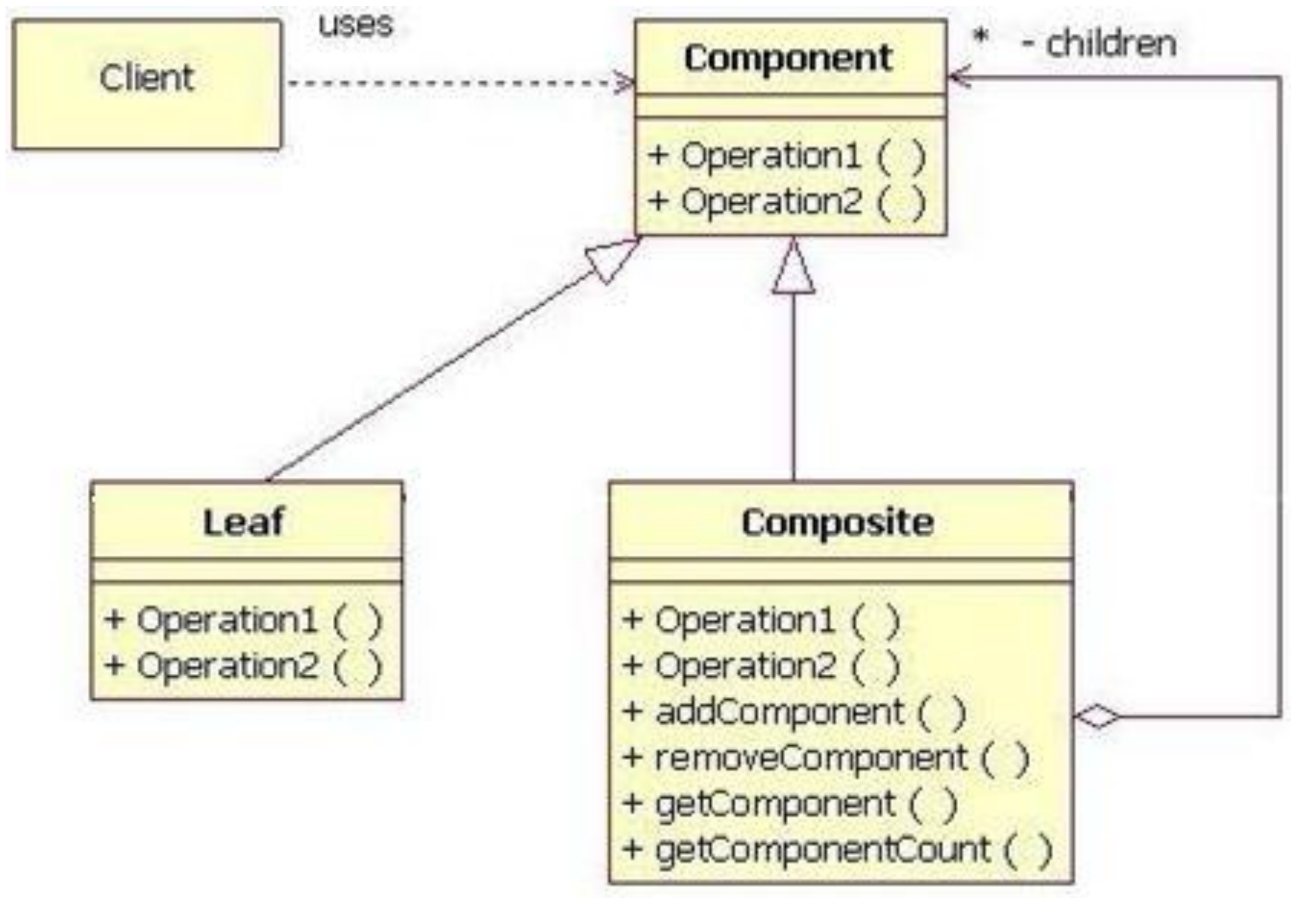
BITS Pilani
Pilani Campus

DESIGN PATTERNS (Lecture-12)

Composite Pattern

- **Composite objects: objects that contain other objects; for example, a drawing may be composed of** graphic primitives, such as lines, circles, rectangles, text, and so on. Developers need the Composite pattern because we often must manipulate composites exactly the same way we manipulate primitive objects. For example, graphic primitives such as lines or text must be drawn, moved, and resized. But we also want to perform the same operation on composites, such as drawings, that are composed of those primitives. Ideally, we'd like to perform operations on both primitive objects and composites in exactly the same manner, *without distinguishing between the two. If* we must distinguish between primitive objects and composites to perform the same operations on those two types of objects, our code would become more complex and more difficult to implement, maintain, and extend. Implementing the Composite pattern is easy. Composite classes extend a base class that represents primitive objects.

Figure-1


Client
uses
Component
+ Operation1 ( )
+ Operation2 ( )
* - children
Leaf
+ Operation1 ( )
+ Operation2 ( )
Composite
+ Operation1 ( )
+ Operation2 ( )
+ addComponent ( )
+ removeComponent ( )
+ getComponent ( )
+ getComponentCount ( )

Composite Pattern (contd.)

- **Component represents a base class (or possibly an interface) for primitive objects, and Composite** represents a composite class. For example, the **Component class might represent a base class for graphic** primitives, whereas the **Composite class might represent a Drawing class. Leaf class represents a** concrete primitive object; for example, a **Line class or a Text class. The Operation1() and Operation2() methods represent domain-specific methods implemented by both the Component** and **Composite classes.**
- The **Composite class maintains a collection of components. Typically, Composite** methods are implemented by iterating over that collection and invoking the appropriate method for each **Component in the collection. For example, a Drawing class might implement its draw() method** like this:

Composite Pattern (contd.)

```
// This method is a Composite method
public void draw() {
      // Iterate over the components
      for(int i=0; i < getComponentCount(); ++i) {
            // Obtain a reference to the component and invoke its draw method
            Component component = getComponent(i);
            component.draw();
      }
}
```

- For every method implemented in the **Component class, the Composite class implements a method** with the same signature that iterates over the composite's components, as illustrated by the **draw()** method listed above.

Composite Pattern (contd.)

- The **Composite class extends the Component class, so you can pass a composite to a method that** expects a component; for example, consider the following method:

```
// this method is implemented in a class that's unrelated to the
// Component and Composite classes
public void repaint(Component component) {
     // the component can be a composite, but since it extends
     // the Component class, this method need not
     // distinguish between components and composites
     component.draw();
}
```

- The preceding method is passed a component either a simple component or a composite then it invokes that component's **draw() method. Because the Composite class extends Component, the repaint()method need not distinguish between components and composites it simply invokes the draw() method for the component (or composite).**

Composite Pattern (contd.)

- Figure-1's Composite pattern class diagram illustrate one problem with the pattern: *you must distinguish between* ***components and composites when you reference a Component, and you must invoke a composite specific method, such as addComponent(). You typically fulfill that requirement by*** adding a method, such as ***isComposite(), to the Component class. That method returns false for*** components and is overridden in the **Composite class to return true. Additionally, you must also cast** the **Component reference to a Composite instance, like this:**

```
...
if(component.isComposite()) {
Composite composite = (Composite)component;
composite.addComponent(someComponentThatCouldBeAComposite);
}
...
```

- Notice that the **addComponent() method is passed a Component reference, which can be either a** primitive component or a composite.

