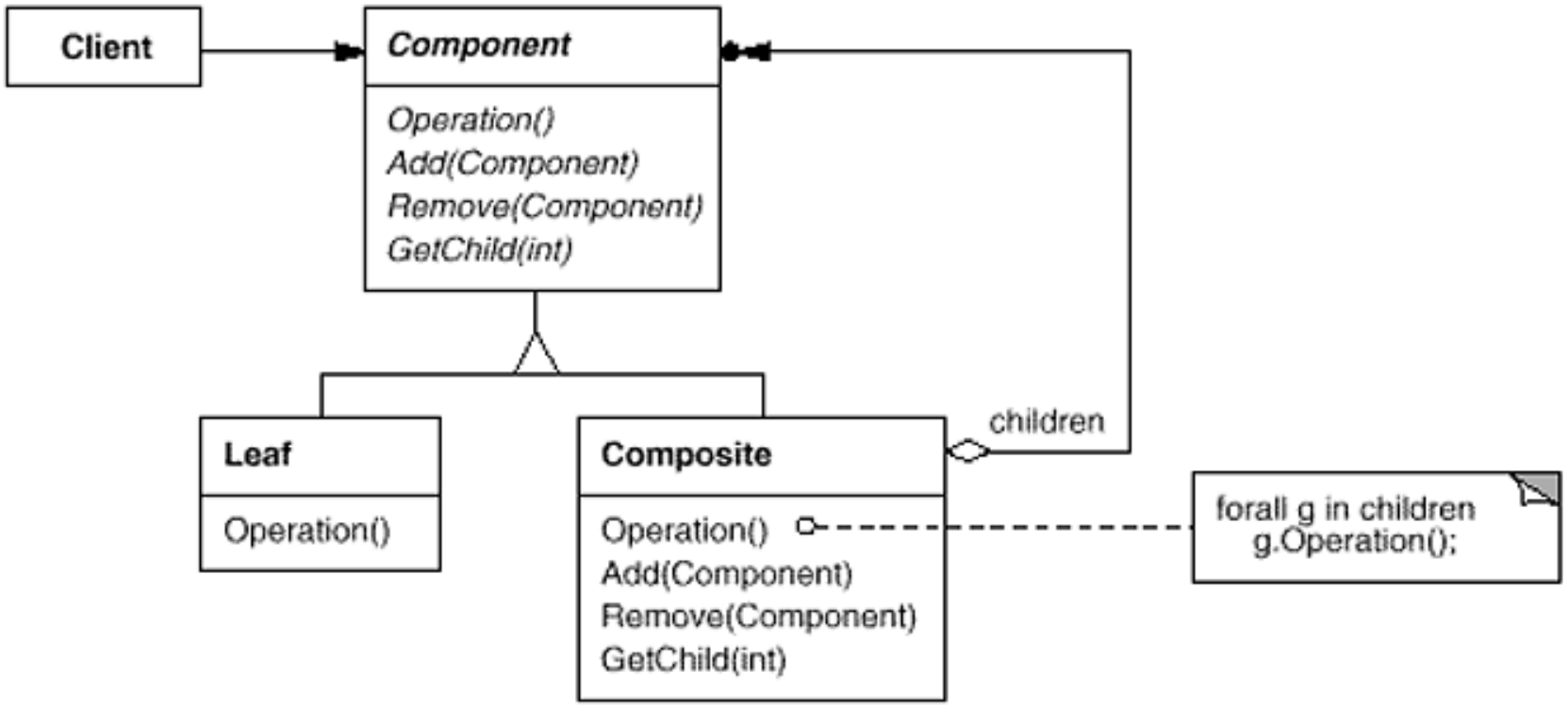

Soda
MediumSoda
SuperSizeSoda
SodaImp
SodaImpSingleton
CherrySodaImp
OrangeSodaImp
GrapeSodaImp

Composite

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

Composite

- Structure



Composite

- Applicability

- Use the Composite pattern when
  - you want to represent part-whole hierarchies of objects.
  - you want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

Composite

- Consequences

- The Composite pattern
  - defines class hierarchies consisting of primitive objects and composite objects.
  - makes the client simple.
  - makes it easier to add new kinds of components.
  - can make your design overly general.

Composite

- Implementation

- There are many issues to consider when implementing the Composite pattern:
  - Explicit parent references.
  - Sharing components.
  - Maximizing the Component interface.
  - Declaring the child management operations.
  - Should Component implement a list of Components?
  - Child ordering.
  - Caching to improve performance.
  - Who should delete components?
  - What's the best data structure for storing components?

Composite

- TeaBags.java //the abstract base class for the composite

```
import java.util.LinkedList;
import java.util.ListIterator;
public abstract class TeaBags {
 LinkedList teaBagList;
 TeaBags parent;
 String name;

 public abstract int countTeaBags();

 public abstract boolean add(TeaBags teaBagsToAdd);
 public abstract boolean remove(TeaBags teaBagsToRemove);
 public abstract ListIterator createListIterator();

 public void setParent(TeaBags parentIn) {parent = parentIn;}
 public TeaBags getParent() {return parent;}

 public void setName(String nameIn) {name = nameIn;}
 public String getName() {return name;}
}
```

Composite

- OneTeaBag.java
  //one composite extension - the "leaf“

```
import java.util.ListIterator;
public class OneTeaBag extends TeaBags
{
 public OneTeaBag(String nameIn) {this.setName(nameIn);}

 public int countTeaBags() {return 1;}

 public boolean add(TeaBags teaBagsToAdd) {return false;}
 public boolean remove(TeaBags teaBagsToRemove) {return false;}
 public ListIterator createListIterator() {return null;}
}
```

