Behavioral Design Patterns

CSE219, Computer Science III

Stony Brook University

http://www.cs.stonybrook.edu/~cse219

Behavioral Design Patterns

- Design patterns that identify common communication patterns between objects and realize these patterns.
  - they increase the flexibility in carrying out communication

Behavioral Design Patterns

- **Strategy pattern**: algorithms can be selected on the fly
- **Template method pattern:** describes the program skeleton of a program
- **Observer pattern:** objects register to observe an event that may be raised by another object (i.e., Event Listener)
- **Command pattern:** command objects encapsulate an action and its parameters
- **Iterator pattern:** iterators are used to access the elements of an aggregate object sequentially without exposing its underlying representation
- **State pattern:** a clean way for an object to partially change its type at runtime

Common Design Patterns

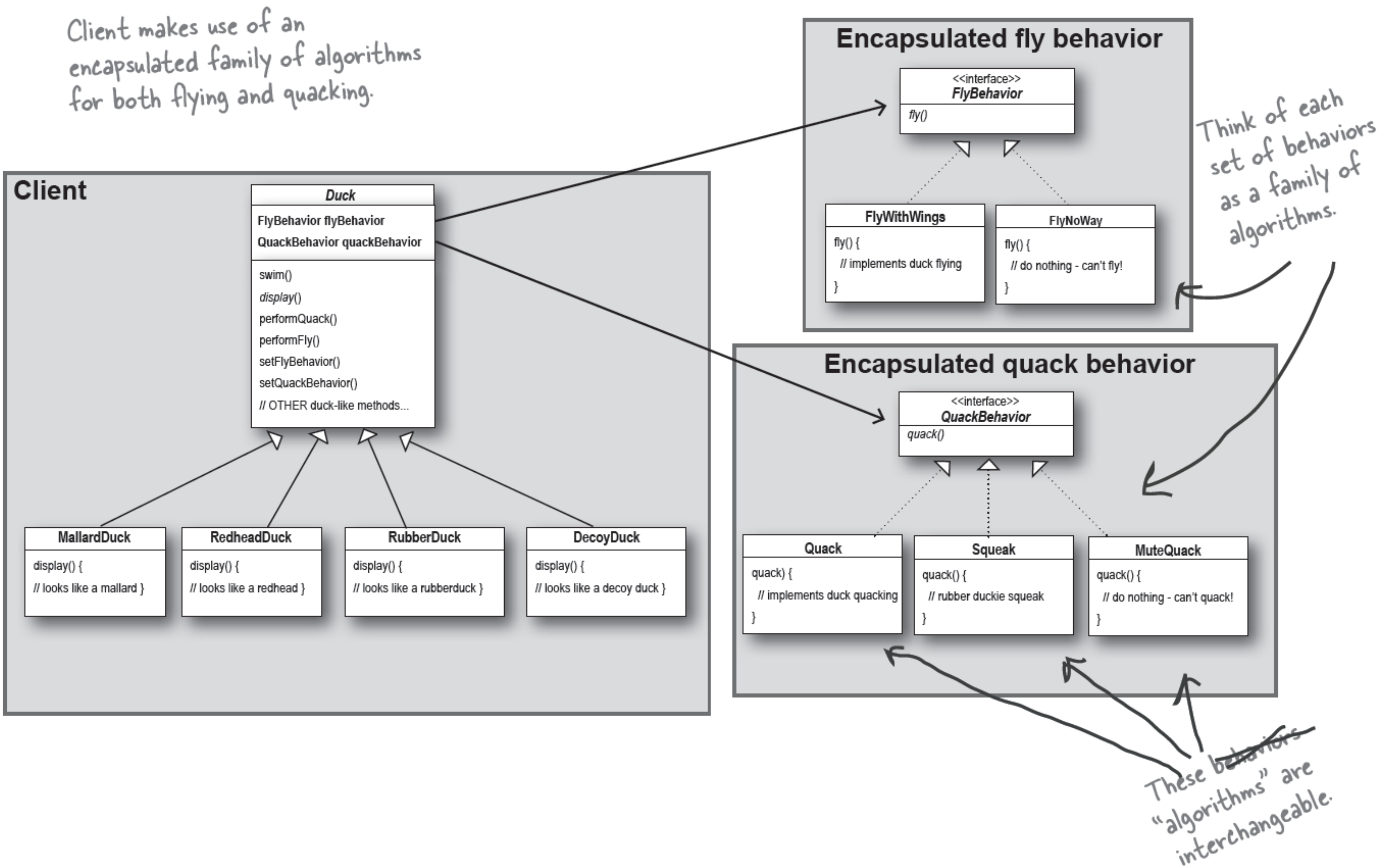
Creational	Structural	Behavioral
• Factory • Singleton • Builder • Prototype	• Decorator • Adapter • Facade • Flyweight • Bridge	• Strategy • Template • Observer • Command • Iterator • State

