SOFTWARE ENGINEERING

Software Design

Design Patterns (Behavioral)

Course ID 06016410, 06016321

Nont Kanungsukkasem, B.Eng., M.Sc., Ph.D.
nont@it.kmitl.ac.th

3) Behavioral Design Patterns
Rationale

- Characterize the ways in which classes or objects interact and distribute responsibility
- Address the problems that arise when assigning responsibilities to classes and when designing algorithms
- Suggest static relationships among objects and classes
- Describe how objects communicate
- Inheritance, aggregation, composition are used

3) Behavioral Design Patterns Catalogue (1)

- **Observer:** Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- **Strategy:** Define a family of algorithms, encapsulate each one, and make them interchangeable, allowing the family to vary independently from its clients.
- **State:** Allow an object to alter its behavior when its internal state changes, thus making the object appear to change its class.
- **Command:** Encapsulate a request as an object, thereby supporting parameterization of different requests from clients; queuing or logging of requests; and reversible operations.

2) Behavioral Design Patterns Catalogue (2)

- **Interpreter:** Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
- **Iterator:** Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- **Mediator:** Define an object that encapsulates how a set of objects interact. The Mediator promotes loose coupling by keeping objects from referring to each other directly and allowing their interaction to vary independently.
- **Memento:** Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

3) Behavioral Design Patterns Catalogue (3)

- **Template Method:** Define a skeleton of an algorithm in an operation, deferring some steps to subclasses. The Template Method lets subclasses redefine steps of the algorithm without changing the algorithm's structure.
- **Visitor:** Represent an operation to be performed on the elements of an object structure. Define a new operation without changing the classes of the elements on which it operates
- **Chain of Responsibility:** Avoid coupling a requesting sender to a receiver by giving more than one receiver a chance to handle the request. The Chain of Responsibility chains receiving objects and passes each request along the chain until an object handles it.

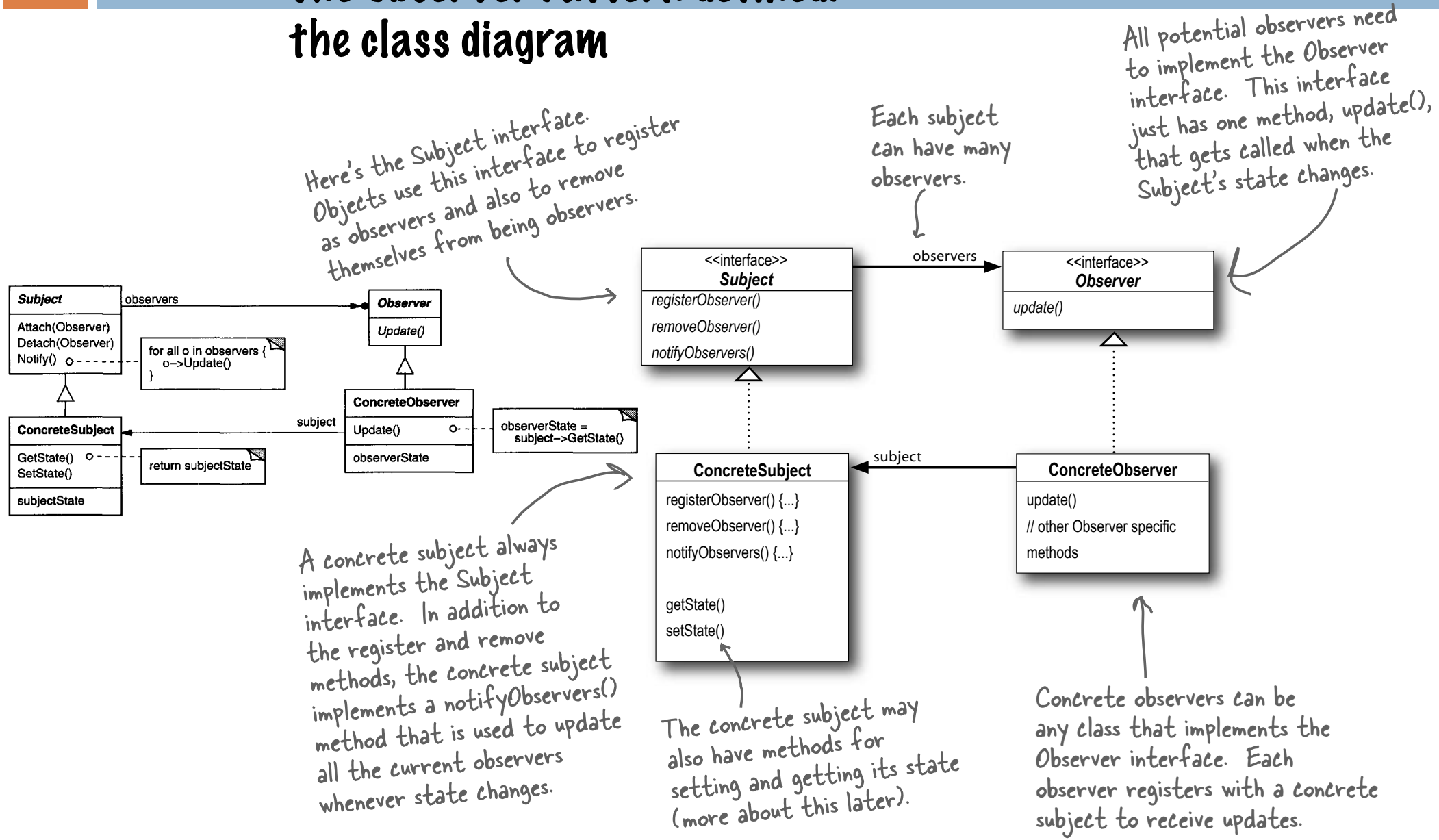
Observer Pattern – Description (1)

- **Name:** Observer
- **Problem:** How can one design a class that its object would be notified or updated automatically when there is a state change in another object?
- **Context:** In some applications, it is important to get notifications or updates regarding any state changes of an object, called *subject*. Another object that is listening for the change is called an *observer* or *listener*.