Composite

- TinOfTeaBags.java //one composite extension - the "node"

```
import java.util.LinkedList;
import java.util.ListIterator;
public class TinOfTeaBags extends TeaBags
{
  public TinOfTeaBags(String nameIn)
 {
    teaBagList = new LinkedList();
    this.setName(nameIn);
  }

  public int countTeaBags() {
    int totalTeaBags = 0;
    ListIterator listIterator = this.createListIterator();
    TeaBags tempTeaBags;
    while (listIterator.hasNext())
    {
      tempTeaBags = (TeaBags)listIterator.next();
      totalTeaBags += tempTeaBags.countTeaBags();
    }
   return totalTeaBags;
 }
```

Composite


```
public boolean add(TeaBags teaBagsToAdd)
{
  teaBagsToAdd.setParent(this);
  return teaBagList.add(teaBagsToAdd);
}

public boolean remove(TeaBags teaBagsToRemove)
{
  ListIterator listIterator = this.createListIterator();
  TeaBags tempTeaBags;
  while (listIterator.hasNext())
  {
   tempTeaBags = (TeaBags)listIterator.next();
   if (tempTeaBags == teaBagsToRemove)
   {
     listIterator.remove();
     return true;
   }
  }
  return false;
 }
```

Composite


```
public ListIterator createListIterator()
{
  ListIterator listIterator = teaBagList.listIterator();
  return listIterator;
 }
}
```

Composite

- TestTeaBagComposite.java //testing the composite

```
class TestTeaBagsComposite
{
   public static void main(String[] args)
   {
      System.out.println("Creating tinOfTeaBags");
      TeaBags tinOfTeaBags = new TinOfTeaBags("tin of tea bags");
      System.out.println("The tinOfTeaBags has " +
                              tinOfTeaBags.countTeaBags() + " tea bags in it.");

      System.out.println(" ");

      System.out.println("Creating teaBag1");
      TeaBags teaBag1 = new OneTeaBag("tea bag 1");
      System.out.println("The teaBag1 has " +
                              teaBag1.countTeaBags() + " tea bags in it.");

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

Composite


```
System.out.println("Creating teaBag2");
TeaBags teaBag2 = new OneTeaBag("tea bag 2");
System.out.println("The teaBag2 has " + teaBag2.countTeaBags() + " tea bags in it.");

System.out.println(" ");

System.out.println("Putting teaBag1 and teaBag2 in tinOfTeaBags");
if (tinOfTeaBags.add(teaBag1)) {
  System.out.println("teaBag1 added successfully to tinOfTeaBags");
} else {
  System.out.println("teaBag1 not added successfully tinOfTeaBags");
}
if (tinOfTeaBags.add(teaBag2)) {
  System.out.println("teaBag2 added successfully to tinOfTeaBags");
} else {
  System.out.println("teaBag2 not added successfully tinOfTeaBags");
}
System.out.println("The tinOfTeaBags now has " + tinOfTeaBags.countTeaBags() + " tea bags in it.");

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

Composite


```
System.out.println("Creating smallTinOfTeaBags");
TeaBags smallTinOfTeaBags = new TinOfTeaBags("small tin of tea bags");
System.out.println("The smallTinOfTeaBags has " + smallTinOfTeaBags.countTeaBags() + " tea
    bags in it.");
System.out.println("Creating teaBag3");
TeaBags teaBag3 = new OneTeaBag("tea bag 3");
System.out.println("The teaBag3 has " + teaBag3.countTeaBags() + " tea bags in it.");

System.out.println("Putting teaBag3 in smallTinOfTeaBags");
if (smallTinOfTeaBags.add(teaBag3)) {
  System.out.println("teaBag3 added successfully to smallTinOfTeaBags");
} else {
  System.out.println("teaBag3 not added successfully to smallTinOfTeaBags");
}
System.out.println("The smallTinOfTeaBags now has " + smallTinOfTeaBags.countTeaBags() + "
    tea bags in it.");

System.out.println(" ");

System.out.println("Putting smallTinOfTeaBags in tinOfTeaBags");
```

Composite

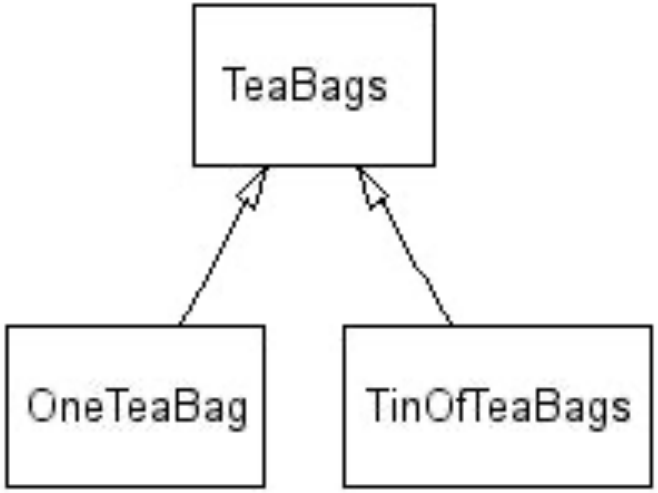

```
if (tinOfTeaBags.add(smallTinOfTeaBags)) {
   System.out.println("smallTinOfTeaBags added successfully to tinOfTeaBags");
} else {
   System.out.println("smallTinOfTeaBags not added successfully to tinOfTeaBags");
}
System.out.println("The tinOfTeaBags now has " + tinOfTeaBags.countTeaBags() + " tea bags in
    it.");

System.out.println(" ");

System.out.println("Removing teaBag2 from tinOfTeaBags");
if (tinOfTeaBags.remove(teaBag2)) {
   System.out.println("teaBag2 successfully removed from tinOfTeaBags");
} else {
   System.out.println("teaBag2 not successfully removed from tinOfTeaBags");
}
System.out.println("The tinOfTeaBags now has " + tinOfTeaBags.countTeaBags() + " tea bags in
    it.");
 }
}
```