Textbook: Head First Design Patterns

The Strategy Pattern

- Defines a family of algorithms, encapsulates each one, and makes them interchangeable
  - lets the algorithm vary independently from the clients that use them
- An algorithm in a box
  - place essential steps for an algorithm in a strategy interface
    - different methods represent different parts of the algorithm
  - **classes implementing this interface customize methods**
- Classes can be composed (HAS-A) of the interface type
  - the interface type is the apparent type
  - the actual type can be determined at run-time

The Strategy Pattern Example



```
abstract class Duck {
         FlyBehaviour flyBehaviour;
         QuackBehaviour quackBehaviour;
         public void fly() {
                  this.flyBehaviour.fly();
         }
         public void quack() {
                  this.quackBehaviour.walk();
         }
}
interface FlyBehaviour {
        void fly();
}
class FlyWithWings implements FlyBehaviour {
        public void fly() {
                System.out.println("Flying with wings ...");
        }
}
class FlyNoWay implements FlyBehaviour {
        public void fly() {
                System.out.println("Cannot Fly.");
        }
}
interface QuackBehaviour {
        void quack();
}
class Quack implements QuackBehaviour {
        public void quack() {
                System.out.println("Quack quack ...");
        }
}
```

```
class RubberQuack implements QuackBehaviour {
        public void quack() {
                System.out.println("Quick quack ...");
        }
}
class MallardDuck extends Duck{
         public MallardDuck() {
                this.flyBehaviour = new FlyWithWings();
                this.quackBehaviour = new Quack();
        }
}
class RubberDuck extends Duck{
         public MallardDuck() {
                this.flyBehaviour = new FlyNoWay();
                this.quackBehaviour = new RubberQuack();
        }
}
public class Main {
        public static void main(String[] args) {
                Duck duck = new MallardDuck();
                duck.fly();
                duck.quack();

                duck = new RubberDuck();
                duck.fly();
                duck.quack();
        }
}
```

The Strategy Pattern

- The HAS-A relationship can be better than IS-A
  - In the example, each duck has a FlyBehavior and a QuackBehavior to which it delegates flying and quacking.
- We are using composition instead of inheriting their behavior: the ducks get their behavior by being composed with the right behavior object.
  - It also allows to ***change the behavior at runtime*** as long as the object implements the correct behavior interface.

```
/** Calculator using the strategy pattern
*
* The classes that implement a concrete strategy should implement Strategy
* The Context class uses this to call the concrete strategy. */
interface Strategy {
    int execute(int a, int b);
}

/** Implements the algorithm using the strategy interface */
class Add implements Strategy {
    public int execute(int a, int b) {
        System.out.println("Called Add's execute()");
        return a + b;  // Do an addition with a and b
    }
}

class Subtract implements Strategy {
    public int execute(int a, int b) {
        System.out.println("Called Subtract's execute()");
        return a - b;  // Do a subtraction with a and b
    }
}

class Multiply implements Strategy {
    public int execute(int a, int b) {
        System.out.println("Called Multiply's execute()");
        return a * b;   // Do a multiplication with a and b
    }
}
```

```java
// Configured with a ConcreteStrategy object and maintains
// a reference to a Strategy object
class Context {
    private Strategy strategy;
    public Context(Strategy strategy) {
        this.strategy = strategy;
    }
    public int executeStrategy(int a, int b) {
        return this.strategy.execute(a, b);
    }
}

public class StrategyExample {
    public static void main(String[] args) {
        Context context;
        // Three contexts following different strategies
        context = new Context(new Add());
        int resultA = context.executeStrategy(3,4);

        context = new Context(new Subtract());
        int resultB = context.executeStrategy(3,4);

        context = new Context(new Multiply());
        int resultC = context.executeStrategy(3,4);

        System.out.println("Result A : " + resultA );
        System.out.println("Result B : " + resultB );
        System.out.println("Result C : " + resultC );
    }
}
```