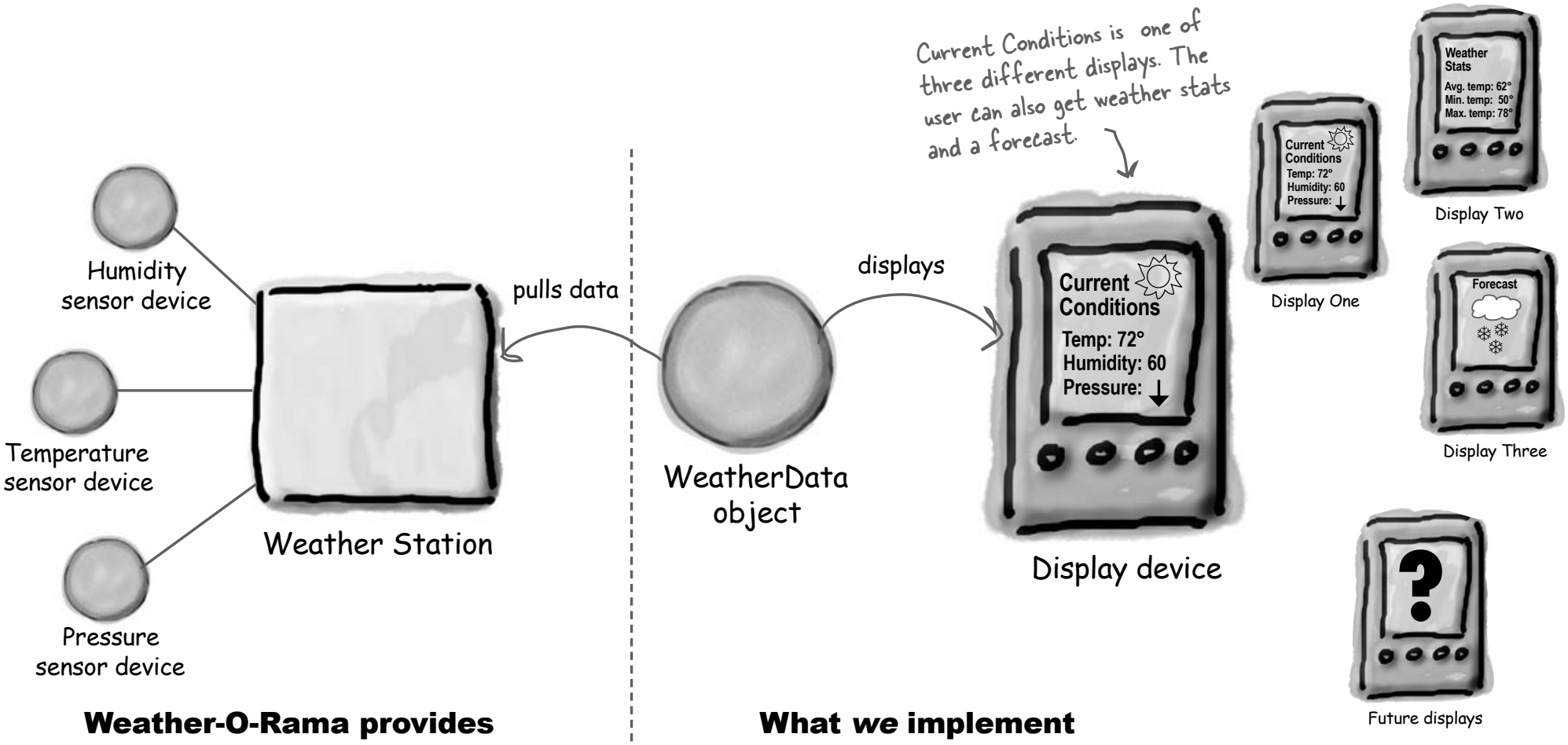
Observer Pattern – Description (2)

7

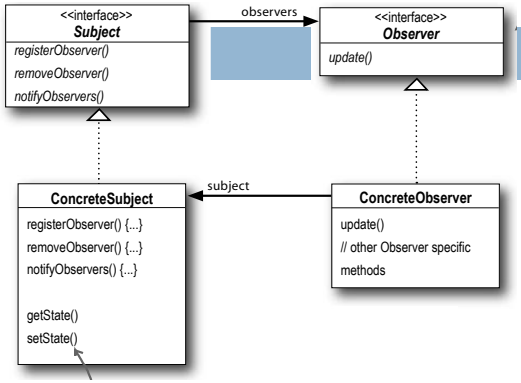
The Observer Pattern defined: the class diagram



Example Code – Weather Monitoring (1)



Example Code – Weather Monitoring (2)




```
public interface Subject {
    public void registerObserver(Observer o);
    public void removeObserver(Observer o);
    public void notifyObservers();
}
```

Both of these methods take an Observer as an argument; that is, the Observer to be registered or removed.

This method is called to notify all observers when the Subject's state has changed.

```
public interface Observer {
    public void update(float temp, float humidity, float pressure);
}
```

These are the state values the Observers get from the Subject when a weather measurement changes

The Observer interface is implemented by all observers, so they all have to implement the update() method. Here we're following Mary and Sue's lead and passing the measurements to the observers.

```
public interface DisplayElement {
    public void display();
}
```

The DisplayElement interface just includes one method, display(), that we will call when the display element needs to be displayed.

Example Code – Weather Monitoring (3)


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

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

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

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

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

WeatherData now implements the Subject interface.

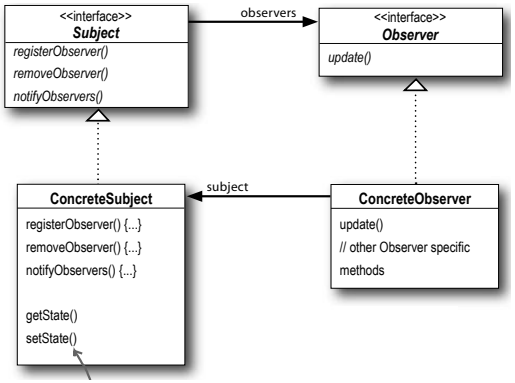
We've added an ArrayList to hold the Observers, and we create it in the constructor.

When an observer registers, we just add it to the end of the list.

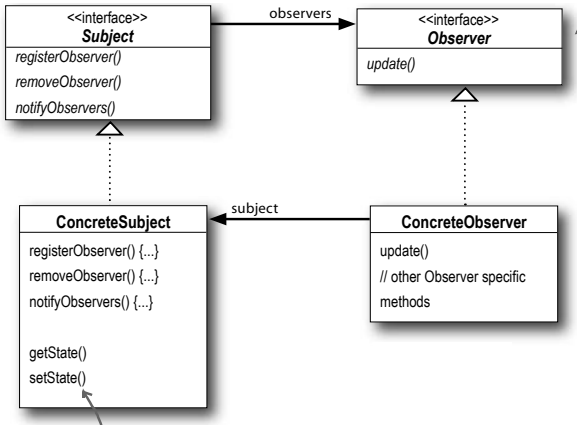
Likewise, when an observer wants to un-register, we just take it off the list.

Here's the fun part; this is where we tell all the observers about the state. Because they are all Observers, we know they all implement update(), so we know how to notify them.