Composite

- Test Results

```
Creating teaBag1
The teaBag1 has 1 tea bags in it.

Creating teaBag2
The teaBag2 has 1 tea bags in it.

Putting teaBag1 and teaBag2 in tinOfTeaBags
teaBag1 added successfully to tinOfTeaBags
teaBag2 added successfully to tinOfTeaBags
The tinOfTeaBags now has 2 tea bags in it.

Creating smallTinOfTeaBags
The smallTinOfTeaBags has 0 tea bags in it.
Creating teaBag3
The teaBag3 has 1 tea bags in it.
Putting teaBag3 in smallTinOfTeaBags
teaBag3 added successfully to smallTinOfTeaBags
The smallTinOfTeaBags now has 1 tea bags in it.

Putting smallTinOfTeaBags in tinOfTeaBags
smallTinOfTeaBags added successfully to tinOfTeaBags
The tinOfTeaBags now has 3 tea bags in it.
Removing teaBag2 from tinOfTeaBags
teaBag2 successfully removed from tinOfTeaBags
The tinOfTeaBags now has 2 tea bags in it.
```

Composite

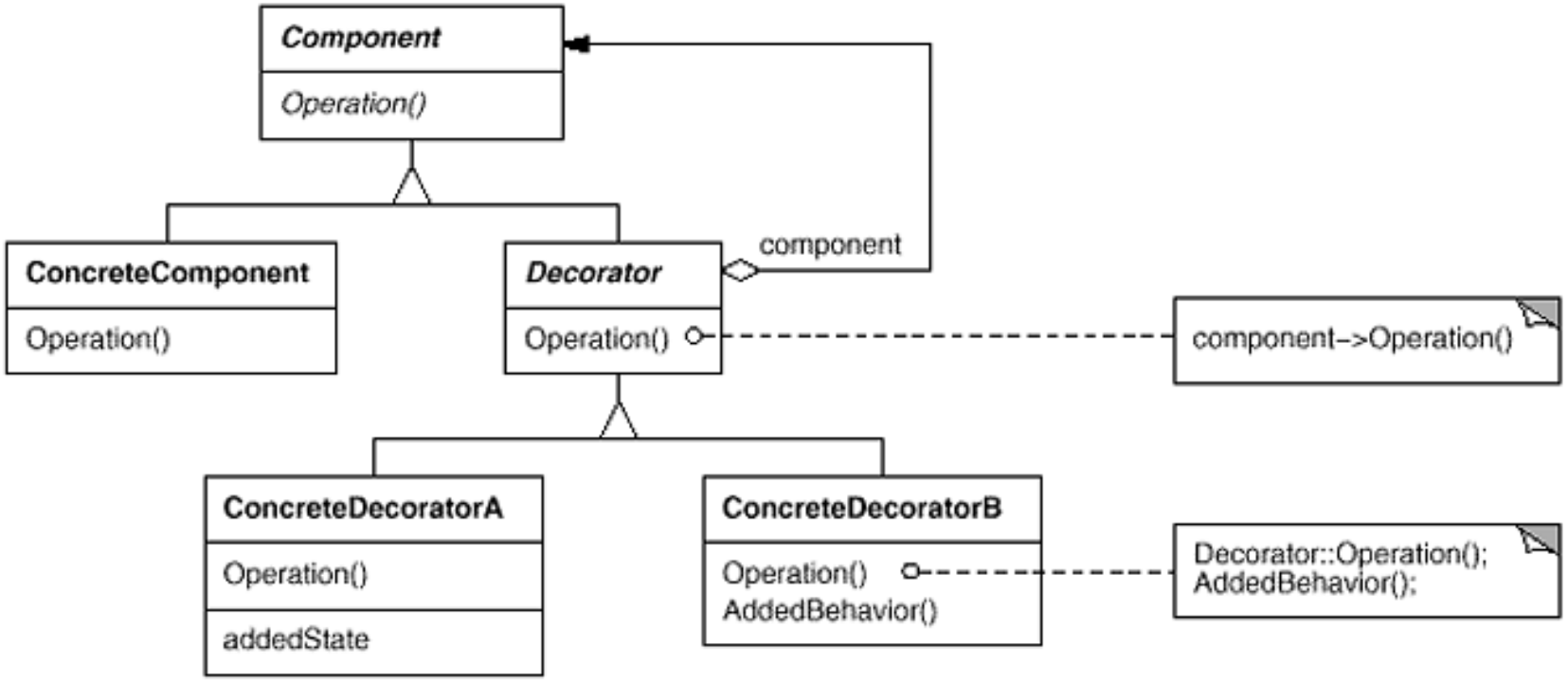


Decorator

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

Decorator

- Structure



Decorator

- Applicability

- Use Decorator
  - to add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.
  - for responsibilities that can be withdrawn.
  - when extension by subclassing is impractical.

Decorator

- Consequences

- The Decorator pattern has at least two key benefits and two liabilities:
  - More flexibility than static inheritance
  - Avoids feature-laden classes high up in the hierarchy
  - A decorator and its component aren't identical.
  - Lots of little objects.

Decorator

- Implementation

- Several issues should be considered when applying the Decorator pattern:
  - Interface conformance.
  - Omitting the abstract Decorator class.
  - Keeping Component classes lightweight.
  - Changing the skin of an object versus changing its guts.