Strategy Patterns in the Java API

- **`LayoutManager`**s interface
  - describes how to arrange components:
    - resizing
    - components added to or removed from the container.
    - size and position the components it manages.

```
public interface LayoutManager {
  void addLayoutComponent(String name, Component comp);
  void removeLayoutComponent(Component comp);
  Dimension preferredLayoutSize(Container parent);
  Dimension minimumLayoutSize(Container parent);
  void layoutContainer(Container parent);
}
```

Containers use LayoutManagers

- Composition / at runtime:

```
public class Container extends Component {
  private List<Component> component;
  private LayoutManager layoutMgr;
  …
  public LayoutManager getLayout() {
    return layoutMgr;
  }
  public void setLayout(LayoutManager mgr) {
    layoutMgr = mgr;
  }
}
```

Dynamic interchange is great

- Interchangeable layout algorithms in a box:

```
public void initGUI() {
  northPanel = new JPanel();
  northPanel.setLayout(new BorderLayout());
  northOfNorthPanel = new Jpanel();
  northPanel.add(northOfNorthPanel, "NORTH");
  northPanelOfNorthPanel.add(cutButton);
  northPanelOfNorthPanel.add(pasteButton);
  ...
}
```

Common Design Patterns

Creational	Structural	Behavioral
• Factory • Singleton • Builder • Prototype	• Decorator • Adapter • Facade • Flyweight • Bridge	• Strategy • Template • Observer • Command • Iterator • State

Textbook: Head First Design Patterns

Template Method Pattern

- A template for an algorithm
- Defines the skeleton of an algorithm in a method, deferring some steps to subclasses.
- Lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Template Method Pattern

- Example: Starbuzz beverages.
  - Coffee:
    - (1) Boil some water
    - (2) Brew coffee in boiling water
    - (3) Pour coffee in cup
    - (4) Add sugar and milk
  - Tea: // eliminate code duplication
    - (1) Boil some water
    - (2) Steep tea in boiling water
    - (3) Pour tea in cup
    - (4) Add lemon

Template Method Pattern

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).

Template Method Pattern

```
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.

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

```
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");
    }
}
```

We've recognized that the two recipes are essentially the same, although some of the steps require different implementations. So we've generalized the recipe and placed it in the base class.

Tea

1. Boil some water
2. Steep the teabag in the water
3. Pour tea in a cup
4. Add lemon

Coffee

1. Boil some water
2. Brew the coffee grinds
3. Pour coffee in a cup
4. Add sugar and milk

generalize

generalize

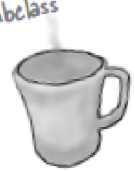
Caffeine Beverage

1. Boil some water
2. Brew
3. Pour beverage in a cup
4. Add condiments

relies on subclass for some steps

relies on subclass for some steps

Tea subclass



2. Steep the teabag in the water
4. Add lemon

Coffee subclass

2. Brew the coffee grinds
4. Add sugar and milk

Caffeine Beverage knows and controls the steps of the recipe, and performs steps 1 and 3 itself, but relies on Tea or Coffee to do steps 2 and 4.