Here we implement the Subject Interface.



Example Code – Weather Monitoring (4)

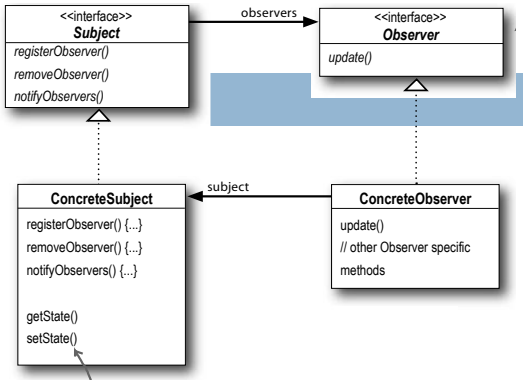


We notify the Observers when we get updated measurements from the Weather Station.

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

    public void setMeasurements(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();
    }

    // other WeatherData methods here
}
```

Okay, while we wanted to ship a nice little weather station with each book, the publisher wouldn't go for it. So, rather than reading actual weather data off a device, we're going to use this method to test our display elements. Or, for fun, you could write code to grab measurements off the web.

Example Code – Weather Monitoring (5)



This display implements Observer so it can get changes from the WeatherData object.

It also implements DisplayElement, because our API is going to require all display elements to implement this interface.

```
public class CurrentConditionsDisplay implements Observer, DisplayElement {
    private float temperature;
    private float humidity;
    private Subject weatherData;

    public CurrentConditionsDisplay(Subject weatherData) {
        this.weatherData = weatherData;
        weatherData.registerObserver(this);
    }

    public void update(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        display();
    }

    public void display() {
        System.out.println("Current conditions: " + temperature
            + "F degrees and " + humidity + "% humidity");
    }
}
```

The constructor is passed the weatherData object (the Subject) and we use it to register the display as an observer.

When update() is called, we save the temp and humidity and call display().

The display() method just prints out the most recent temp and humidity.

Example Code – Weather Monitoring (6)


```
public class WeatherStation {
    public static void main(String[] args) {
        WeatherData weatherData = new WeatherData();

        CurrentConditionsDisplay currentDisplay =
            new CurrentConditionsDisplay(weatherData);
        StatisticsDisplay statisticsDisplay = new StatisticsDisplay(weatherData);
        ForecastDisplay forecastDisplay = new ForecastDisplay(weatherData);

        weatherData.setMeasurements(80, 65, 30.4f);
        weatherData.setMeasurements(82, 70, 29.2f);
        weatherData.setMeasurements(78, 90, 29.2f);
    }
}
```

First, create the WeatherData object.

If you don't want to download the code, you can comment out these two lines and run it.

Create the three displays and pass them the WeatherData object.

Simulate new weather measurements.

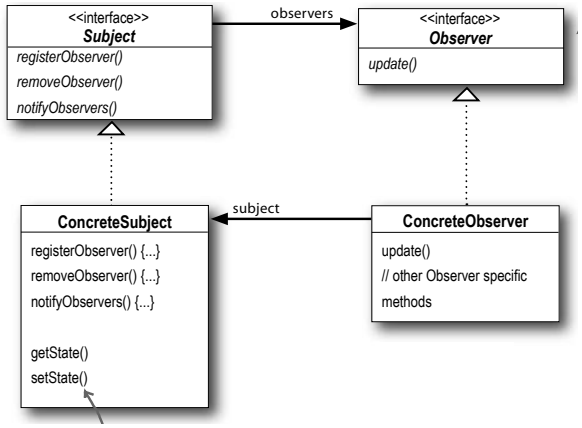
```
Current conditions: 80.0F degrees and 65.0% humidity
Avg/Max/Min temperature = 80.0/80.0/80.0
Forecast: Improving weather on the way!
Current conditions: 82.0F degrees and 70.0% humidity
Avg/Max/Min temperature = 81.0/82.0/80.0
Forecast: Watch out for cooler, rainy weather
Current conditions: 78.0F degrees and 90.0% humidity
Avg/Max/Min temperature = 80.0/82.0/78.0
Forecast: More of the same
```

Java: Observable and Observer

https://docs.oracle.com/javase/8/docs/api/java/util/Observable.html

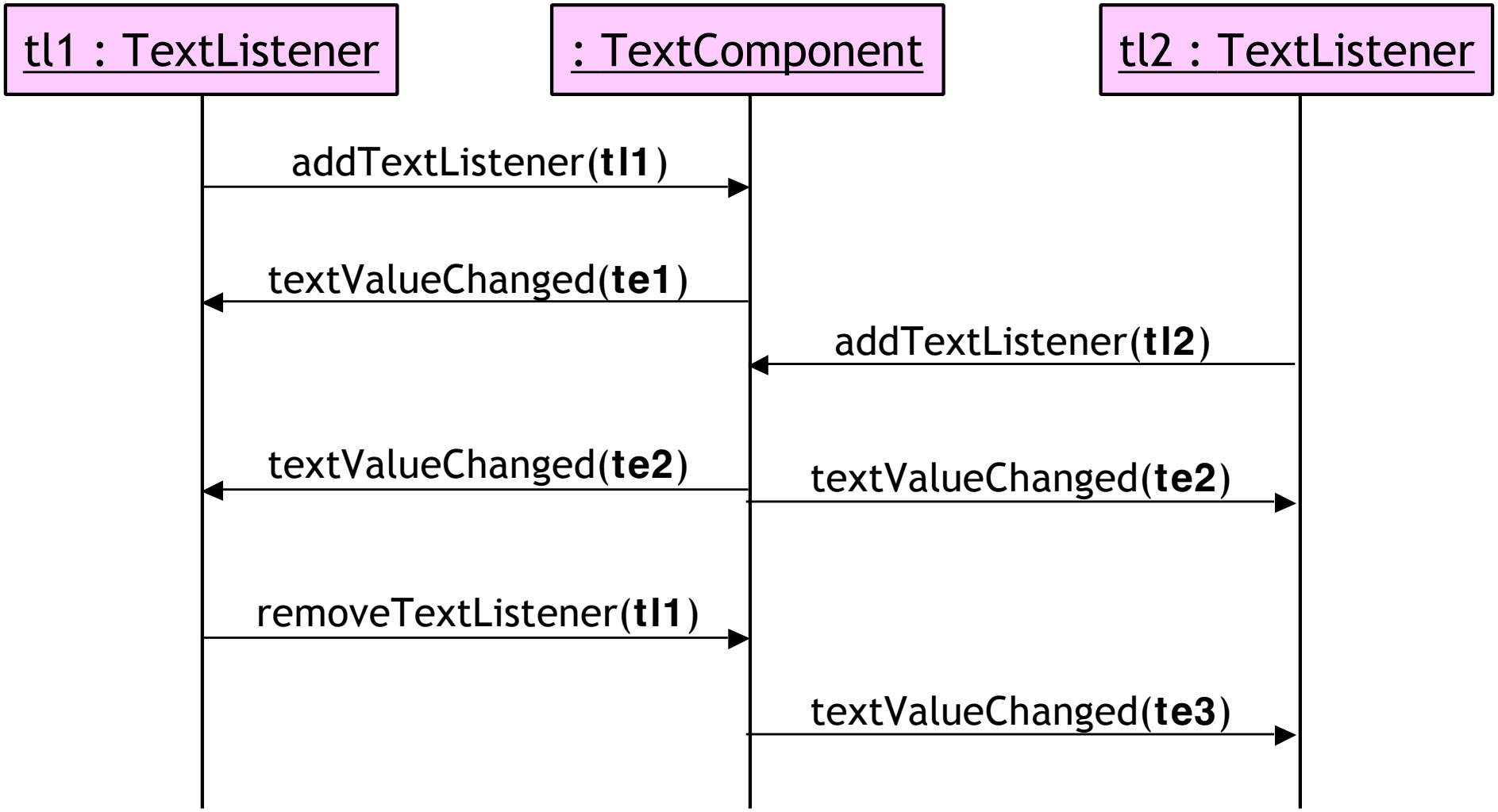
https://docs.oracle.com/javase/8/docs/api/java/util/Observer.html