Decorator

- Tea.java //the abstract base class

```
public abstract class Tea
{
  boolean teaIsSteeped;
  public abstract void steepTea();
}
```

- TeaLeaves.java //the decoratee

```
public class TeaLeaves extends Tea
{
  public TeaLeaves()
  {
    teaIsSteeped = false;
  }

  public void steepTea() {
    teaIsSteeped = true;
    System.out.println("tea leaves are steeping");
  }
}
```

Decorator

- ChaiDecorator.java //the decorator

```
import java.util.ArrayList;
import java.util.ListIterator;

public class ChaiDecorator extends Tea
{
    private Tea teaToMakeChai;
    private ArrayList chaiIngredients = new ArrayList();

    public ChaiDecorator(Tea teaToMakeChai)
    {
        this.addTea(teaToMakeChai);
        chaiIngredients.add("bay leaf");
        chaiIngredients.add("cinnamon stick");
        chaiIngredients.add("ginger");
        chaiIngredients.add("honey");
        chaiIngredients.add("soy milk");
        chaiIngredients.add("vanilla bean");
    }
```

Decorator


```
private void addTea(Tea teaToMakeChaiIn)
 {
   this.teaToMakeChai = teaToMakeChaiIn;
 }
 public void steepTea()
 {
   this.steepChai();
 }
 public void steepChai()
 {
   teaToMakeChai.steepTea();
   this.steepChaiIngredients();
   System.out.println("tea is steeping with chai");
 }
 public void steepChaiIngredients() {
   ListIterator listIterator = chaiIngredients.listIterator();
   while (listIterator.hasNext())
   {
      System.out.println(((String)(listIterator.next())) + " is steeping");
   }
   System.out.println("chai ingredients are steeping");
 }
}
```

Decorator

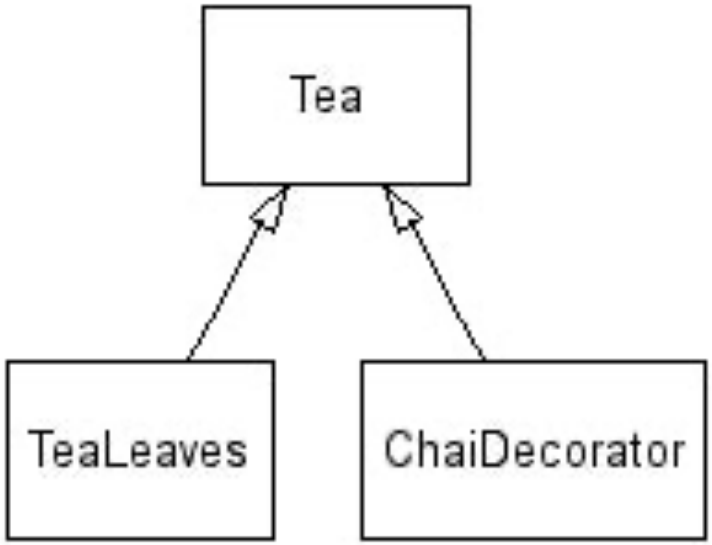
- testing the decorator

```
class TestChaiDecorator {
  public static void main(String[] args)
  {
     Tea teaLeaves = new TeaLeaves();
     Tea chaiDecorator = new ChaiDecorator(teaLeaves);
     chaiDecorator.steepTea();
   }
}
```

- Test Results

```
tea leaves are steeping
bay leaf is steeping
cinnamon stick is steeping
ginger is steeping
honey is steeping
soy milk is steeping
vanilla bean is steeping
chai ingredients are steeping
tea is steeping with chai
```

Decorator

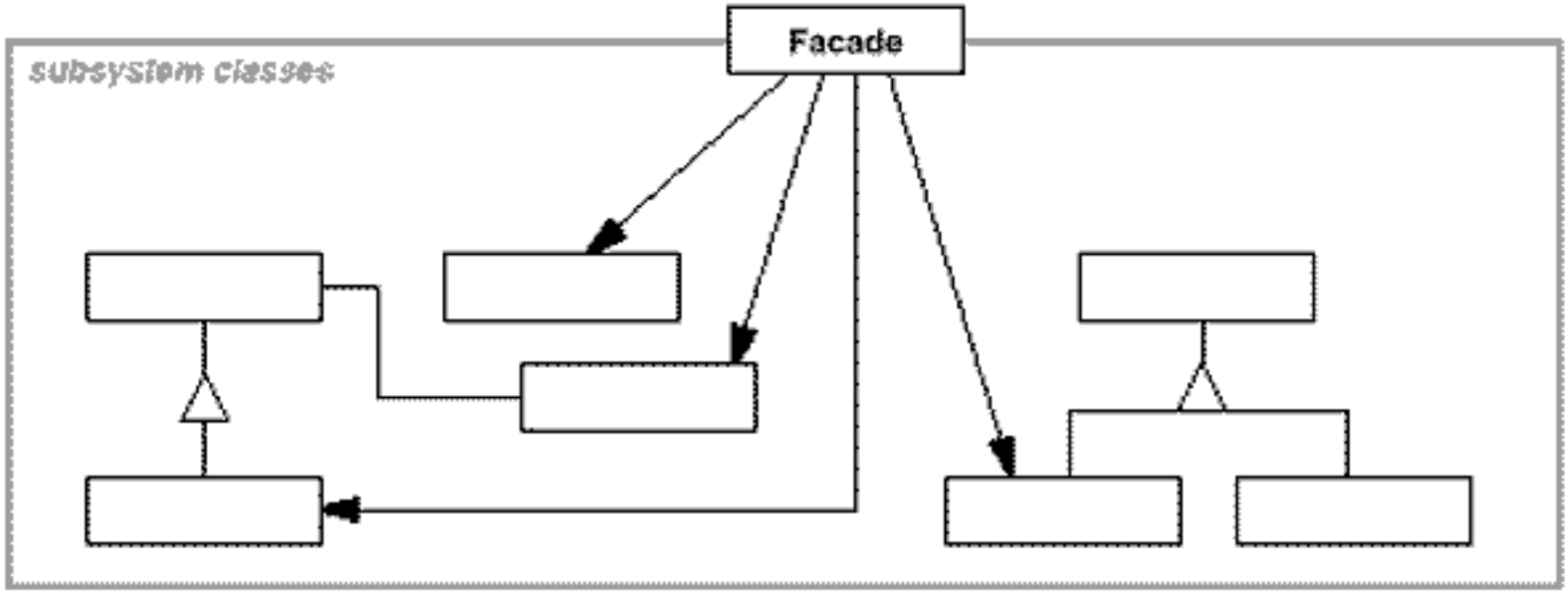


Facade

- 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.