Alternative Composite pattern implementation

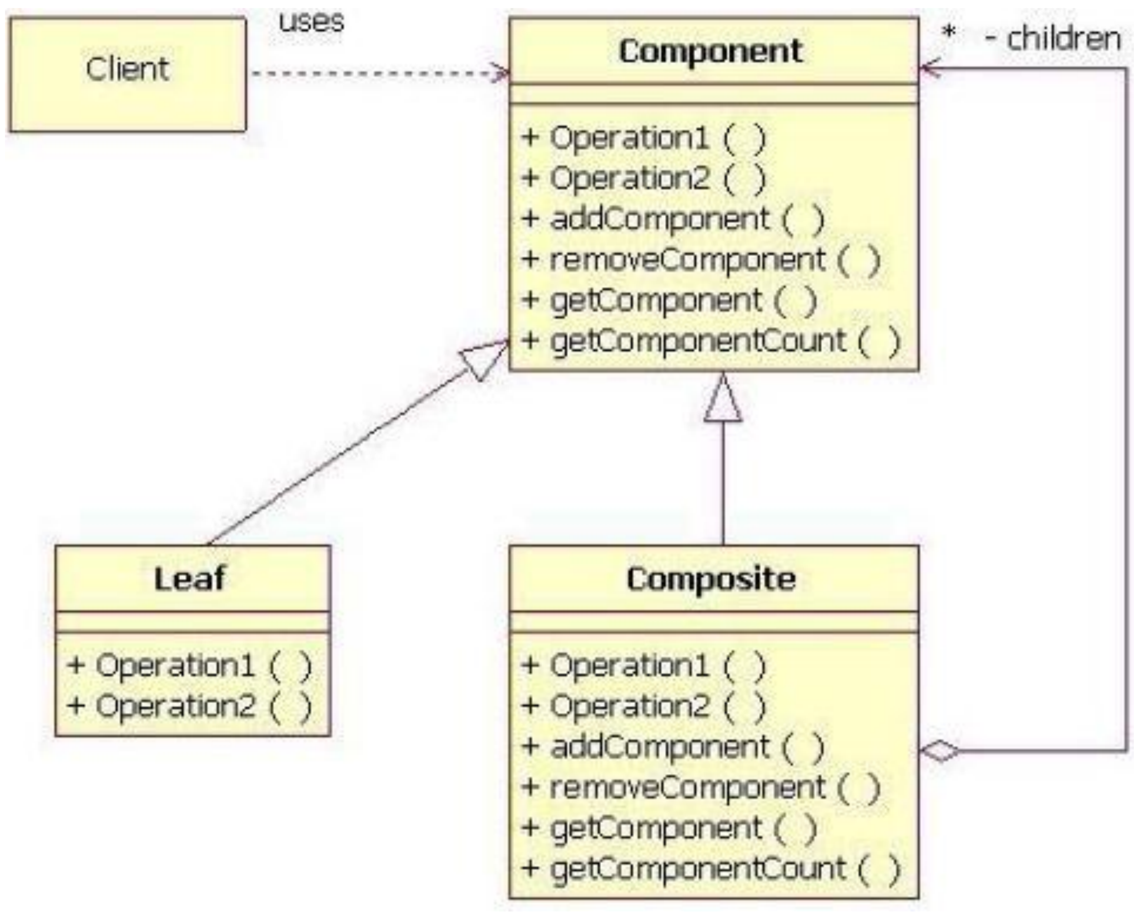


Figure-2

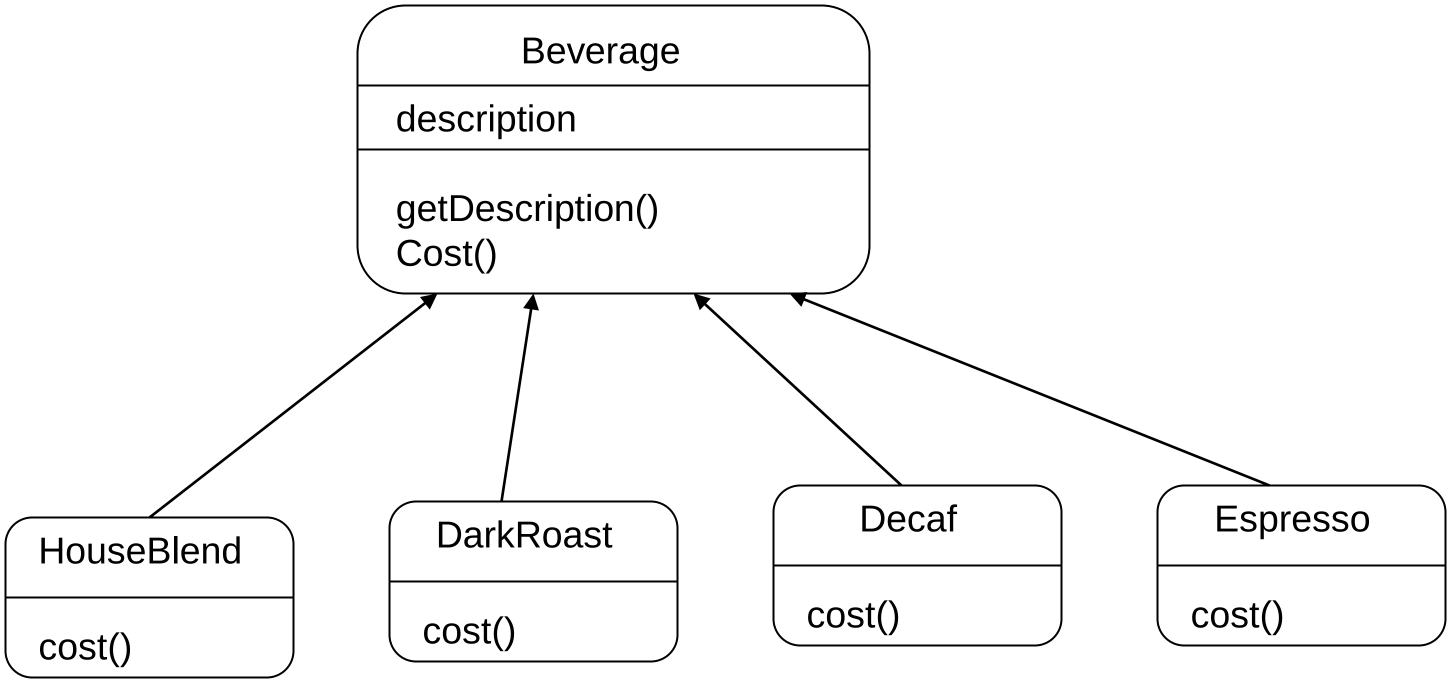
- If you implement Figure-2's Composite pattern, you don't ever have to distinguish between components and composites, and you don't have to cast a Component reference to a Composite instance. So the code fragment listed above reduces to a single line:

```
...
component.addComponent(someComponentThatCouldBeAComposite);
...
```

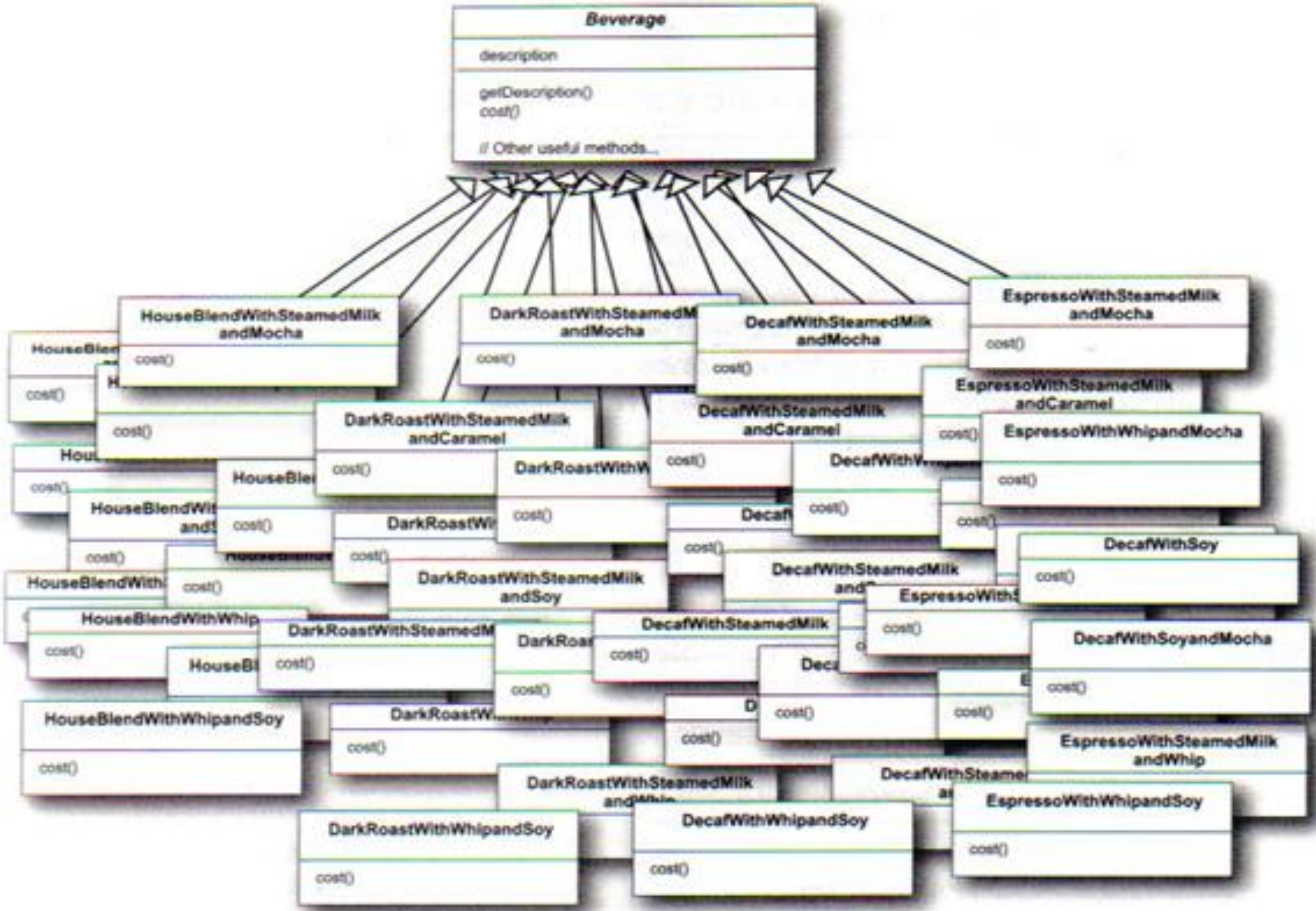
- But, if the Component reference in the preceding code fragment does not refer to a **Composite, what** should the **addComponent() do? That's a major point of contention with Figure 2's Composite pattern** implementation. Because primitive components do not contain other components, adding a component to another component makes no sense, so the **Component.addComponent() method can either fail** silently or throw an exception. Typically, adding a component to another primitive component is considered an error, so throwing an exception is perhaps the best course of action.

- Which Composite pattern implementation would you prefer and why?

Decorator Pattern: A coffee shop example…



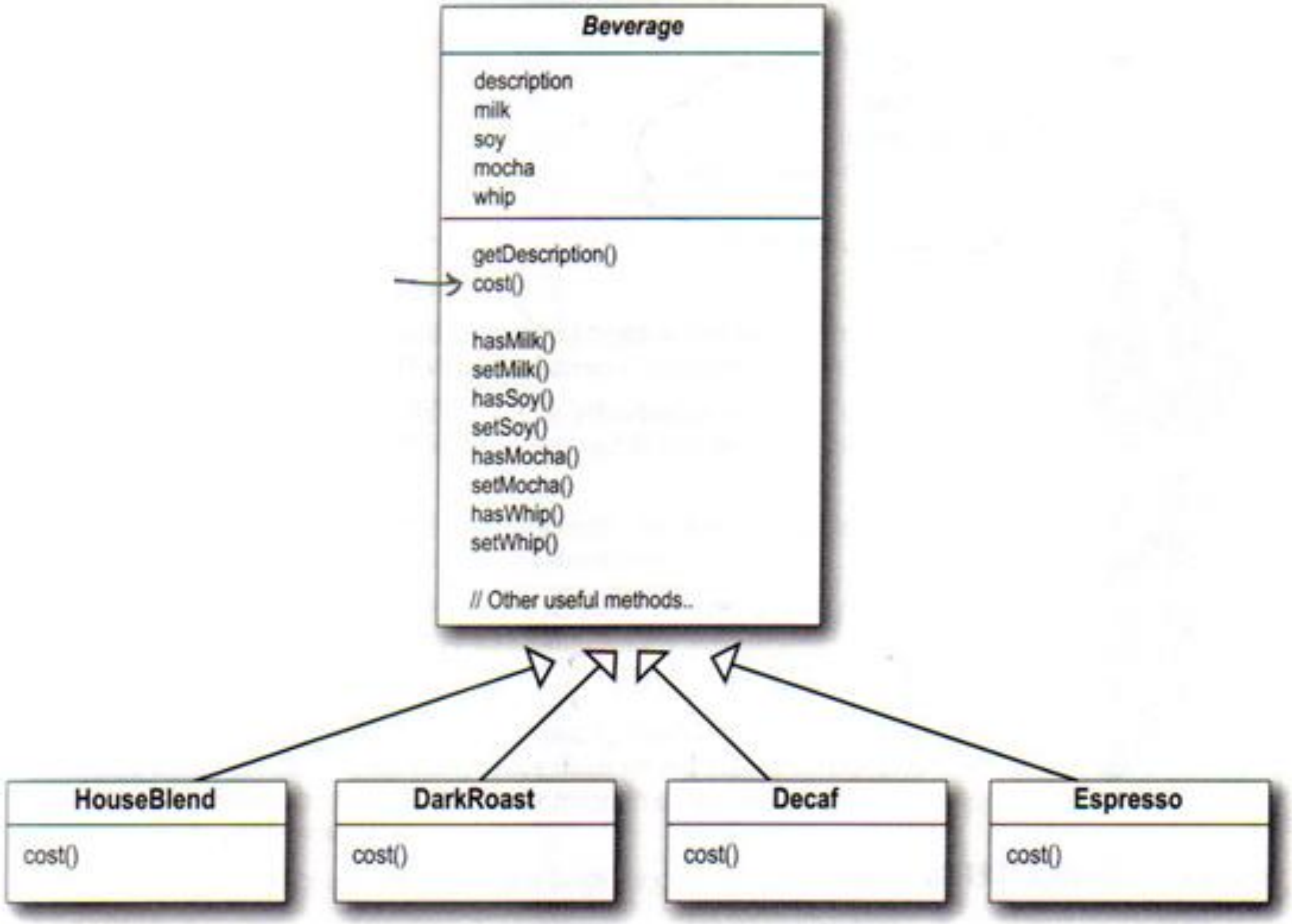
What if you want to show the addition of condiments such as steamed milk, soy, mocha and whipped milk?