```
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.

```
boilWater();
brew();
pourInCup();
addCondiments();
```

The prepareRecipe() method controls the algorithm, no one can change this, and it counts on subclasses to provide some or all of the implementation.

1. Okay, first we need a Tea object...

```
Tea myTea = new Tea();
```

2. Then we call the template method:

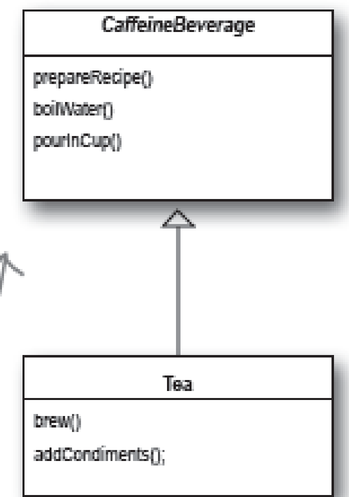
```
myTea.prepareRecipe();
```

which follows the algorithm for making caffeine beverages...

3. First we boil water:

```
boilWater();
```

which happens in CaffeineBeverage.



4. Next we need to brew the tea, which only the subclass knows how to do:

```
brew();
```

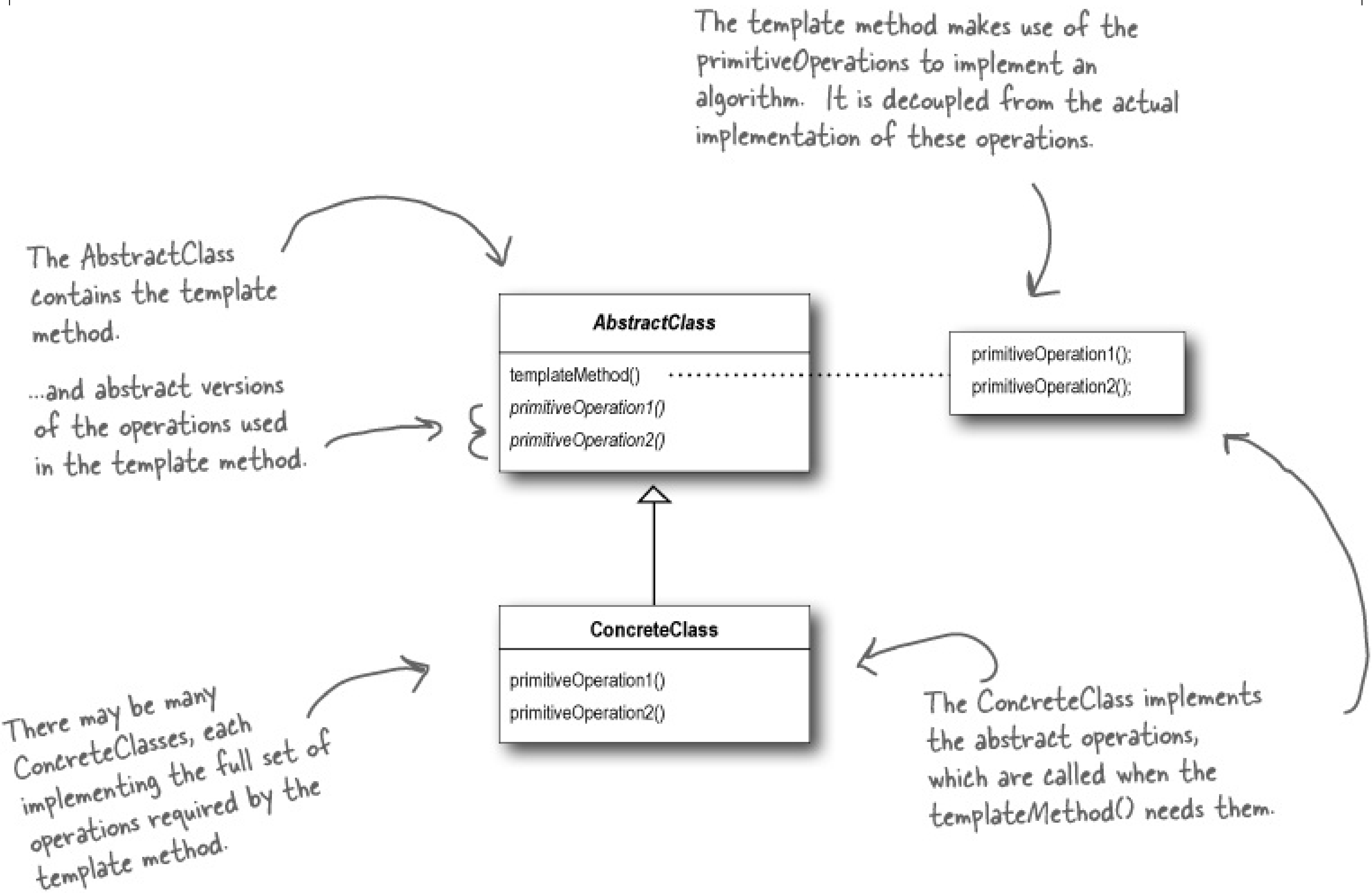
5. Now we pour the tea in the cup; this is the same for all beverages so it happens in CaffeineBeverage:

```
pourInCup();
```