Facade

- Structure



Facade

- Applicability

- Use the Facade pattern when
  - you want to provide a simple interface to a complex subsystem.
  - there are many dependencies between clients and the implementation classes of an abstraction. Introduce a facade to decouple the subsystem from clients and other subsystems, thereby promoting subsystem independence and portability.
  - you want to layer your subsystems. Use a facade to define an entry point to each subsystem level.

Facade

- Consequences

- The Facade pattern offers the following benefits:
  - It shields clients from subsystem components, thereby reducing the number of objects that clients deal with and making the subsystem easier to use.
  - It promotes weak coupling between the subsystem and its clients.
  - It doesn't prevent applications from using subsystem classes if they need to. Thus you can choose between ease of use and generality.

Facade

- Implementation

- Consider the following issues when implementing a facade:
  - Reducing client-subsystem coupling.
  - Public versus private subsystem classes.

Facade

- FacadeCuppaMaker.java //the Facade

```
public class FacadeCuppaMaker
{
  boolean teaBagIsSteeped;
  public FacadeCuppaMaker() {
        System.out.println("FacadeCuppaMaker ready to make you a cuppa!");
  }
  public FacadeTeaCup makeACuppa()
  {
    FacadeTeaCup cup = new FacadeTeaCup();
    FacadeTeaBag teaBag = new FacadeTeaBag();
    FacadeWater water = new FacadeWater();
    cup.addFacadeTeaBag(teaBag);
    water.boilFacadeWater();
    cup.addFacadeWater(water);
    cup.steepTeaBag();
    return cup;
   }
 }
```

Facade

- FacadeTeaCup.java // one of three classes the facade calls

```
public class FacadeTeaCup
{
    boolean teaBagIsSteeped;
    FacadeWater facadeWater;
    FacadeTeaBag facadeTeaBag;

    public FacadeTeaCup()
    {
      setTeaBagIsSteeped(false);
      System.out.println("behold the beautiful new tea cup");
     }

    public void setTeaBagIsSteeped(boolean isTeaBagSteeped) {
      teaBagIsSteeped = isTeaBagSteeped;
    }

    public boolean getTeaBagIsSteeped() {
      return teaBagIsSteeped;
    }
```

Facade


```
public void addFacadeTeaBag(FacadeTeaBag facadeTeaBagIn) {
    facadeTeaBag = facadeTeaBagIn;
    System.out.println("the tea bag is in the tea cup");
}

 public void addFacadeWater(FacadeWater facadeWaterIn) {
    facadeWater = facadeWaterIn;
    System.out.println("the water is in the tea cup");
 }

 public void steepTeaBag() {
    if ( (facadeTeaBag != null) && ((facadeWater != null) &&
        (facadeWater.getWaterIsBoiling())) ) {
      System.out.println("the tea is steeping in the cup");
      setTeaBagIsSteeped(true);
    } else {
      System.out.println("the tea is not steeping in the cup");
      setTeaBagIsSteeped(false);
     }
}
```

Facade


```
public String toString() {
    if (this.getTeaBagIsSteeped()) {
      return ("A nice cuppa tea!");
    } else {
      String tempString = new String("A cup with ");
      if (facadeWater != null) {
        if (facadeWater.getWaterIsBoiling()) {
          tempString = (tempString + "boiling water ");
        } else {
          tempString = (tempString + "cold water ");
        }
      } else {
        tempString = (tempString + "no water ");
      }
      if (facadeTeaBag != null) {
        tempString = (tempString + "and a tea bag");
      }else {
         tempString = (tempString + "and no tea bag");
       }
      return tempString;
      }
   }
}
```

Facade

- FacadeWater.java //two of three classes the facade calls

```
public class FacadeWater
 {
    boolean waterIsBoiling;
    public FacadeWater()
   {
       setWaterIsBoiling(false);
       System.out.println("behold the wonderous water");
    }
   public void boilFacadeWater() {
       setWaterIsBoiling(true);
       System.out.println("water is boiling");
   }
   public void setWaterIsBoiling(boolean isWaterBoiling) {
       waterIsBoiling = isWaterBoiling;
   }
   public boolean getWaterIsBoiling() { return waterIsBoiling; }
 }
```

Facade

- FacadeTeaBag.java

```
// three of three classes the facade calls
public class FacadeTeaBag
{
   public FacadeTeaBag()
   {
     System.out.println("behold the lovely tea bag");
    }
 }
```

- TestFacade.java

```
// testing the Facade
class TestFacade {
   public static void main(String[] args)
   {
      FacadeCuppaMaker cuppaMaker = new FacadeCuppaMaker();
      FacadeTeaCup teaCup = cuppaMaker.makeACuppa();
      System.out.println(teaCup);
    }
}
```

Facade

- Test Results

FacadeCuppaMaker ready to make you a cuppa!
behold the beautiful new tea cup
behold the lovely tea bag
behold the wonderous water
the tea bag is in the tea cup
water is boiling
the water is in the tea cup
the tea is steeping in the cup
A nice cuppa tea!