Beverage
description
getDescription()
cost()
// Other useful methods...
HouseBlendWithSteamedMilk andMocha
DarkRoastWithSteamedMilk andMocha
DecafWithSteamedMilk andMocha
EspressoWithSteamedMilk andMocha
EspressoWithSteamedMilk andCaramel
DarkRoastWithSteamedMilk andCaramel
DecafWithSteamedMilk andCaramel
EspressoWithWhipandMocha
DecafWithSoy
DarkRoastWithSteamedMilk andSoy
DecafWithSteamedMilk
DecafWithSoyandMocha
HouseBlendWithWhip
HouseBlendWithWhipandSoy
EspressoWithSteamedMilk andWhip
DarkRoastWithSteamedMilk andWhip
EspressoWithWhipandSoy
DarkRoastWithWhipandSoy
DecafWithWhipandSoy
cost()

Whoa!
Can you say
"class explosion?"

Each cost method computes the cost of the coffee along with the other condiments in the order.

Beverage class redone



Potential problems with the design so far?

- Solution is not easily extendable
  - How to deal with
    - new condiments
    - Price changes
    - New beverages that may have a different set of condiments – a smoothie?
    - Double helpings of condiments

Design Principle

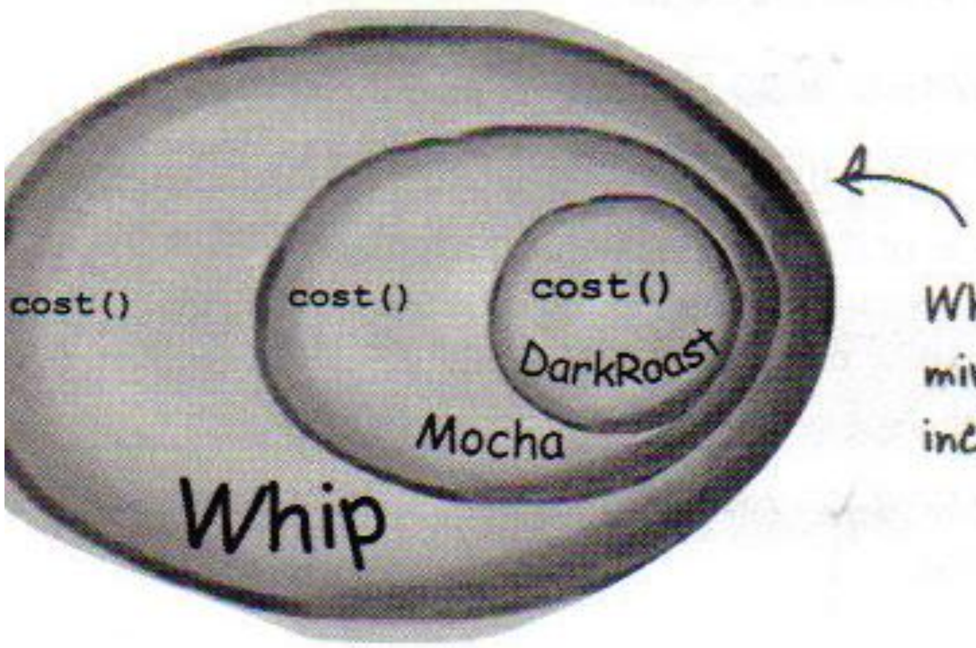
The Open-Closed Principle

Classes should be open for extension, but closed for modification.

The Decorator Pattern

- Take a coffee beverage object – say DarkRoast object
- Decorate it with Mocha
- Decorate it with Whip
- Call the cost method and rely on delegation to correctly compute the composite cost

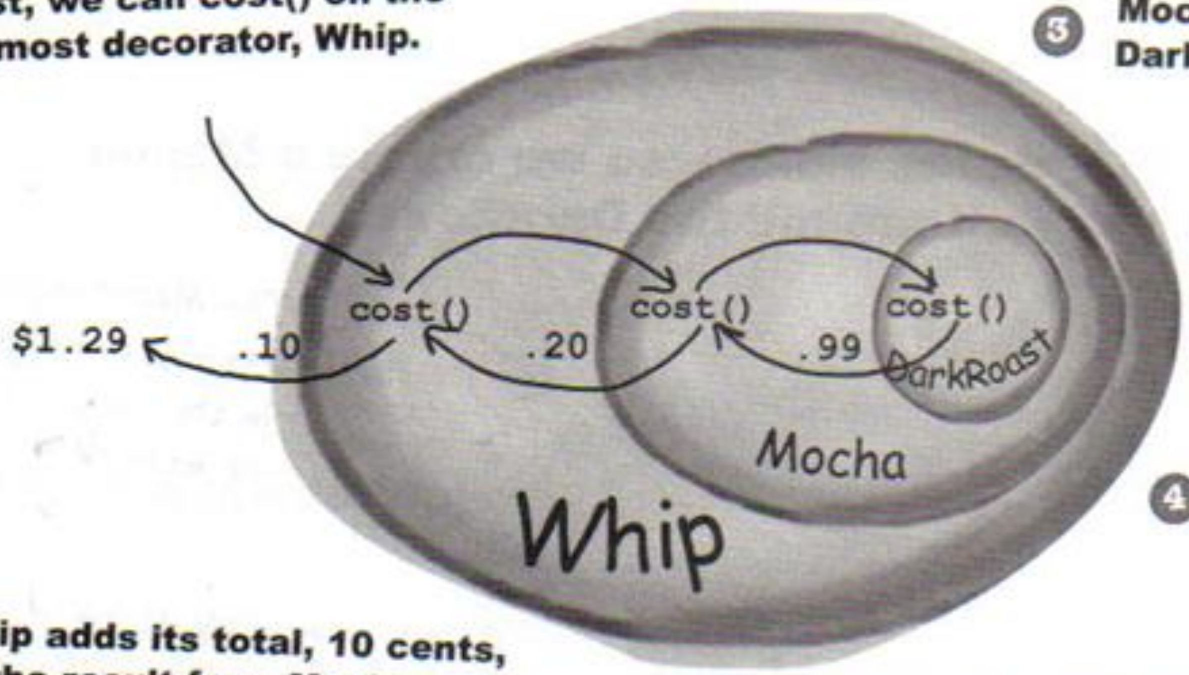
Decorator Pattern approach



Whip is a decorator, so it also mirrors DarkRoast's type and includes a cost() method.

DarkRoast wrapped in Mocha and Whip is still
rage and we can do anything with it we can do
DarkRoast, including call its cost() method.

Computing Cost using the decorator pattern



1 First, we call cost() on the outmost decorator, Whip.

2 Whip calls cost() on Mocha.

3 Mocha calls cost() on DarkRoast.

4 DarkRoast returns its cost, 99 cents.

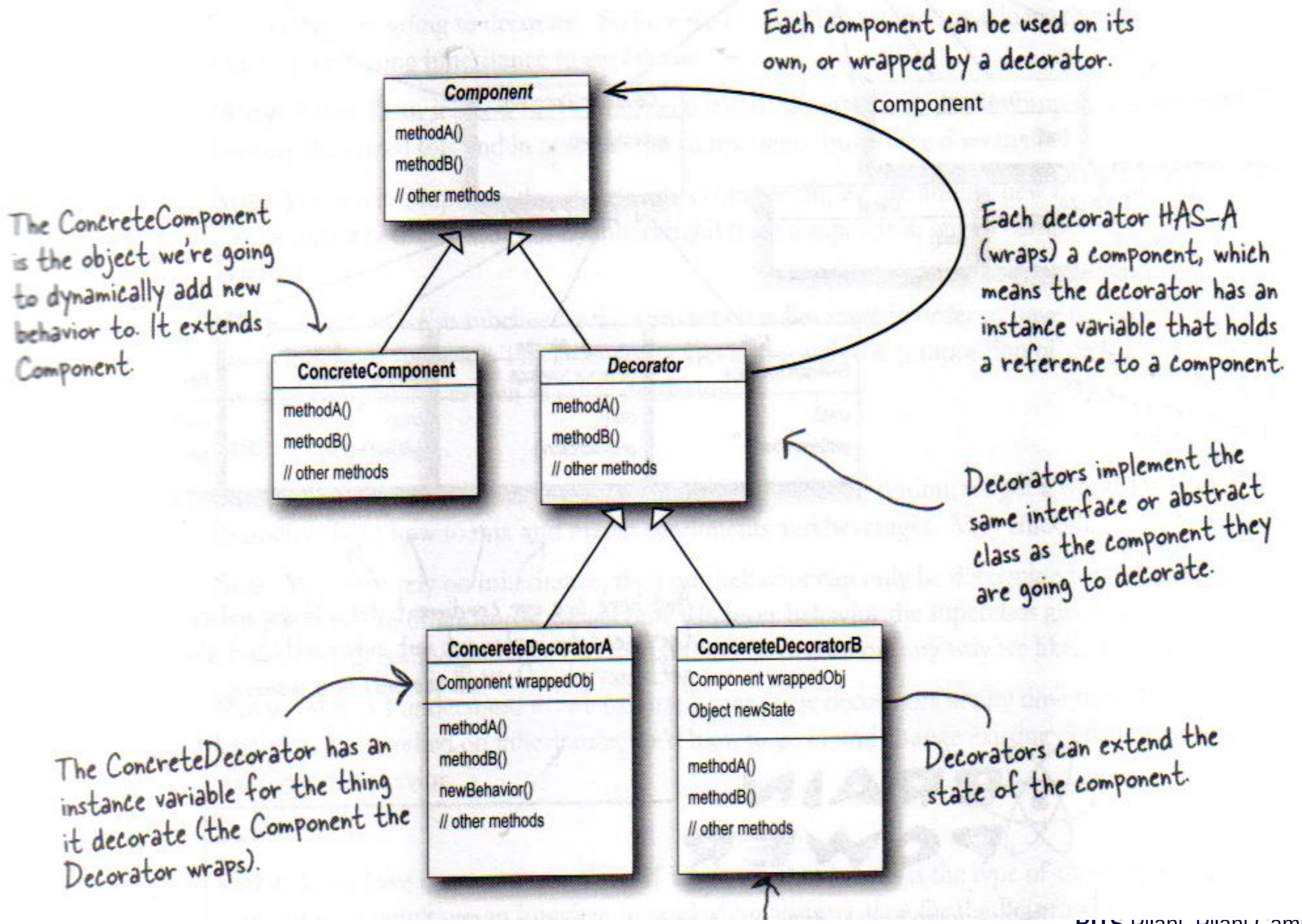
5 Mocha adds its cost, 20 cents, to the result from DarkRoast, and returns the new total, $1.19.