```java
// Java code to demonstrate addObserver() method
import java.util.*;

// This is the observer
class Observer1 implements Observer
{
    public void update(Observable obj, Object arg)
    {
        System.out.println("Observer1: BeingObserved is updated");
    }
}

// This is class being observed
class BeingObserved extends Observable
{
    void incre()
    {
        setChanged();
        notifyObservers();
    }
}

class ObserverDemo {
    // Driver method of the program
    public static void main(String args[])
    {
        BeingObserved beingObserved = new BeingObserved();
        Observer1 observer1 = new Observer1();
        beingObserved.addObserver(observer1);
        beingObserved.incre();
    }
}
```



Observer Pattern – Example (2)

java.util.EventListener java.awt.TextListener



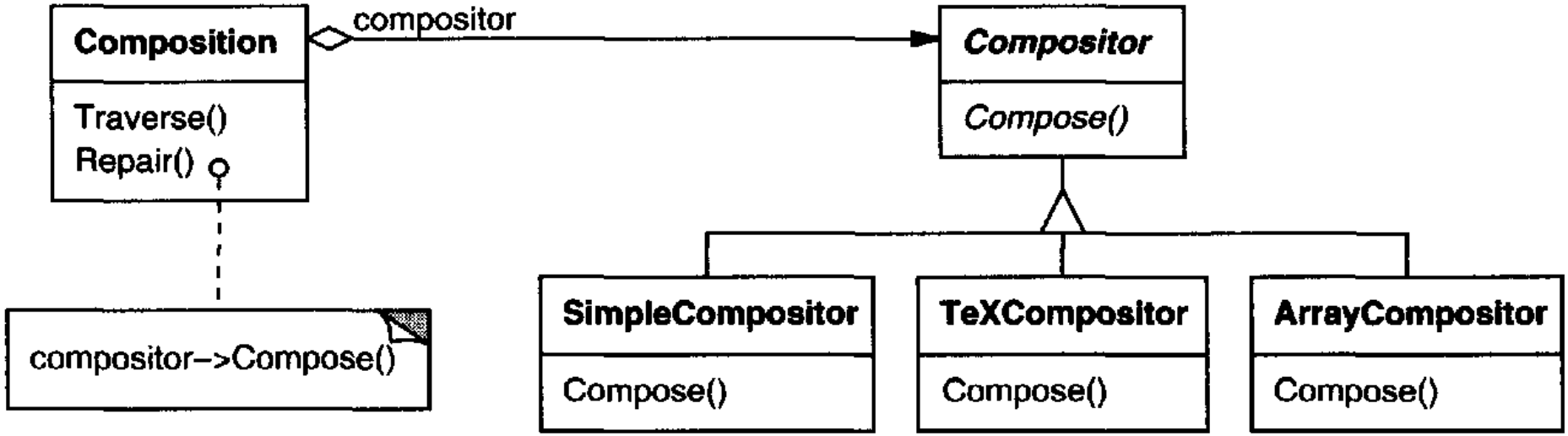
Observer Pattern: Consequences

- Abstract coupling between Subject and Observer.
- Support for broadcast communication.
- Unexpected updates.

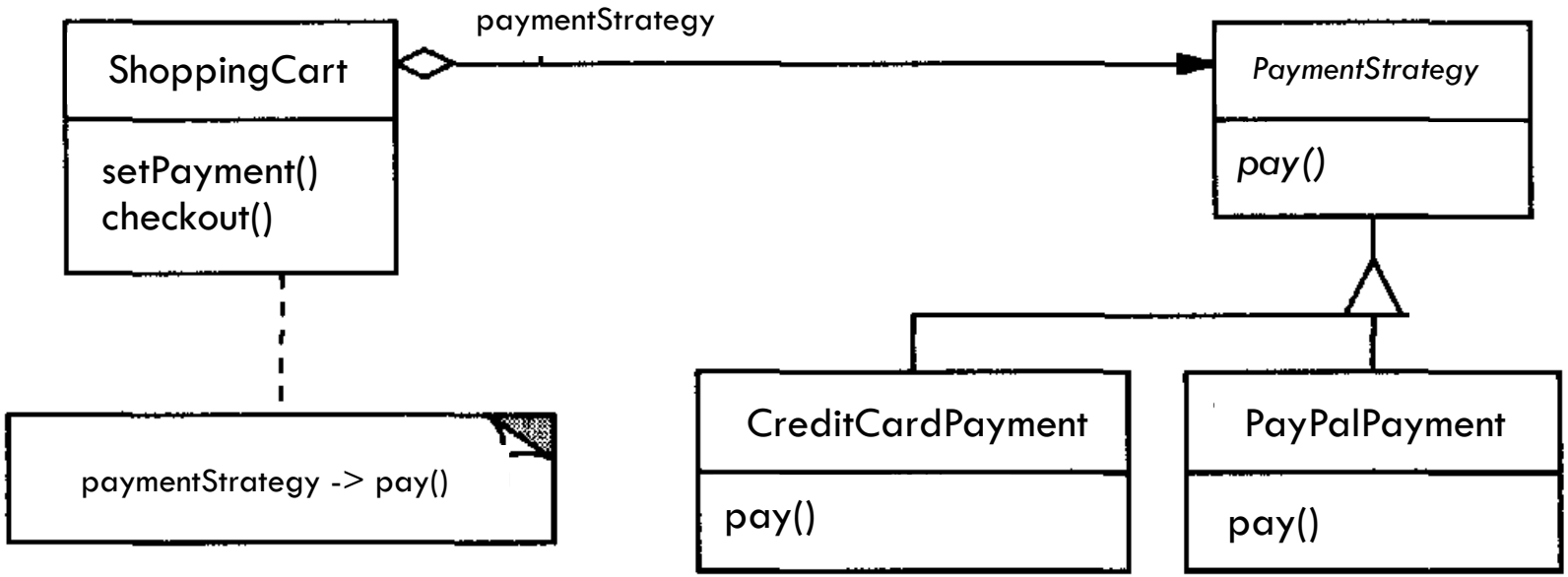
Strategy Pattern – Description (1)

- **Name:** Strategy