Facade

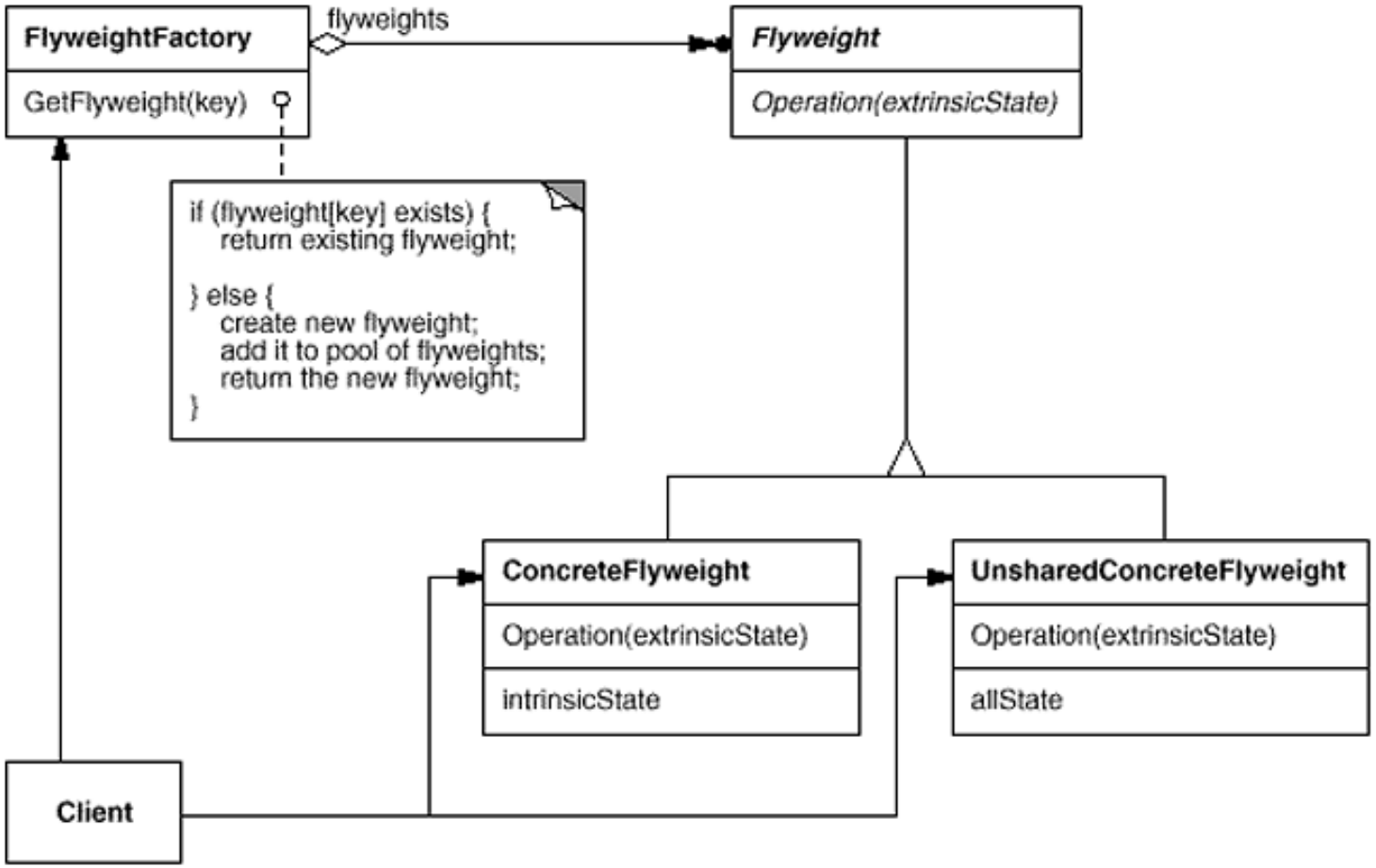

FacadeCuppaMaker
FacadeTeaCup
FacadeWater
FacadeTeaBag

Flyweight

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

Flyweight

- Structure



Flyweight

- Applicability

- Apply the Flyweight pattern when *all* of the following are true:
  - An application uses a large number of objects.
  - Storage costs are high because of the sheer quantity of objects.
  - Most object state can be made extrinsic.
  - Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed.
  - The application doesn't depend on object identity. Since flyweight objects may be shared, identity tests will return true for conceptually distinct objects.

Flyweight

- Consequences
  - Flyweights may introduce run-time costs associated with transferring, finding, and/or computing extrinsic state, especially if it was formerly stored as intrinsic state.
  - Storage savings are a function of several factors:
    - the reduction in the total number of instances that comes from sharing
    - the amount of intrinsic state per object
    - whether extrinsic state is computed or stored.
  - The more flyweights are shared, the greater the storage savings.

Flyweight

- Implementation

- Consider the following issues when implementing the Flyweight pattern:
  - Removing extrinsic state.
  - Managing shared objects.

Flyweight

TeaOrder.java: //the Flyweight

```
public abstract class TeaOrder {
    public abstract
     void serveTea(TeaOrderContext teaOrderContext);
}
```

Flyweight

TeaFlavor.java: //the Concrete Flyweight

```
public class TeaFlavor extends TeaOrder {
   String teaFlavor;
   TeaFlavor(String teaFlavor) {this.teaFlavor = teaFlavor;}
   public String getFlavor() {return this.teaFlavor;}
   public void serveTea(TeaOrderContext teaOrderContext)
       {
          System.out.println("Serving tea flavor " + teaFlavor +
                    " to table number " + teaOrderContext.getTable());
       }
}
```

Flyweight

TeaOrderContext.java: //the Context

```
public class TeaOrderContext {
   int tableNumber;
  TeaOrderContext(int tableNumber) {
    this.tableNumber = tableNumber;
   }
   public int getTable() { return this.tableNumber; }
 }
```