6 Whip adds its total, 10 cents, to the result from Mocha, and returns the final result—$1.29.

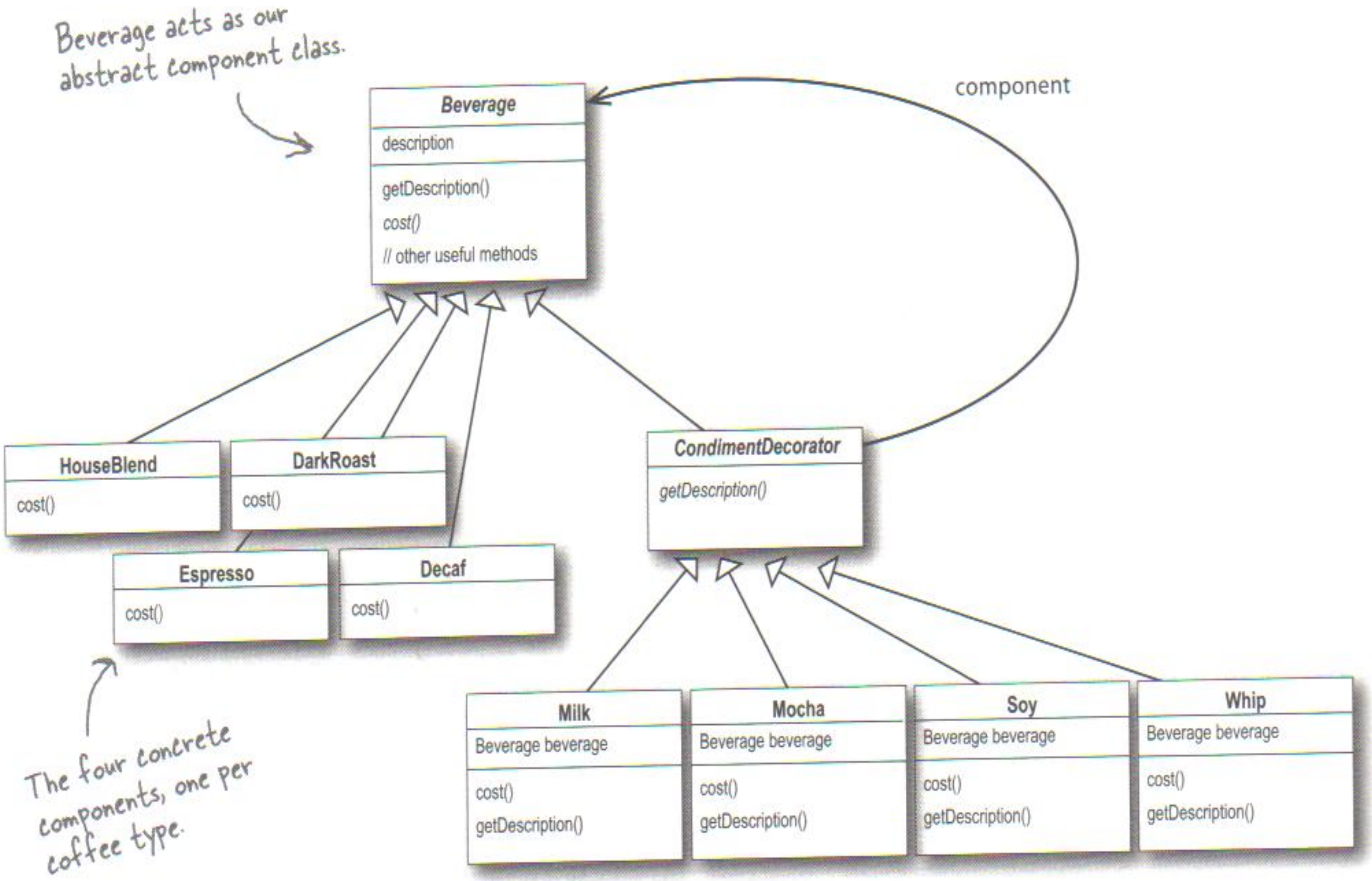
Decorator Pattern

The decorator pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub-classing for extending functionality.

Decorator Pattern Defined


Each component can be used on its own, or wrapped by a decorator.
component
Component
methodA()
methodB()
// other methods
The ConcreteComponent is the object we're going to dynamically add new behavior to. It extends Component.
ConcreteComponent
methodA()
methodB()
// other methods
Decorator
methodA()
methodB()
// other methods
Each decorator HAS-A (wraps) a component, which means the decorator has an instance variable that holds a reference to a component.
Decorators implement the same interface or abstract class as the component they are going to decorate.
ConcereteDecoratorA
Component wrappedObj
methodA()
methodB()
newBehavior()
// other methods
ConcereteDecoratorB
Component wrappedObj
Object newState
methodA()
methodB()
// other methods
The ConcreteDecorator has an instance variable for the thing it decorate (the Component the Decorator wraps).
Decorators can extend the state of the component.

Decorator Pattern for Beverage Example



Keyboard Input

The **System** class in java provides an **InputStream** object: **System.in**

And a buffered **PrintStream** object **System.out**

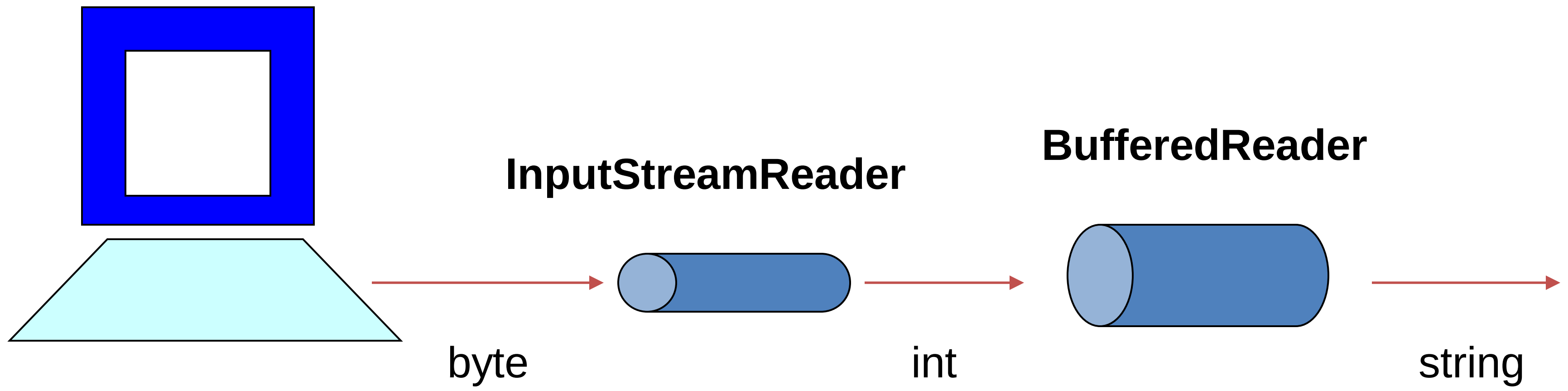
The PrintStream class (**System.out**) provides support for outputting primitive data type values.

However, the InputStream class only provides methods for reading byte values.

To extract data that is at a “higher level” than the byte, we must “encase” the **InputStream**, **System.in**, inside an **InputStreamReader** object that converts byte data into 16-bit character values (**returned as an int**).

We next “wrap” a **BufferedReader** object around the **InputStreamReader** to enable us to use the methods **read**( ) which returns a char value and **readLine**( ), which return a String.

Keyboard Input



```
import java.io.*;            //for keyboard input stream

InputStreamReader isr = new InputStreamReader(System.in);
BufferedReader br = new BufferedReader(isr);
String st = br.readLine( );
                             //reads chars until eol and forms string
```

innovate achieve lead

Observer Design Pattern

Observer Design Pattern

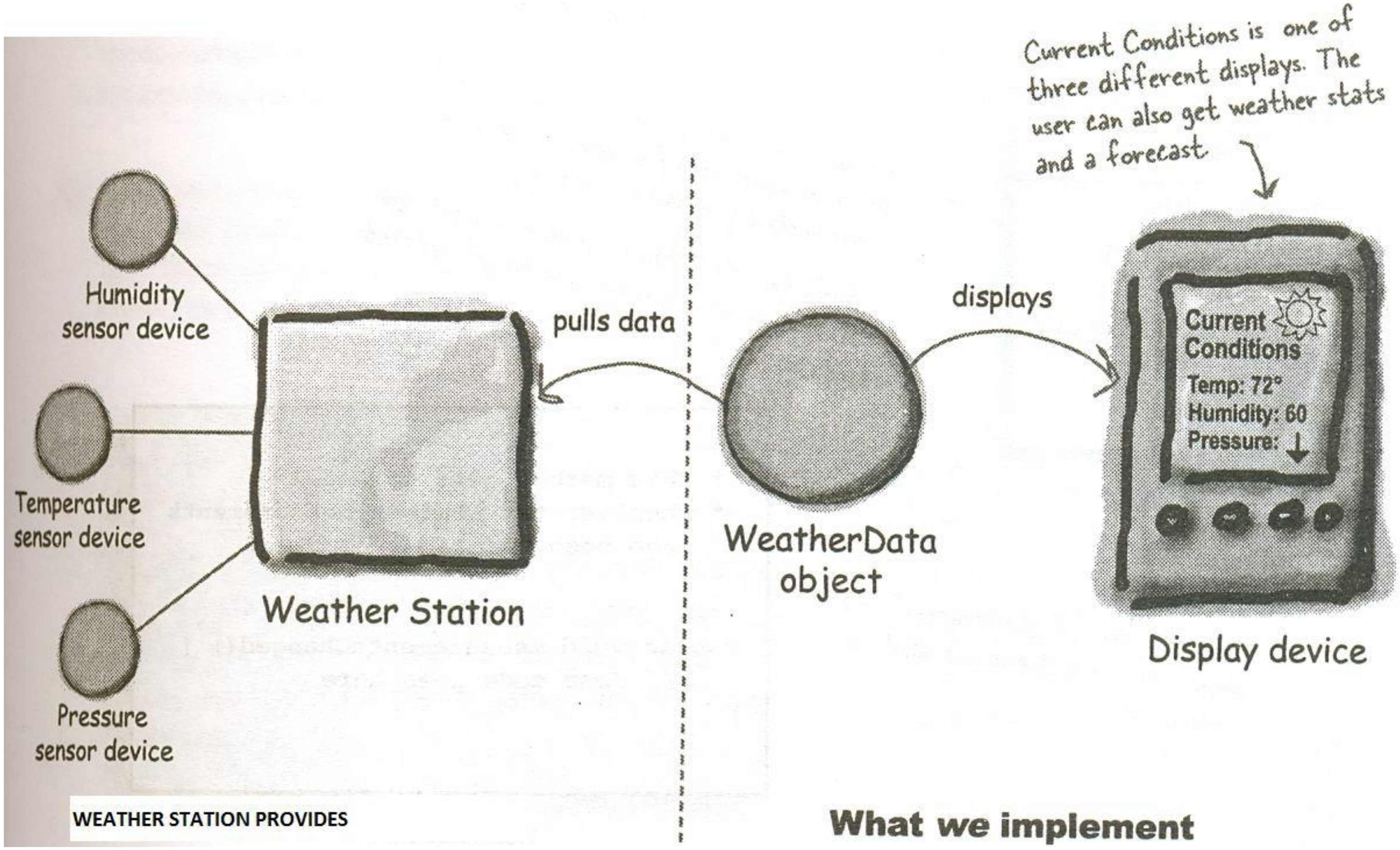
INTERNET BASED WEATHER MONITORING STATION

- The weather station is based on WeatherData object, which tracks current weather conditions (temperature, humidity, and barometric pressure). We have to create an application that initially provides three display elements:
  - Current conditions
  - Weather statistics
  - Simple forecast
- All these display elements are updated in real-time as the WeatherData object acquires the most recent measurements.
- This is an expandable weather station. We need to release an API so that other developers can write their own weather displays and plug them right in.