- **Problem:** How to encapsulate each algorithm in a family of algorithms and make them interchangeable.
- **Context:** Many algorithms exist for breaking a stream of text into lines. Hard-wiring all such algorithms into the classes that require them isn't desirable. We can avoid these problems by defining classes that encapsulate different line-breaking algorithms.
- Strategy lets the algorithm vary independently from clients that use it.

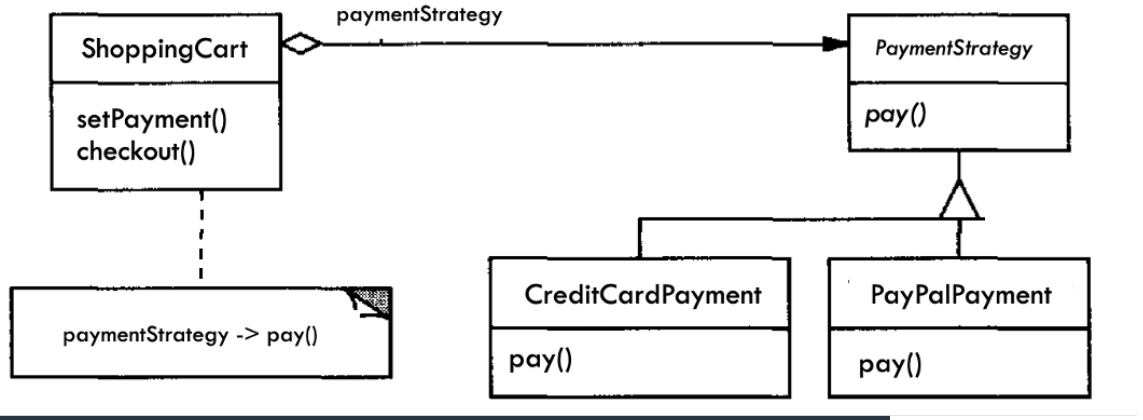
Strategy Pattern – Diagram



Strategy Pattern – Example (1)

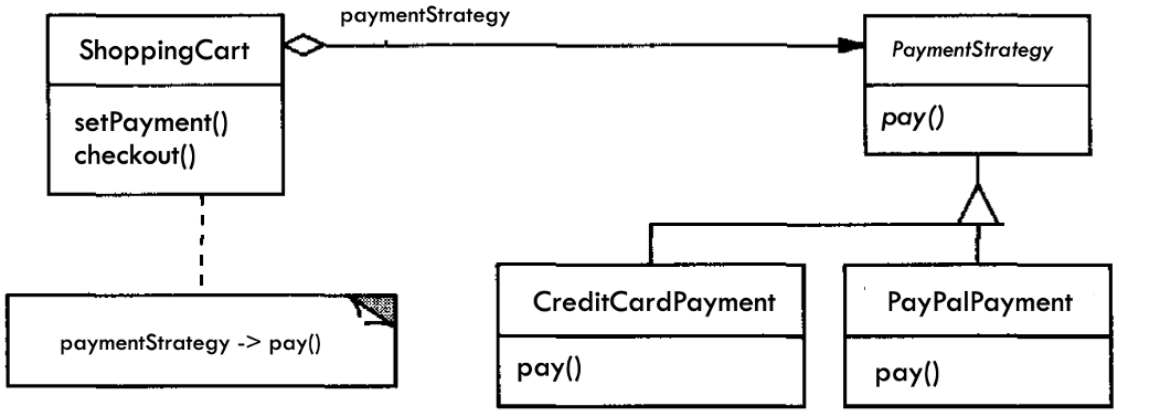


Strategy Pattern – Example (2)



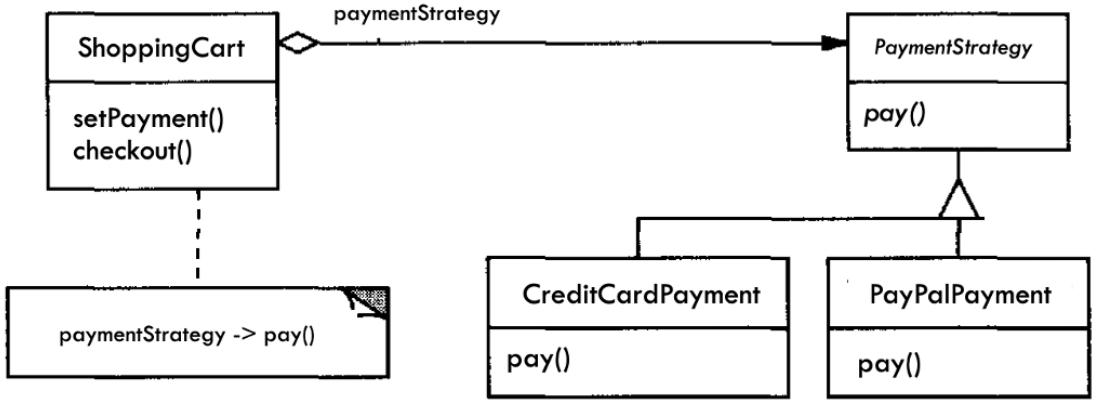

```java
// Strategy interface
interface PaymentStrategy {
    void pay(int amount);
}

// Concrete strategies
class CreditCardPayment implements PaymentStrategy {
    @Override
    public void pay(int amount) {
        System.out.println("Paid " + amount + " using credit card.");
    }
}

class PayPalPayment implements PaymentStrategy {
    @Override
    public void pay(int amount) {
        System.out.println("Paid " + amount + " using PayPal.");
    }
}
```

Strategy Pattern – Example (3)