Flyweight


```
TeaFlavorFactory.java: // the Factory
public class TeaFlavorFactory {
   TeaFlavor[] flavors = new TeaFlavor[10];
   int teasMade = 0;
   public TeaFlavor getTeaFlavor(String flavorToGet) {
     if (teasMade > 0) {
       for (int i = 0; i < teasMade; i++) {
          if (flavorToGet.equals((flavors[i]).getFlavor()))
            { return flavors[i]; }
        }
     }
     flavors[teasMade] = new TeaFlavor(flavorToGet);
     return flavors[teasMade++];
   }
  public int getTotalTeaFlavorsMade() {return teasMade;}
}
```

Flyweight


```
TestFlyweight.java: //the Client, tests the Flyweight
class TestFlyweight {
  static TeaFlavor[] flavors = new TeaFlavor[100];
  static TeaOrderContext[] tables = new TeaOrderContext[100];
  static int ordersMade = 0;
  static TeaFlavorFactory teaFlavorFactory;

  static void takeOrders(String flavorIn, int table) {
    flavors[ordersMade] = teaFlavorFactory.getTeaFlavor(flavorIn);
    tables[ordersMade++] = new TeaOrderContext(table);
  }
  public static void main(String[] args) {
    teaFlavorFactory = new TeaFlavorFactory();
    takeOrders("chai", 2);
    takeOrders("chai", 2);
    takeOrders("camomile", 1);
```

Flyweight

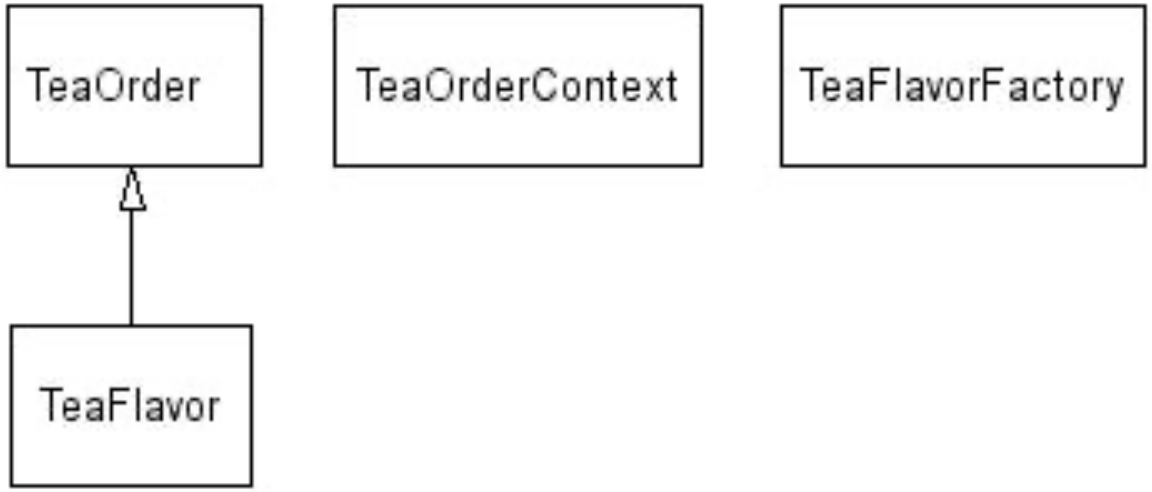

```
    takeOrders("camomile", 1);
    takeOrders("earl grey", 1);
    takeOrders("camomile", 897);
    takeOrders("chai", 97);
    takeOrders("chai", 97);
    takeOrders("camomile", 3);
    takeOrders("earl grey", 3);
    takeOrders("chai", 3);
    takeOrders("earl grey", 96);
    takeOrders("camomile", 552);
    takeOrders("chai", 121);
    takeOrders("earl grey", 121);
    for (int i = 0; i < ordersMade; i++) { flavors[i].serveTea(tables[i]); }
    System.out.println(" ");
    System.out.println("total teaFlavor objects
       made: " + teaFlavorFactory.getTotalTeaFlavorsMade());
  }
}
```

Flyweight

Test Results

```
Serving tea flavor chai to table number 2
Serving tea flavor chai to table number 2
Serving tea flavor camomile to table number 1
Serving tea flavor camomile to table number 1
Serving tea flavor earl grey to table number 1
Serving tea flavor camomile to table number 897
Serving tea flavor chai to table number 97
Serving tea flavor chai to table number 97
Serving tea flavor camomile to table number 3
Serving tea flavor earl grey to table number 3
Serving tea flavor chai to table number 3
Serving tea flavor earl grey to table number 96
Serving tea flavor camomile to table number 552
Serving tea flavor chai to table number 121
Serving tea flavor earl grey to table number 121

total teaFlavor objects made: 3
```