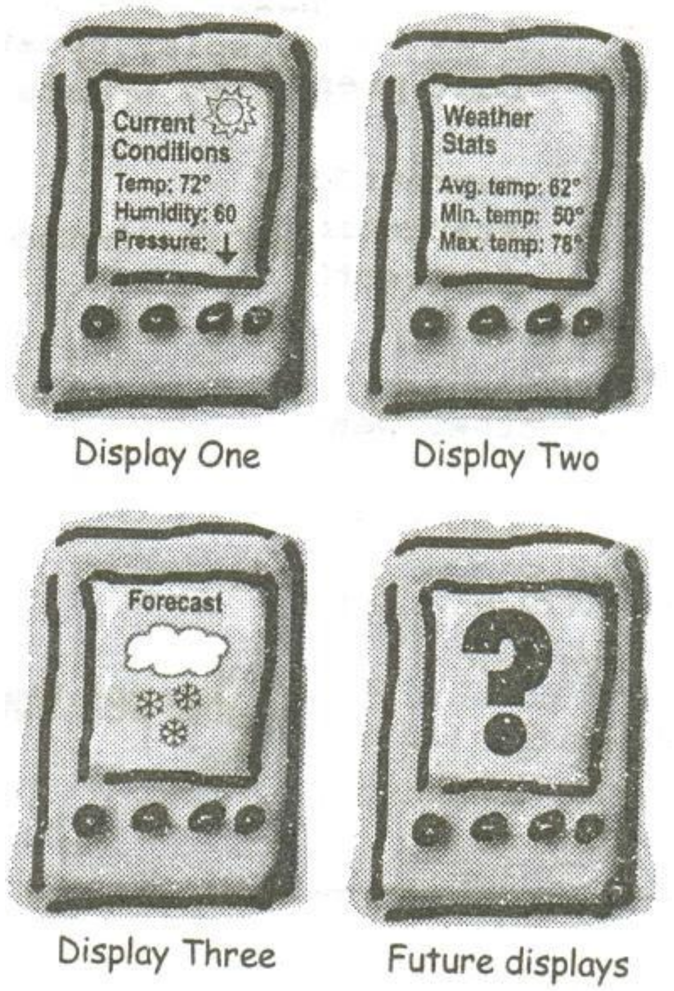
Application Overview

- The three players in the system are the weather station (the physical device that acquire actual
- weather data), the WeatherData object (that tracks the data coming from the Weather Station and
- updates the displays), and the display that shows the users the current weather conditions.

Weather Monitoring Application

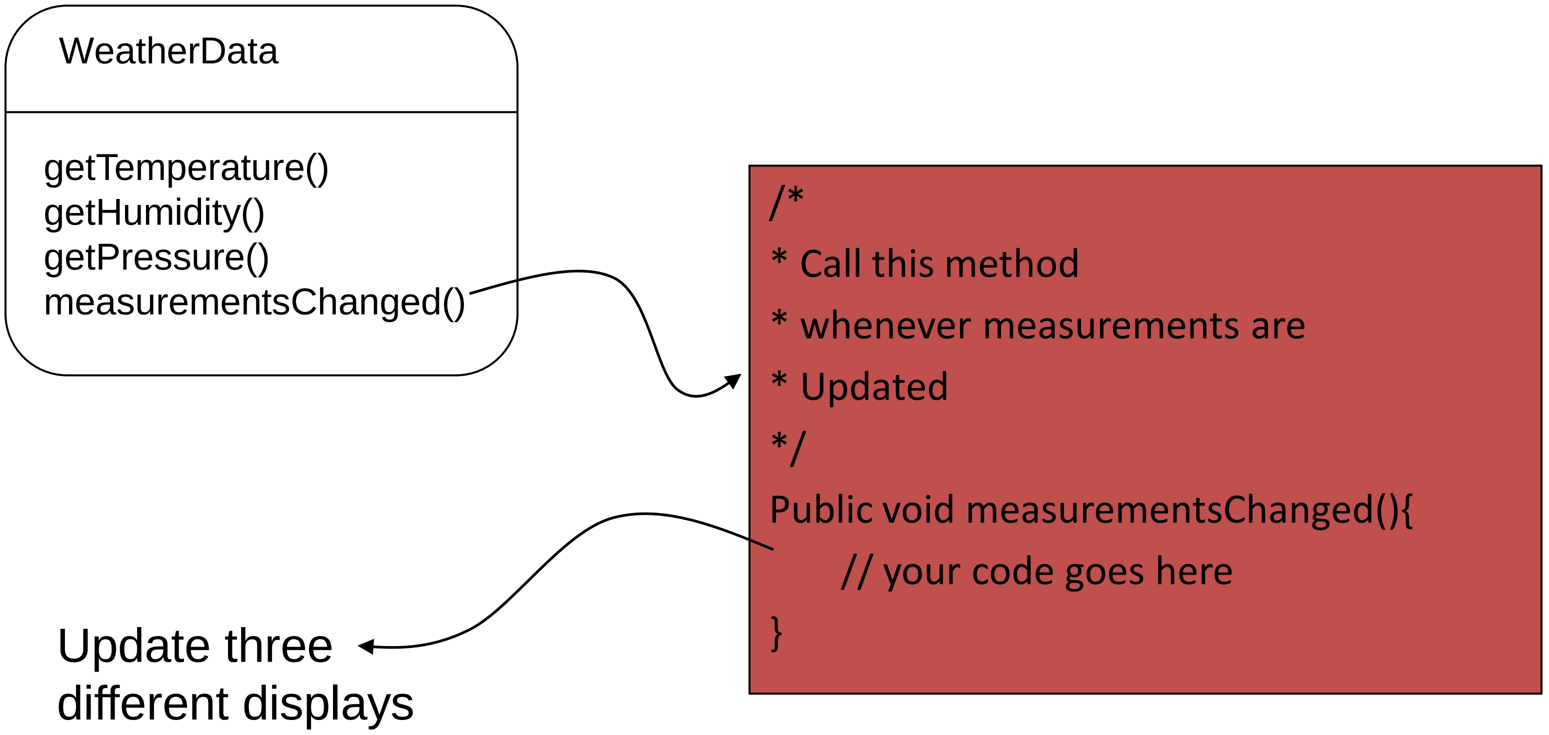


Display Elements



What needs to be done?

Create an application that uses the WeatherData object to update three displays for current conditions, weather stats, and forecast.



```
/*
* Call this method
* whenever measurements are
* Updated
*/
Public void measurementsChanged(){
     // your code goes here
}
```

What we know so far

* The WeatherData class has getter methods for 3 measurement values.

* The measurementChanged method is called any time new measurements data is available. (We don't care how this method is called)

* we need to implement 3 display elements that uses the weather data
  → current-Condition Display
  → Statistics Display
  → Forecast Display

* These displays must be update each time Weather Data has new measurements

* The system must be expandable - other developers can create new custom display elements & users can add or remove as many displays as they want.

* Currently we know about only 3 displays.

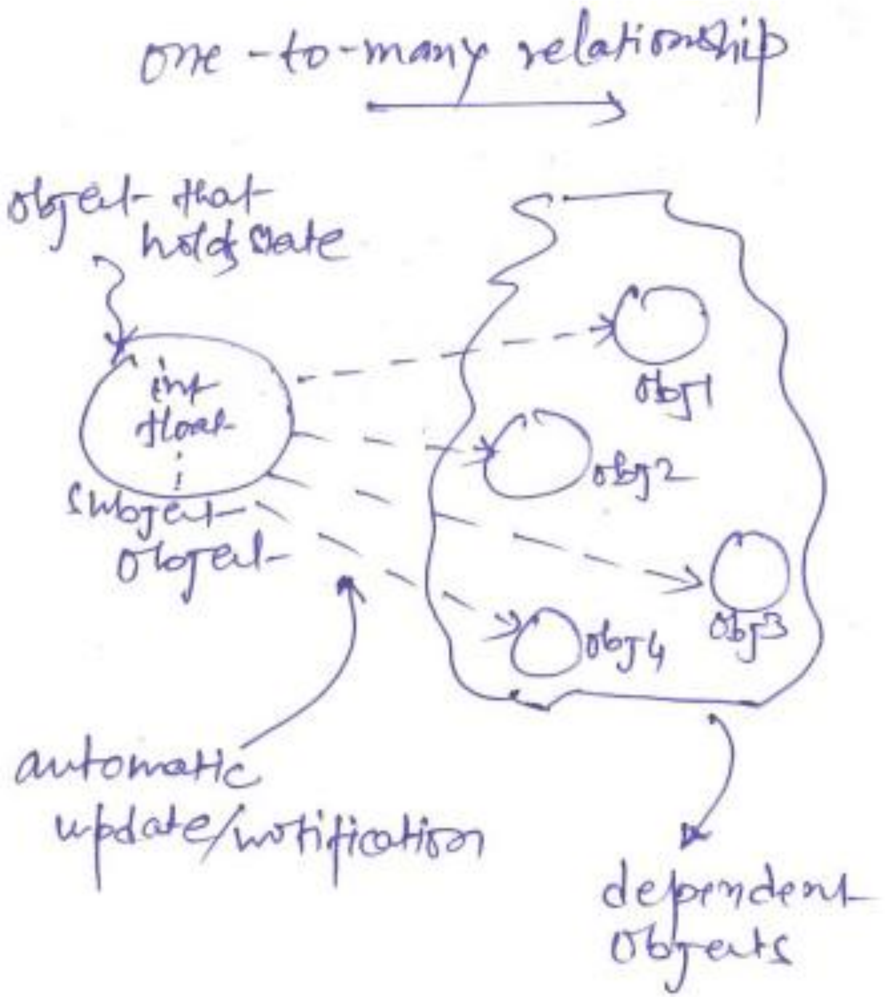
```java
public class WeatherData {

      //instance variables
      public void measurementChanged(){

            //grab the most recent measurements by calling
            //the WeatherData getter methods (already implemented)
            float temp = getTemperature();
            float humidity = getHumidity();
            float pressure = getPressure();


            //update the display by calling each display element to update
            //its display, passing it the most recent measurements

            currentConditionsDisplay.update(temp, humidity, pressure);

            statisticsDisplay.update(temp, humidity, pressure);

            forecastDisplay.update(temp, humidity, pressure);

      }
}
```

By coding to concrete implementations we have no way to add or remove other display elements without making changes to the program

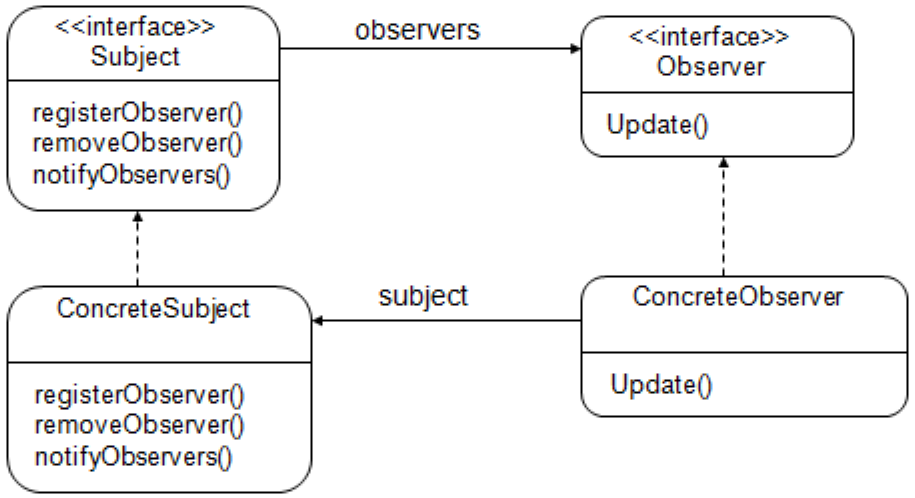
Atleast we seem to be using a common interface to talk to the display elements... they all have an update() method takes the temp, humidity, and pressure data

Observer Pattern Defined

The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.