```
// Context
class ShoppingCart {
    private PaymentStrategy paymentStrategy;

    void setPaymentStrategy(PaymentStrategy paymentStrategy) {
        this.paymentStrategy = paymentStrategy;
    }

    void checkout(int amount) {
        paymentStrategy.pay(amount);
    }
}
```

Strategy Pattern – Example (4)




```java
// Client code
public class StrategyPatternExample {
    public static void main(String[] args) {
        ShoppingCart cart = new ShoppingCart();

        // Using CreditCardPayment strategy
        cart.setPaymentStrategy(new CreditCardPayment());
        cart.checkout(100);

        // Using PayPalPayment strategy
        cart.setPaymentStrategy(new PayPalPayment());
        cart.checkout(50);
    }
}
```

```
Paid 100 using credit card.
```

```
Paid 50 using PayPal.
```

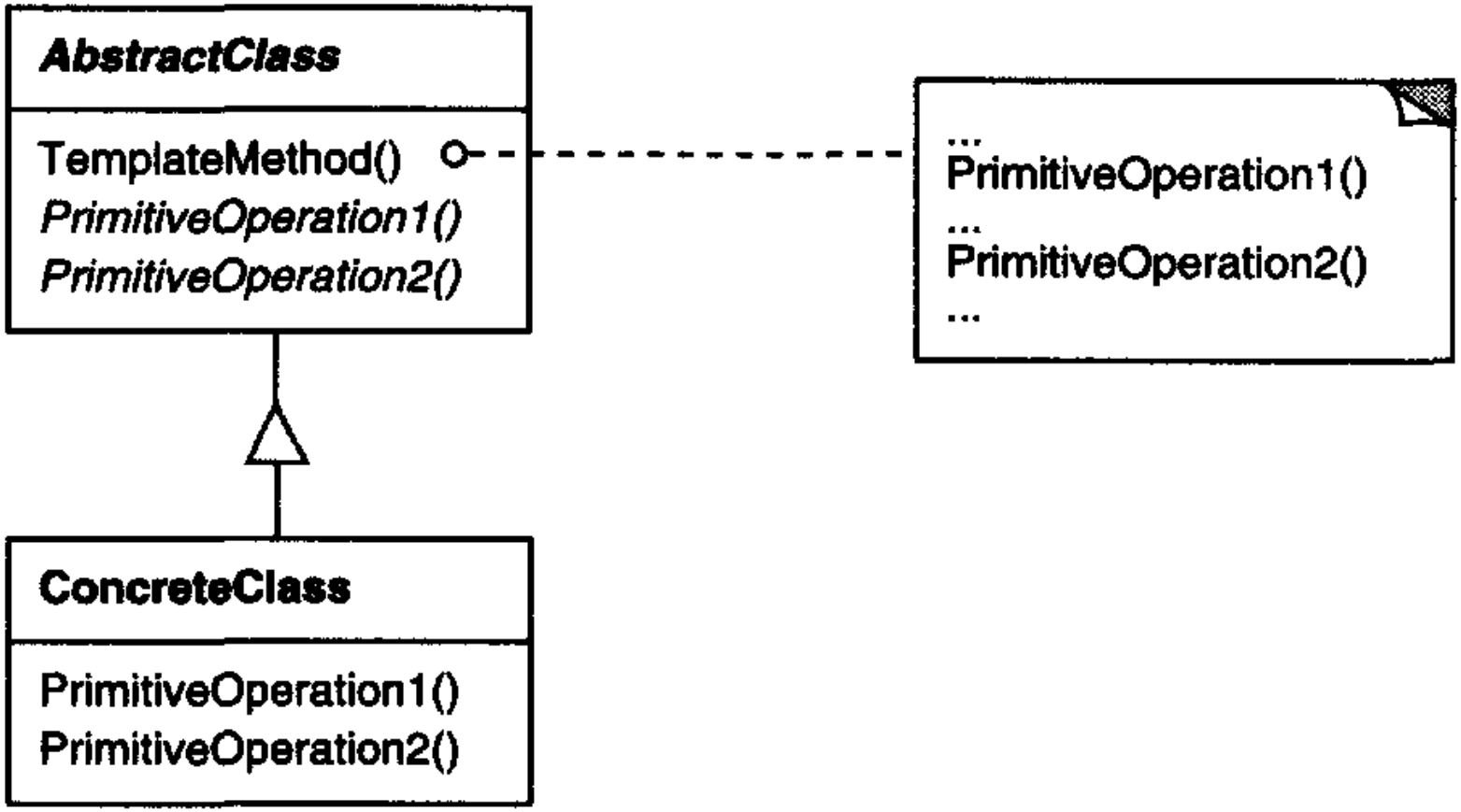
Strategy Pattern – Consequences

- Families of related algorithms.
- An alternative to subclassing.
- Strategies eliminate conditional statements.
- A choice of implementations.
- Clients must be aware of different Strategies.

Template Method Pattern – Description (1)

- **Name:** Template Method pattern
- **Problem:** How can one design a class that can *encapsulate algorithms* so that subclasses can redefine certain step of algorithms anytime they want? By applying this pattern, subclasses are allowed to redefine algorithms of a base class without changing the algorithms' structure?