6. Finally, we add the condiments, which are specific to each beverage, so the subclass implements this:

```
addCondiments();
```

Template Method Pattern



Template Method Pattern

We've changed the templateMethod() to include a new method call.

```
abstract class AbstractClass {

    final void templateMethod() {
        primitiveOperation1();
        primitiveOperation2();
        concreteOperation();
        hook();
    }

    abstract void primitiveOperation1();

    abstract void primitiveOperation2();

    final void concreteOperation() {
        // implementation here
    }

    void hook() {}

}
```

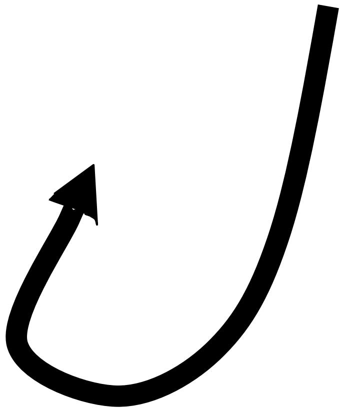
We still have our primitive methods; these are abstract and implemented by concrete subclasses.

A concrete operation is defined in the abstract class. This one is declared final so that subclasses can't override it. It may be used in the template method directly, or used by subclasses.

A concrete method, but it does nothing!

We can also have concrete methods that do nothing by default; we call these "hooks." Subclasses are free to override these but don't have to. We're going to see how these are useful on the next page.

What's a hook?



- A type of a concrete method
- Declared in the abstract class
  - only given an empty or default implementation
- Gives subclasses the ability to "hook into" the algorithm at various points, if they wish
  - a subclass is also free to ignore the hook.

```java
public abstract class CaffeineBeverageWithHook {

    void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        if (customerWantsCondiments()) {
            addCondiments();
        }
    }

    abstract void brew();

    abstract void addCondiments();

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

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

    boolean customerWantsCondiments() {
        return true;
    }
}
```

We've added a little conditional statement that bases its success on a concrete method, customerWantsCondiments(). If the customer WANTS condiments, only then do we call addCondiments().

Here we've defined a method with a (mostly) empty default implementation. This method just returns true and does nothing else.