Observer Pattern – Class diagram



* Subject interface →
Objects use this to register as observers & also to remove themselves from being observer.

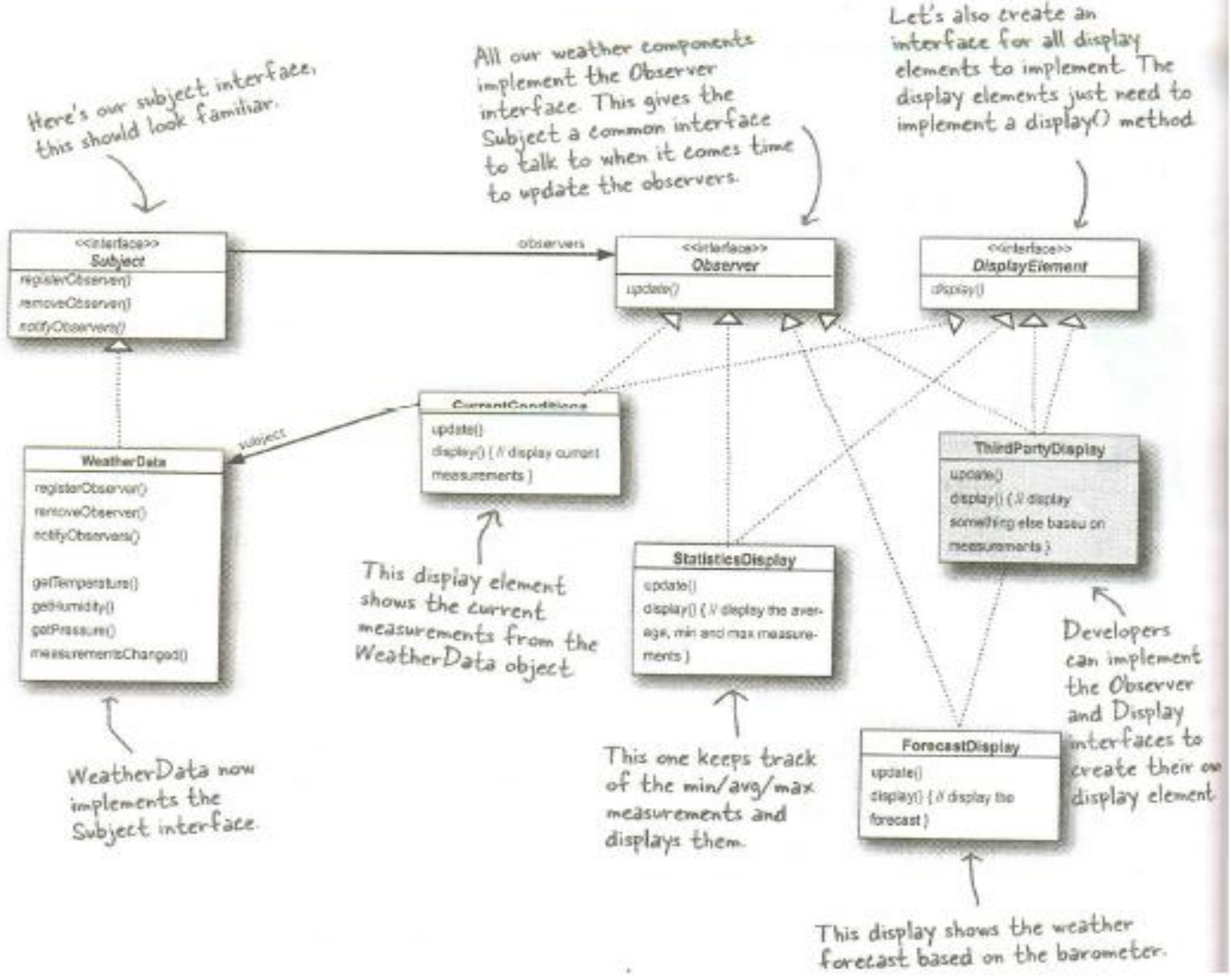
* Observer interface →
observers need to implement the observer interface. This interface just has one method update that gets called when the subject's state change.

* Concrete Subject → implement subject interface. It has implementations for register, remove & a method to notify observers.

* Concrete observer →
Concrete observers can be any class that implements the observer interface. Each observer registers with a concrete subject to receive updates.

Observer pattern – power of loose coupling

- The only thing that the subject knows about an observer is that it implements an interface
- Observers can be added at any time and subject need not be modified to add observers
- Subjects and observers can be reused or modified without impacting the other [as long as they honor the interface commitments]


Here's our subject interface, this should look familiar.
All our weather components implement the Observer interface. This gives the Subject a common interface to talk to when it comes time to update the observers.
Let's also create an interface for all display elements to implement. The display elements just need to implement a display() method.
<<interface>> Subject
registerObserver()
removeObserver()
notifyObservers()
observers
<<interface>> Observer
update()
<<interface>> DisplayElement
display()
subject
WeatherData
registerObserver()
removeObserver()
notifyObservers()
getTemperature()
getHumidity()
getPressure()
measurementsChanged()
CurrentConditions
update()
display() { // display current measurements }
ThirdPartyDisplay
update()
display() { // display something else based on measurements }
StatisticsDisplay
update()
display() { // display the average, min and max measurements }
ForecastDisplay
update()
display() { // display the forecast }
This display element shows the current measurements from the WeatherData object
WeatherData now implements the Subject interface.
This one keeps track of the min/avg/max measurements and displays them.
Developers can implement the Observer and Display interfaces to create their ow display element.
This display shows the weather forecast based on the barometer.

Weather data interfaces

```
public interface Subject {
    public void registerObserver(Observer o);
    public void removeObserver(Observer o);
    public void notifyObservers();
}
public interface Observer {
    public void update(float temp, float humidity, float pressure);
}
public interface DisplayElement {
    public void display();
}
```

Implementing subject interface

```
public class WeatherData implements Subject {
    private ArrayList observers;
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() {
        observers = new ArrayList();
    }
```

Register and unregister

```
public void registerObserver(Observer o) {
        observers.add(o);
}

public void removeObserver(Observer o) {
        int i = observers.indexOf(o);
        if (i >= 0) {
                observers.remove(i);
        }
}
```

Notify methods

```
public void notifyObservers() {
        for (int i = 0; i < observers.size(); i++) {
                Observer observer = (Observer)observers.get(i);
                observer.update(temperature, humidity, pressure);
        }
   }

   public void measurementsChanged() {
        notifyObservers();
   }
```

Push or pull

- The notification approach used so far pushes all the state to all the observers
- One can also just send a notification that some thing has changed and let the observers pull the state information
- Java observer pattern support has built in support for both push and pull in notification

Problems with Java implementation

- Observable is a class
  - You have to subclass it
  - You cannot add observable behavior to an existing class that already extends another superclass
  - *You have to program to an implementation – not interface*
- Observable protects crucial methods
  - Methods such as setChanged() are protected and not accessible unless one subclasses Observable.
  - *You cannot favor composition over inheritance.*
- You may have to roll your own observer interface if Java utilities don't work for your application

Strategy Design Pattern

Intent

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

Motivation

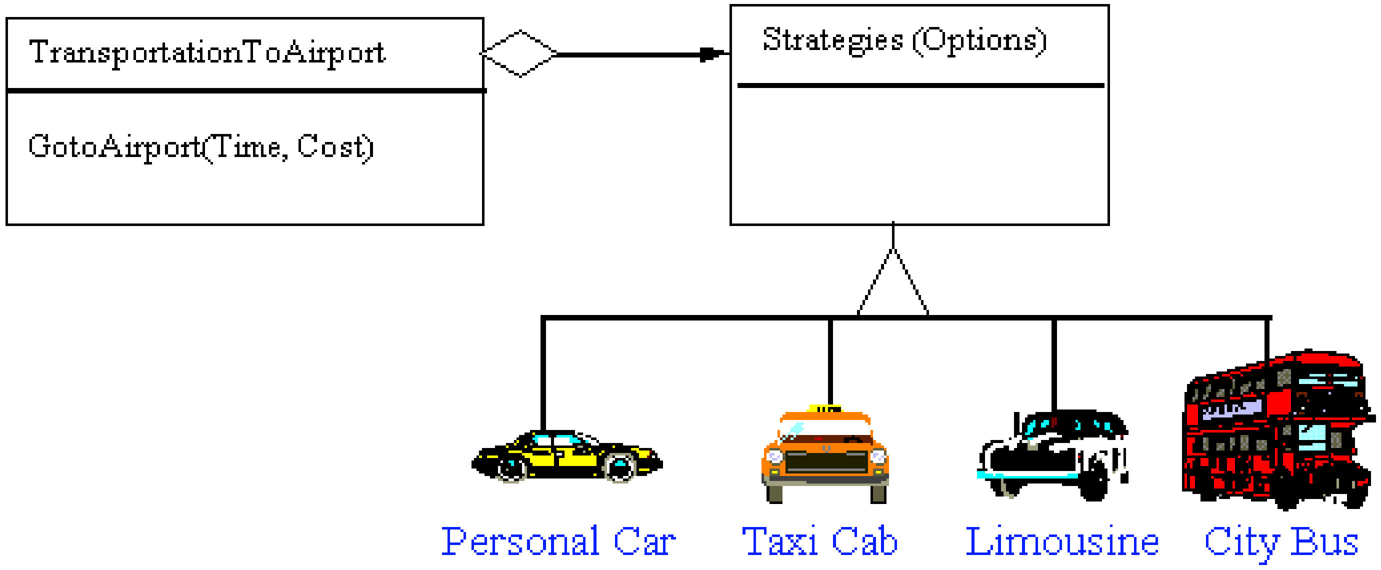
- Hard-wiring all algorithms into a class makes the client more complex, bigger and harder to maintain.
- We do not want to use all the algorithms.
- It’s difficult to add and vary algorithms when the algorithm code is an integral part of a client.