- **Context:** creating a template for an algorithm.
- **Force:** The Template Method defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

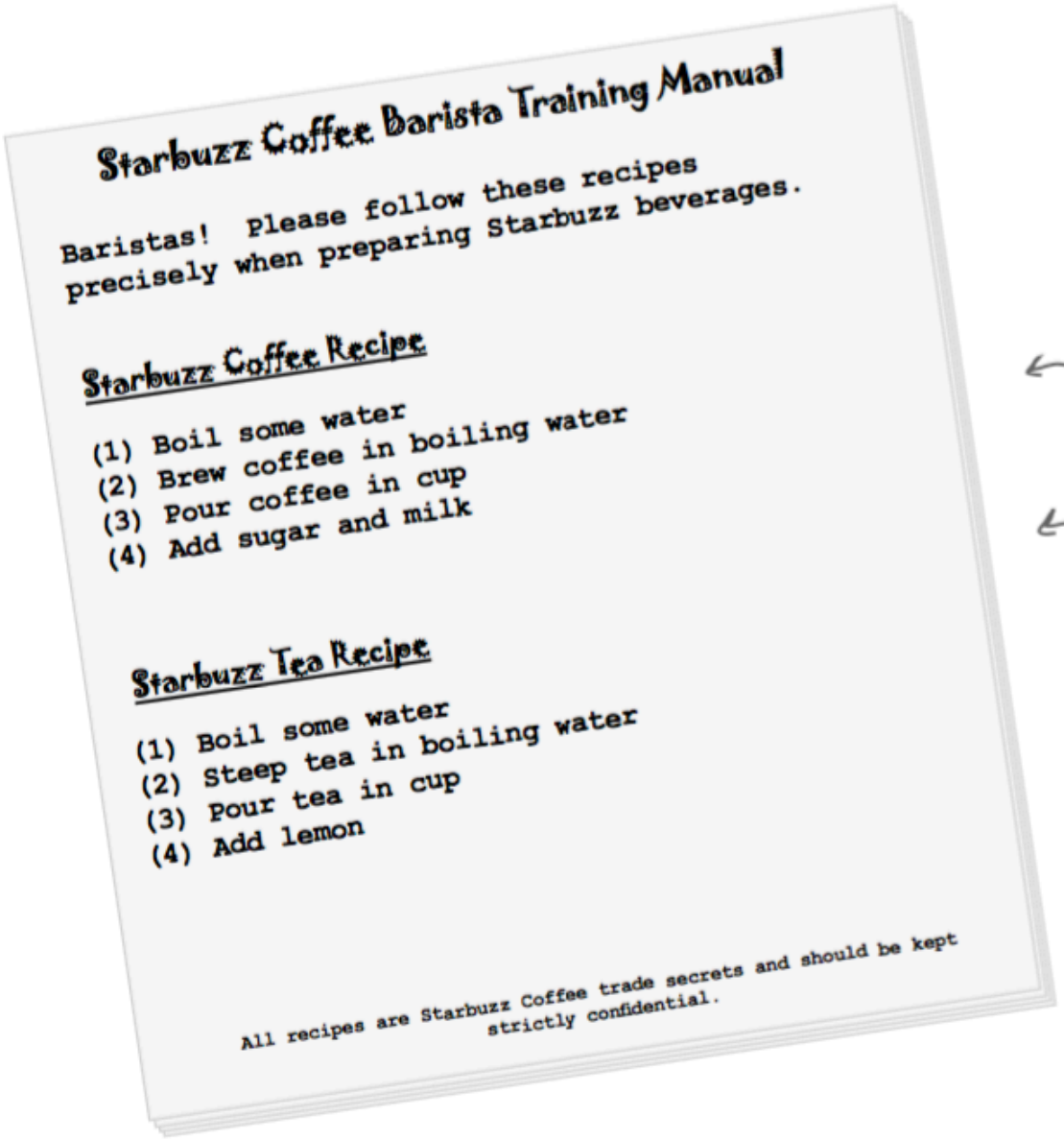
Template Method Pattern – Diagram



It's time for some more caffeine

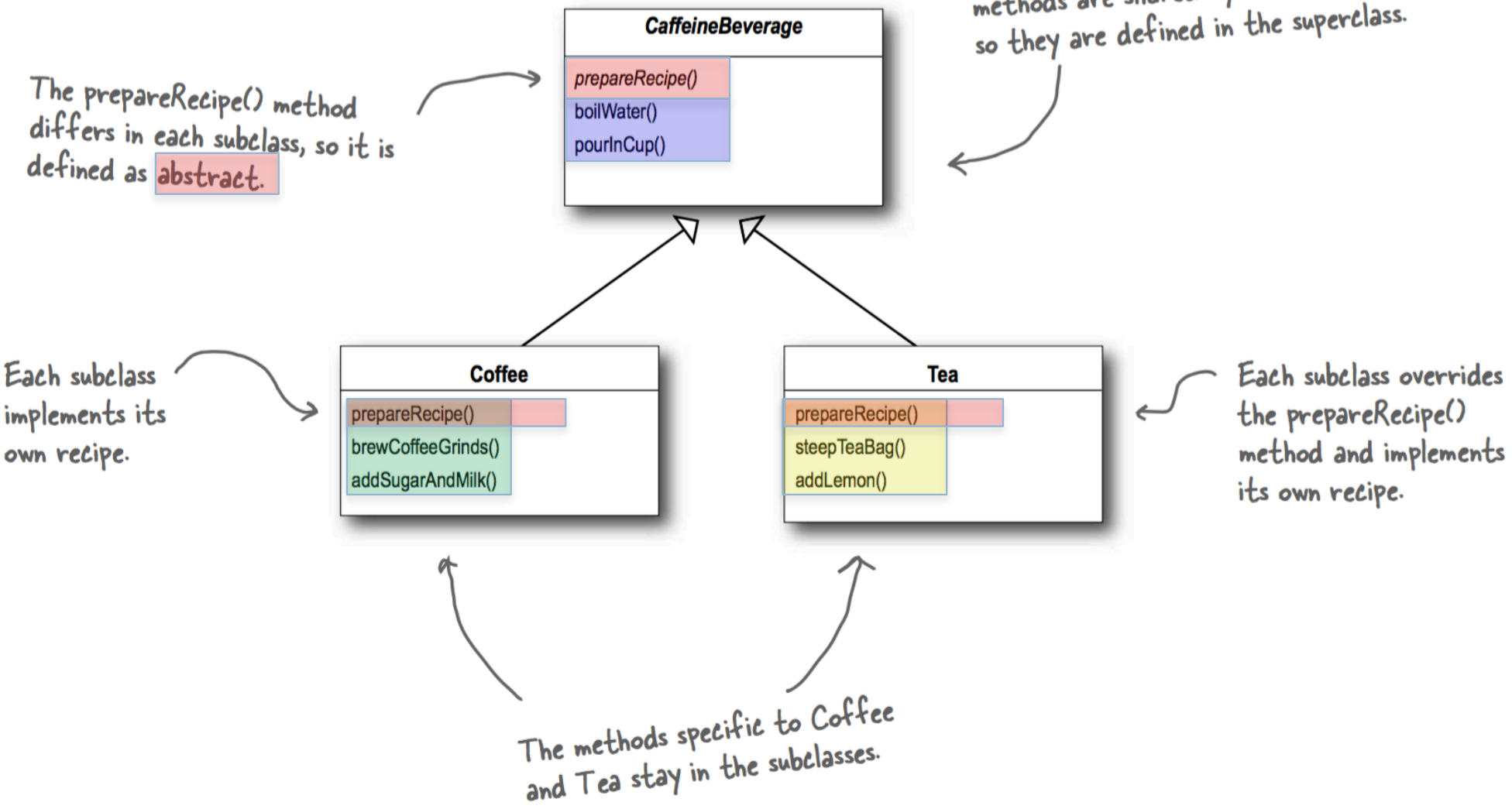
Some people can't live without their coffee; some people can't live without their tea. The common ingredient? Caffeine of course!

But there's more; tea and coffee are made in very similar ways. Let's check it out:

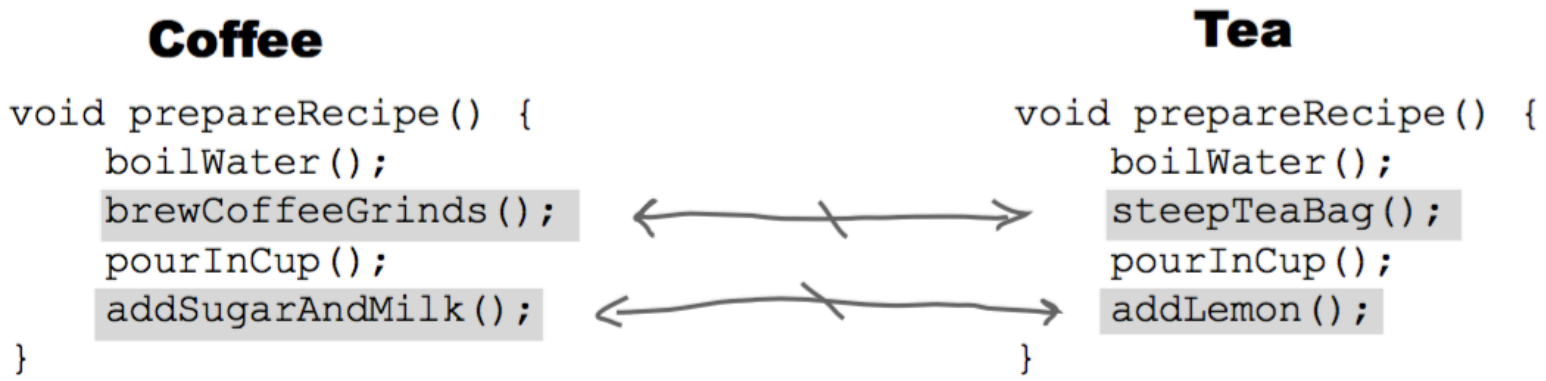


Let's abstract our Coffee and Tea.

So, can we find a way to abstract prepareRecipe() too? Yes, let's find out...



Abstracting prepareRecipe()



- Let's think through this: steeping and brewing aren't so different; they're pretty analogous. So let's make a new method name, say, brew(), and we'll use the same name whether we're brewing coffee or steeping tea.

- Likewise, adding sugar and milk is pretty much the same as adding a lemon: both
are adding condiments to the beverage. Let's also make up a new method name, addCondiments(), to handle this. So, our new prepareRecipe() method will look like this:

Abstracting prepareRecipe() - continued