This is a **hook** because the subclass can override this method, but doesn't have to.

```java
abstract class CaffeineBeverage {                    // Complete program
        public final void prepareRecipe() {
                boilWater();
                brew();
                pourInCup();
                addCondiments();
                hook();
        }
        void boilWater() {
                System.out.println("Boiling water ...");
                hookWaterHasBoiled();
        }
        void pourInCup() {
                System.out.println("Pour fluid in cup.");
        }
        abstract void brew();
        abstract void addCondiments();
        void hook(){}
        void hookWaterHasBoiled(){}
}
class Coffee extends CaffeineBeverage {
        @Override
        void brew() {
                System.out.println("Brew/filter the coffee ...");
        }
        @Override
        void addCondiments() {
                System.out.println("Add milk and sugar to the coffeee.");
        }
```

```
        @Override
        void hook() {
                System.out.println("Did you like the coffee?");
        }
}
class Tea extends CaffeineBeverage {
        @Override
        void brew() {
                System.out.println("Put teabag in the water.");
        }
        @Override
        void addCondiments() {
                System.out.println("Add sugar and honey to the tea");
        }
        @Override
        void hookWaterHasBoiled() {
                System.out.println("The tea water is boiling!!!");
        }
}
public class Main {
        public static void main(String[] args) {
                CaffeineBeverage beverage = new Coffee();
                beverage.prepareRecipe();

                beverage = new Tea();
                beverage.prepareRecipe();
        }
}
```

Strategy vs. Template Method

- What’s the difference?
- Strategy
  - encapsulate interchangeable behaviors and use delegation to decide which behavior to use
- Template Method
  - subclasses decide how to implement steps in an algorithm

```java
/** An abstract class that is common to several games in which players play
 * against the others, but only one is playing at a given time.
 */
abstract class Game {
    protected int playersCount;
    abstract void initializeGame();
    abstract void makePlay(int player);
    abstract boolean endOfGame();
    abstract void printWinner();
    /* A template method: */
    public final void playOneGame(int playersCount) {
        this.playersCount = playersCount;
        initializeGame();
        int j = 0;
        while (!endOfGame()) {
            makePlay(j);
            j = (j + 1) % playersCount;
        }
        printWinner();
    }
}
//Now we can extend this class in order
//to implement actual games:
class Monopoly extends Game {
    /* Implementation of necessary concrete methods */
    void initializeGame() {
        // Initialize players
        // Initialize money
    }
```

Play template example

```
    void makePlay(int player) {
        // Process one turn of player
    }
    boolean endOfGame() {
        // Return true if game is over
        // according to Monopoly rules
    }
    void printWinner() {
        // Display who won
    }
    /* Specific declarations for the Monopoly game. */
}
class Chess extends Game {
    /* Implementation of necessary concrete methods */
    void initializeGame() {
        // Initialize players
        // Put the pieces on the board
    }
    void makePlay(int player) {
        // Process a turn for the player
    }
    boolean endOfGame() {
        // Return true if in Checkmate or
        // Stalemate has been reached
    }
    void printWinner() {
        // Display the winning player
    }
    /* Specific declarations for the chess game. */
}
```

Common Design Patterns

Creational	Structural	Behavioral
• Factory • Singleton • Builder • Prototype	• Decorator • Adapter • Facade • Flyweight • Bridge	• Strategy • Template • Observer • Command • Iterator • State