Applicability

- Many related classes differ only in their behavior.
- You need different variants of an algorithm. Strategy can be used as a class hierarchy of algorithms.
- An algorithm use data structures that clients shouldn’t know about.
- A class defines many behaviors, and these appear as multiple conditionals in its operation.

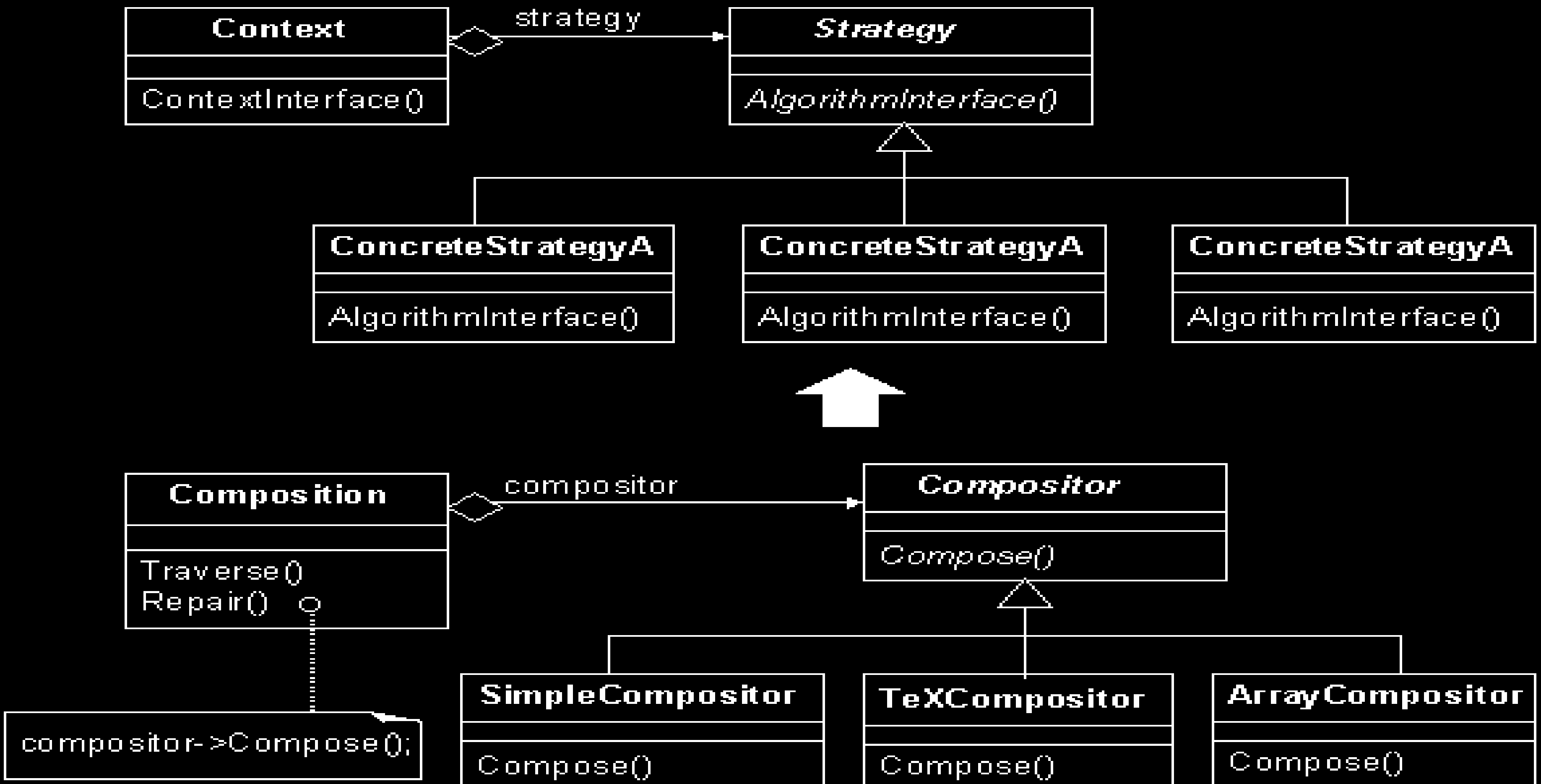
Example

- Modes of transportation to an airport is an example of a Strategy. Several options exist, such as driving one's own car, taking a taxi, an airport shuttle, a city bus, or a limousine service. Any of these modes of transportation will get a traveler to the airport, and they can be used interchangeably. The traveler must chose the Strategy based on tradeoffs between cost, convenience, and time.



Example

- Many algorithms exists for breaking a string into lines.
- Simple Compositor is a simple linebreaking method.
- TeX Compositor uses the TeX linebreaking strategy that tries to optimize linebreaking by breaking one paragraph at a time.
- Array Compositor breaks a fixed number of items into each row.


Context
ContextInterface()
strategy
Strategy
AlgorithmInterface()
ConcreteStrategyA
AlgorithmInterface()
ConcreteStrategyA
AlgorithmInterface()
ConcreteStrategyA
AlgorithmInterface()
Composition
Traverse()
Repair()
compositor
Compositor
Compose()
compositor->Compose();
SimpleCompositor
Compose()
TeXCompositor
Compose()
ArrayCompositor
Compose()

Participants

- Strategy declares an interface common to all supported algorithms. Context uses its interface to call the algorithm defined by a ConcreteStrategy.
- ConcreteStrategy implements a specific algorithm using the Strategy interface.
- Context
  - is configured with a ConcreteStrategy object.
  - maintains a reference to a Strategy object.
  - may define an interface for Strategy

Consequences

- Families of related algorithms
  - Hierarchies of Strategy factor out common functionality of a family of algorithms for contexts to reuse.
- An alternative to subclassing
  - Subclassing a Context class directly hard-wires the behavior into Context, making Context harder to understand, maintain, and extend.
  - Encapsulating the behavior in separate Strategy classes lets you vary the behavior independently from its context, making it easier to understand, replace, and extend.
- Strategies eliminate conditional statements.
  - Encapsulating the behavior into separate Strategy classes eliminates conditional statements for selecting desired behavior.
- A choice of implementations
  - Strategies can provide different implementations of the same behavior with different time and space trade-offs.

Consequences (cont..)

- Clients must be aware of different strategies.
  - A client must understand how Strategies differ before it can select the appropriate one.
  - You should use the Strategy pattern only when the variation in behavior is relevant to clients.
- Communication overhead between Strategy and Context.
  - The Strategy interface is shared by all ConcreteStrategy classes.
  - It's likely that some ConcreteStrategies will not use all the information passed to them through this common interface.
  - To avoid passing data that get never used, you'll need tighter coupling between Strategy and Context.
- Increased number of objects
  - Strategies increase the number of objects in an application.
  - Sometimes you can reduce this overhead by implementing strategies as stateless objects that context can share.

Iterator Design Pattern

Diner and Pancake House Merger

Objectville diner and Objectville pancake house are merging into one entity. Thus, both menus need to merged. The problem is that the menu items have been stored in an ArrayList for the pancake house and an Array for the diner. Neither of the owners are willing to change their implementation.

Problems

- Suppose we are required to print every item on both menus.
- Two loops will be needed instead of one.
- If a third restaurant is included in the merger, three loops will be needed.
- Design principles that would be violated:
  - Coding to implementation rather than interface
  - The program implementing the joint print_menu() needs to know the internal structure of the collection of each set of menu items.
  - Duplication of code

Solution

- Encapsulate what varies, i.e. encapsulate the iteration.
- An iterator is used for this purpose.
- The *DinerMenu* class and the *PancakeMenu* class need to implement a method called *createIterator()*.
- The *Iterator* is used to iterate through each collection without knowing its type (i.e. Array or ArrayList)

Original Iteration

- Getting the menu items:

```
PancakeHouseMenu pancakeHouseMenu= new PancakeHouseMenu();
ArrayList breakfastItems = pancakeHouseMenu.getMenuItems();
DinerMenu dinerMenu = new DinerMenu();
MenuItems[] lunchItems = dinerMenu.getMenuItems():
```

- Iterating through the breakfast items:

```
for(int i=0; i < breakfastItems.size(); ++i)
{
  MenuItem menuItem=
             (MenuItem) breakfastItems.get(i)
}
```

- Iterating through the lunch items:

```
for(int i=0; I < lunchItems.length; i++)
{
     MenuItem menuItem = lunchItems[i]
}
```

Using an Iterator

- Iterating through the breakfast items:

```
Iterator iterator = breakfastMenu.createInstance();
while(iterator.hasNext())
{
    MenuItem menuItem = (MenuItem)iterator.next();
}
```

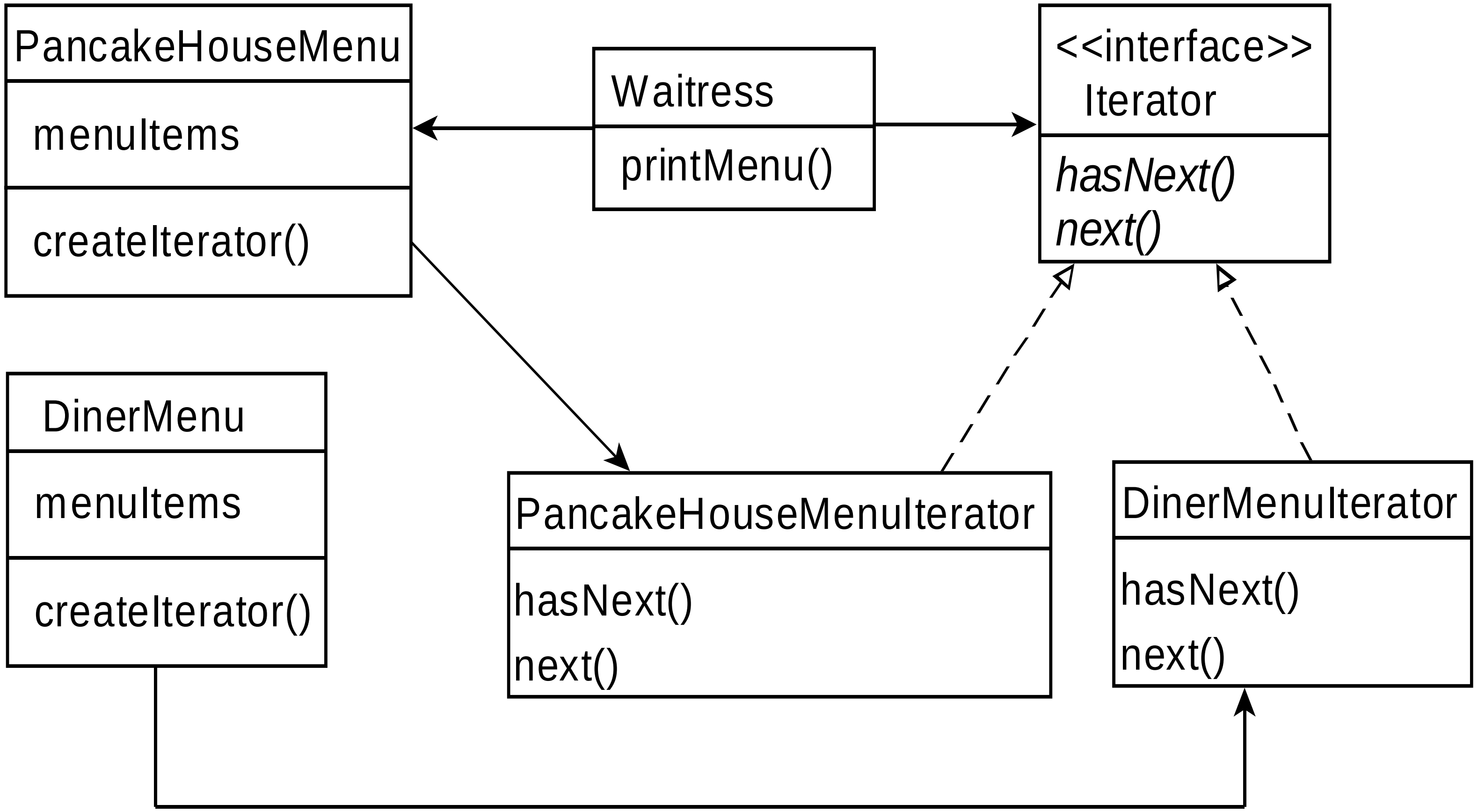
- Iterating through the lunch items:

```
Iterator iterator = lunchMenu.createIterator();
while(iterator.hasNext())
{
    MenuItem menuItem = (MenuItem)iterator.next();
}
```