```
void prepareRecipe() {
    boilWater();
    brew();
    pourInCup();
    addCondiments();
}
```

- Now we have a new prepareRecipe() method, but we need to fit it into the code. To do this we are going to start with the CaffeineBeverage superclass:

Finalizing prepareRecipe()

CaffeineBeverage is abstract, just like in the class design.

```
public abstract class CaffeineBeverage {

    final void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }

    abstract void brew();

    abstract void addCondiments();

    void boilWater() {
        System.out.println("Boiling water");
    }

    void pourInCup() {
        System.out.println("Pouring into cup");
    }
}
```

Now, the same prepareRecipe() method will be used to make both Tea and Coffee. prepareRecipe() is declared final because we don't want our subclasses to be able to override this method and change the recipe! We've generalized steps 2 and 4 to brew() the beverage and addCondiments().

Because Coffee and Tea handle these methods in different ways, they're going to have to be declared as abstract. Let the subclasses worry about that stuff!

Remember, we moved these into the CaffeineBeverage class (back in our class diagram).

Concreting abstract methods in the Coffee and Tea classes


```
public class Tea extends CaffeineBeverage {
    public void brew() {
        System.out.println("Steeping the tea");
    }
    public void addCondiments() {
        System.out.println("Adding Lemon");
    }
}
```

As in our design, Tea and Coffee now extend CaffeineBeverage.

Tea needs to define brew() and addCondiments() – the two abstract methods from Beverage.

```
public class Coffee extends CaffeineBeverage {
    public void brew() {
        System.out.println("Dripping Coffee through filter");
    }
    public void addCondiments() {
        System.out.println("Adding Sugar and Milk");
    }
}
```

Same for Coffee, except Coffee deals with coffee, and sugar and milk instead of tea bags and lemon.

Meet the Template Method Pattern

- We actually implement the Template method in the CaffeineBeverage class.

```
public abstract class CaffeineBeverage {

    void final prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }

    abstract void brew();

    abstract void addCondiments();

    void boilWater() {
        // implementation
    }

    void pourInCup() {
        // implementation
    }

}
```

prepareRecipe() is our **template method**. Why?

Because:

(1) It is a method, after all.

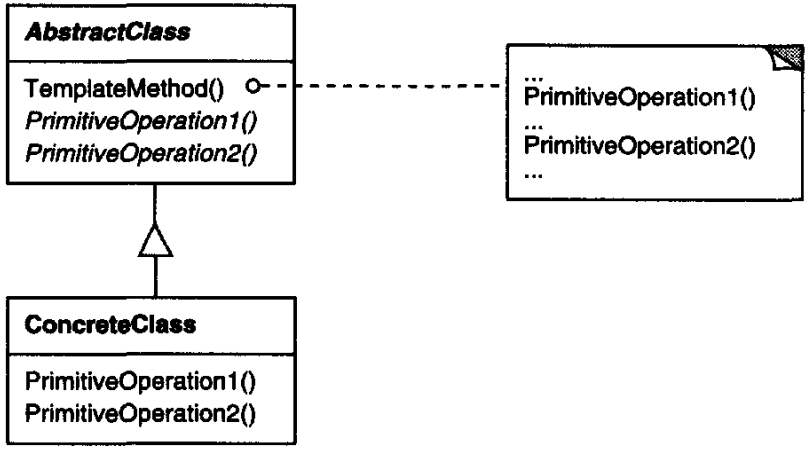
(2) It serves as a template for an algorithm, in this case, an algorithm for making caffeinated beverages.

In the template, each step of the algorithm is represented by a method.

Some methods are handled by this class...

...and some are handled by the subclass.

The methods that need to be supplied by a subclass are declared abstract.



Template Method Pattern – Consequences

- 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”
- It's important for template methods to specify which operations are hooks (may be overridden) and which are abstract operations (must be overridden).
- To reuse an abstract class effectively, subclass writers must understand which operations are designed for overriding.

How to choose a Pattern

- Is there a pattern that addresses a given problem?
- Does the pattern suggest a more appropriate solution to the problem?
- Is there a simpler solution than the pattern?
- Is the context of the pattern consistent with the context of the problem?
- Are the consequences of using the pattern acceptable?
- Are there constraints imposed by the problem context that would conflict with the use of the pattern?