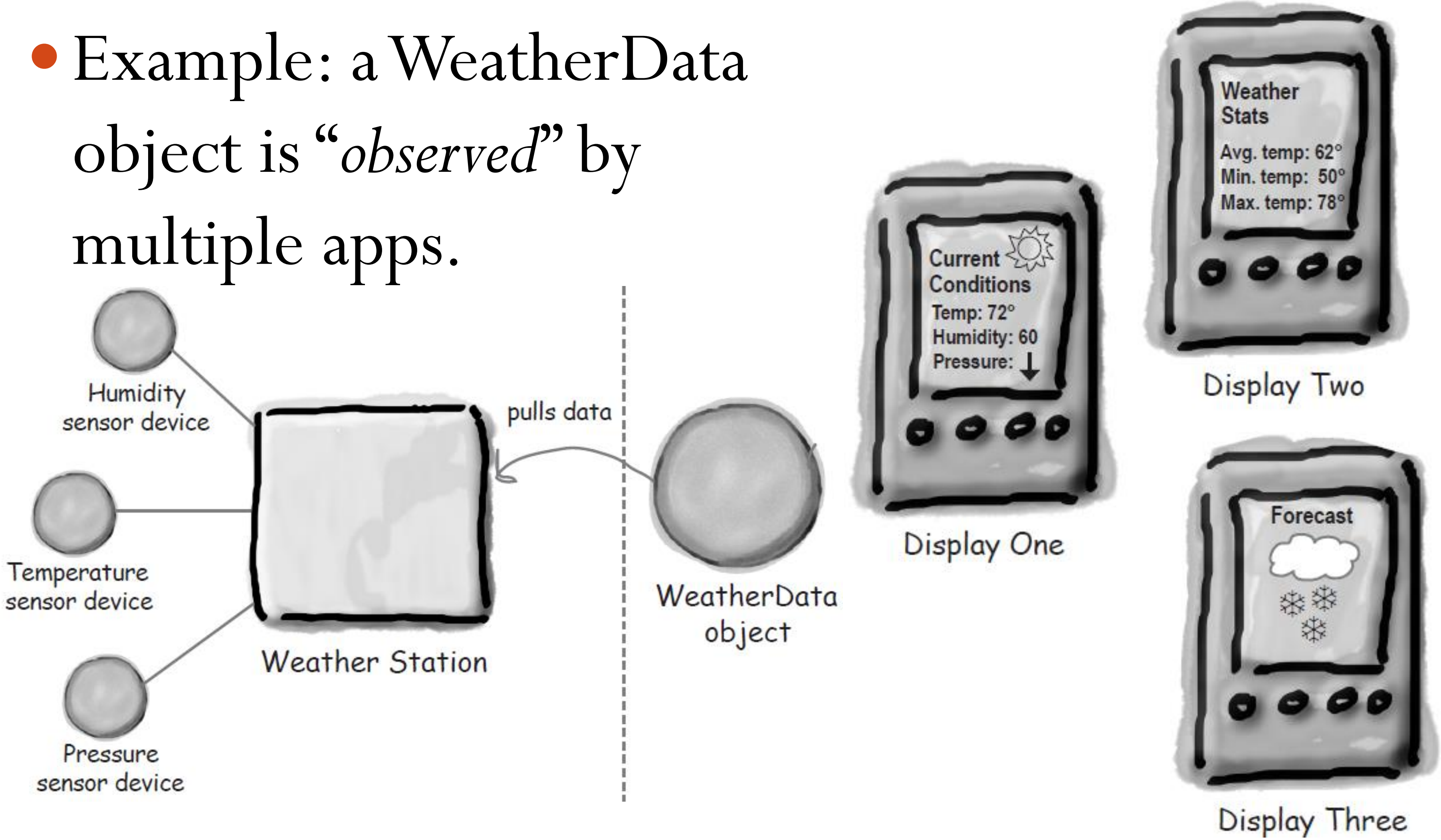
Textbook: Head First Design Patterns

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
- Where have we seen this?
  - in our GUI
  - State Manager class maintains application's state
    - call methods to change app's state
    - app's state change forces update of GUI