Flyweight

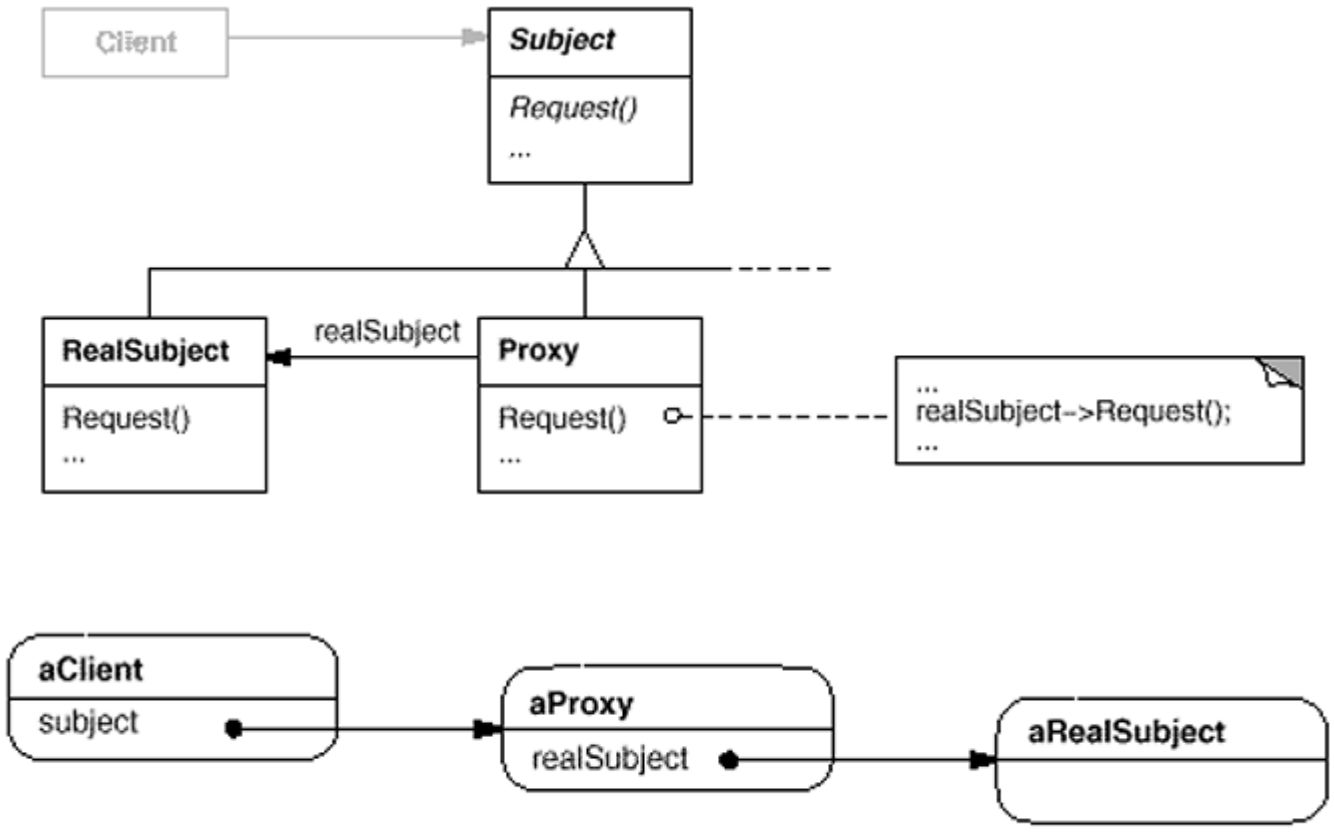


Proxy

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

Proxy

- Structure



Proxy

- Applicability

- Here are several common situations in which the Proxy pattern is applicable:
  - A remote proxy provides a local representative for an object in a different address space.
  - A virtual proxy creates expensive objects on demand.
  - A protection proxy controls access to the original object.
  - A smart reference is a replacement for a bare pointer that performs additional actions when an object is accessed.

Proxy

- Consequences

- The Proxy pattern introduces a level of indirection when accessing an object. The additional indirection has many uses, depending on the kind of proxy:
  - A remote proxy can hide the fact that an object resides in a different address space.
  - A virtual proxy can perform optimizations such as creating an object on demand.
  - Both protection proxies and smart references allow additional housekeeping tasks when an object is accessed.

Proxy

- Implementation

- The Proxy pattern can exploit the following language features:
  - Overloading the member access operator
  - Proxy doesn't always have to know the type of real subject.

Proxy

- PotOfTeaInterface.java //the Subject Interface

```
public interface PotOfTeaInterface
 // PotOfTeaInterface will insure that the "proxy"
 // has the same methods as it's "real subject"
{
    public void pourTea();
}
```

- PotOfTeaProxy.java //the Proxy

```
public class PotOfTeaProxy implements PotOfTeaInterface
{
     PotOfTea potOfTea;
     public PotOfTeaProxy() {}
     public void pourTea()
    {
        potOfTea = new PotOfTea();
        potOfTea.pourTea();
     }
 }
```

Proxy

- PotOfTea.java

```
// the Real Subject
public class PotOfTea implements PotOfTeaInterface
 {
     public PotOfTea()
     {
         System.out.println("Making a pot of tea");
      }

     public void pourTea()
     {
          System.out.println("Pouring tea");
     }
 }
```

Proxy

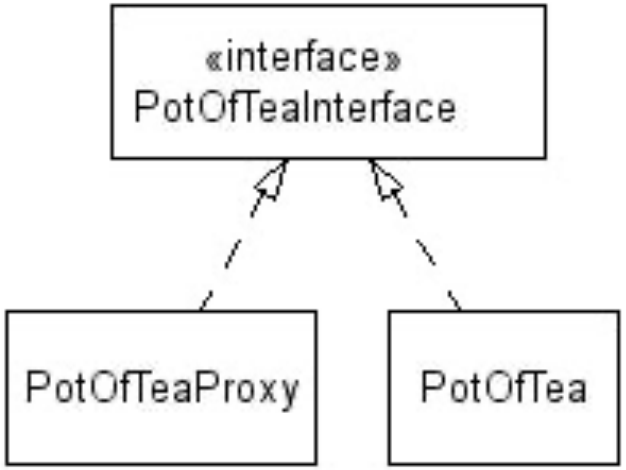
- TestProxy.java

```
// testing the Proxy
class TestProxy
{
   public static void main(String[] args)
  {
     System.out.println("TestProxy: instantiating PotOfTeaProxy");
     PotOfTeaInterface potOfTea = new PotOfTeaProxy();
     System.out.println(" ");
     System.out.println("TestProxy: pouring tea");
     potOfTea.pourTea();
   }
}
```

- Test Results

```
TestProxy: instantiating PotOfTeaProxy
TestProxy: pouring tea
Making a pot of tea
Pouring tea
```

Proxy



- *Web*开发技术
- *Web Application Development*

Thank You!