Iterator Design Pattern

- The iterator pattern encapsulates iteration.
- The iterator pattern requires an interface called *Iterator*.
- The *Iterator* interface has two methods:
  - *hasNext()*
  - *next()*
- Iterators for different types of data structures are implemented from this interface.

Class Diagram for the Merged Diner



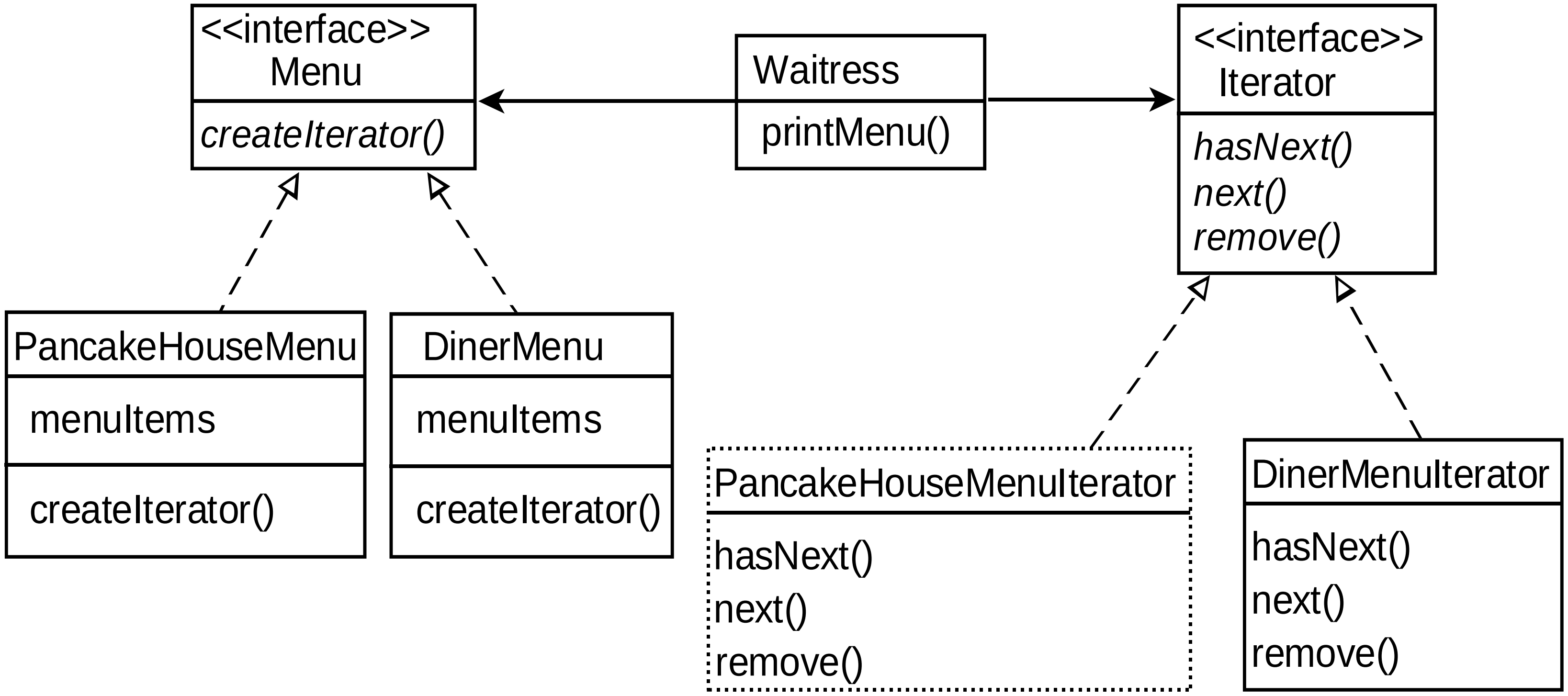
Using the Java *Iterator* Class

- Java has an *Iterator* class.
- The *Iterator* class has the following methods:
  - *hasNext()*
  - *next()*
  - *remove()*
- If the *remove()* method should not be allowed for a particular data structure, a *java.lang.UnsupportedOperationException* should be thrown.

Improving the Diner Code

- Changing the code to use *java.util.iterator*:
  - Delete the PancakeHouseIterator as the ArrayList class has a method to return a Java iterator.
  - Change the *DinerMenuIterator* to implement the Java *Iterator* .
- Another problem - all menus should have the same interface.
  - Include a *Menu* interface

Class Diagram Include *Menu*

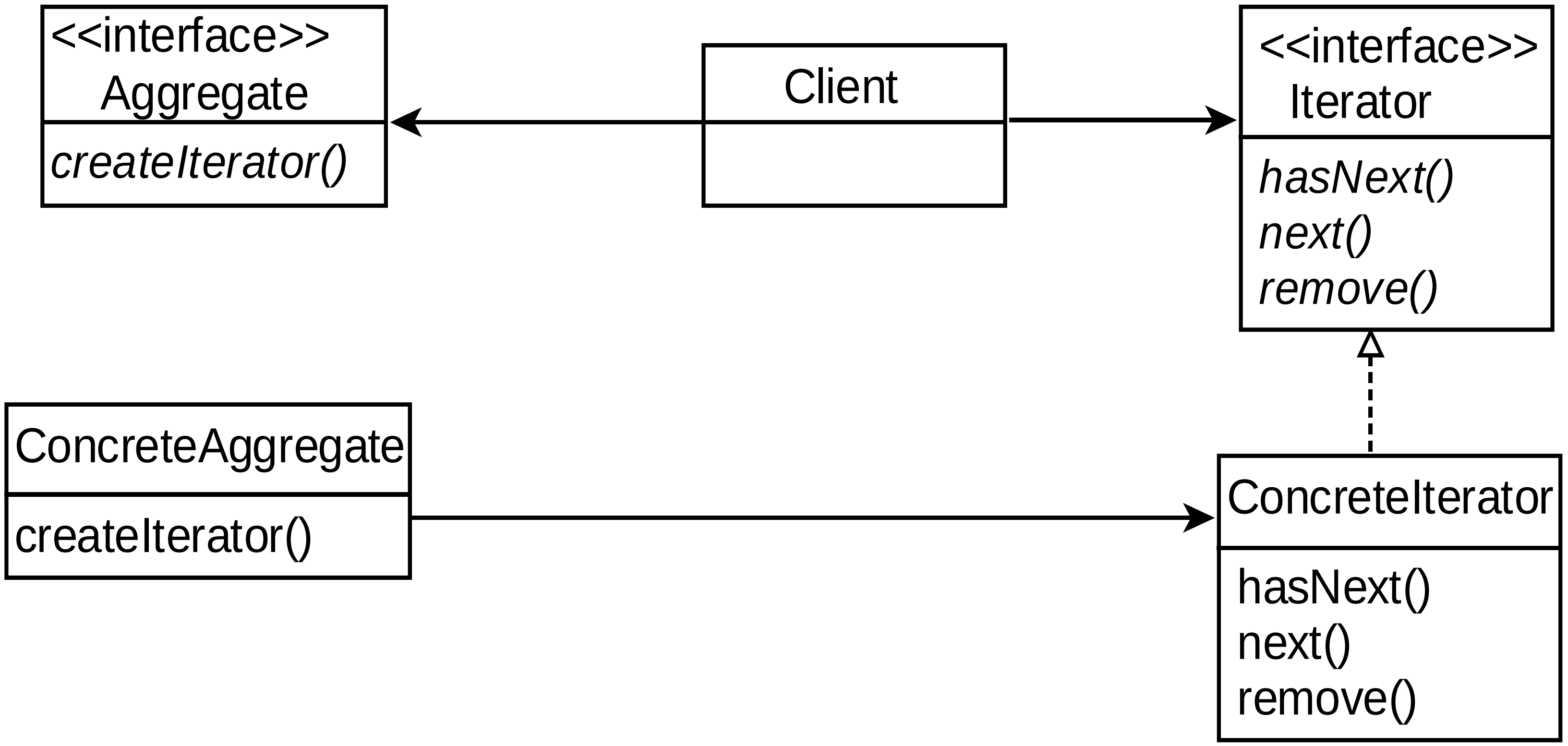


Adding the *Menu* interface

Iterator Pattern Definition

- Provides a way to access elements of a collection object sequentially without exposing its underlying representation.
- The iterator object takes the responsibility of traversing a collection away from collection object.
- This simplifies the collection interface and implementation.
- Allows the traversal of the elements of a collection without exposing the underlying implementation.

Iterator Pattern Class Diagram



Some Facts About the Iterator Pattern

- Earlier methods used by an iterator were *first(), next(), isDone()* and *currentItem()*.
- Two types of iterators: internal and external.
- An iterator can iterate forward and backwards.
- Ordering of elements is dictated by the underlying collection.
- Promotes the use of “polymorphic” iteration by writing methods that take Iterators as parameters.
- *Enumeration* is a predecessor of *Iterator*.

Design Principle

- If collections have to manage themselves as well as iteration of the collection this gives the class two responsibilities instead of one.
- Every responsibility if a potential area for change.
- More than one responsibility means more than one area of change.
- Thus, each class should be restricted to a single responsibility.
- Single responsibility: A class should have only one reason to change.
- High cohesion vs. low cohesion.

innovate achieve lead

Thank You