The Observer Pattern

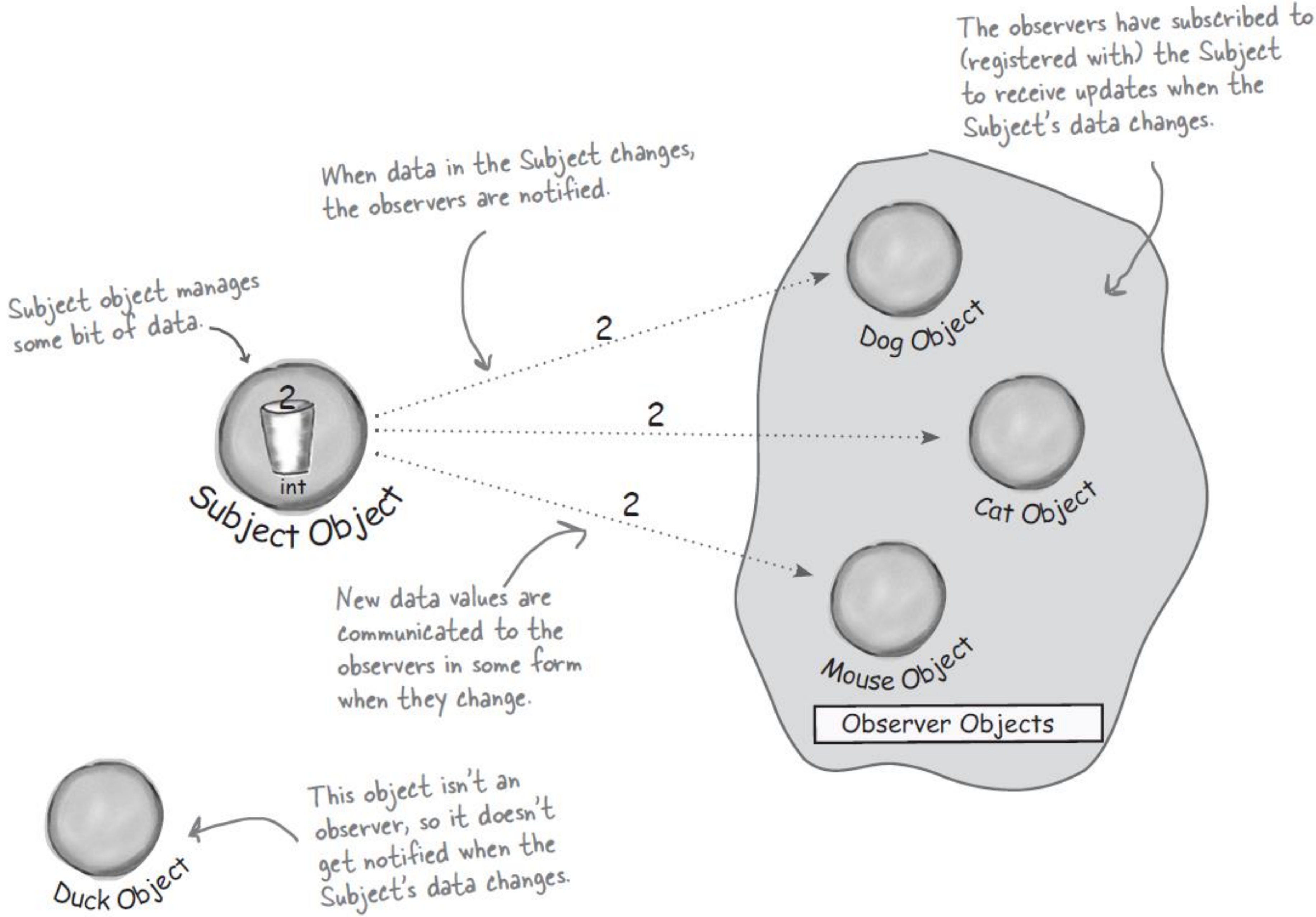
- Example: a WeatherData object is “*observed*” by multiple apps.



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

class Observable {
        List<Observer> observers = new ArrayList<Observer>();
        void publish(Object data) {
                for (Observer obs : this.observers) {
                        obs.update(this,data);
                }
        }
        public void subscribe(Observer obs) {
                this.observers.add(obs);
        }
        public void unsubscribe(Observer obs) {
                this.observers.remove(obs);
        }
}

interface Observer {
        void update(Observable observable, Object object);
}

class WeatherData extends Observable {
        int temperature;
        public void update(int newTemperature) {
                this.temperature = newTemperature;
                this.publish(new Integer(temperature));
        }
}
```

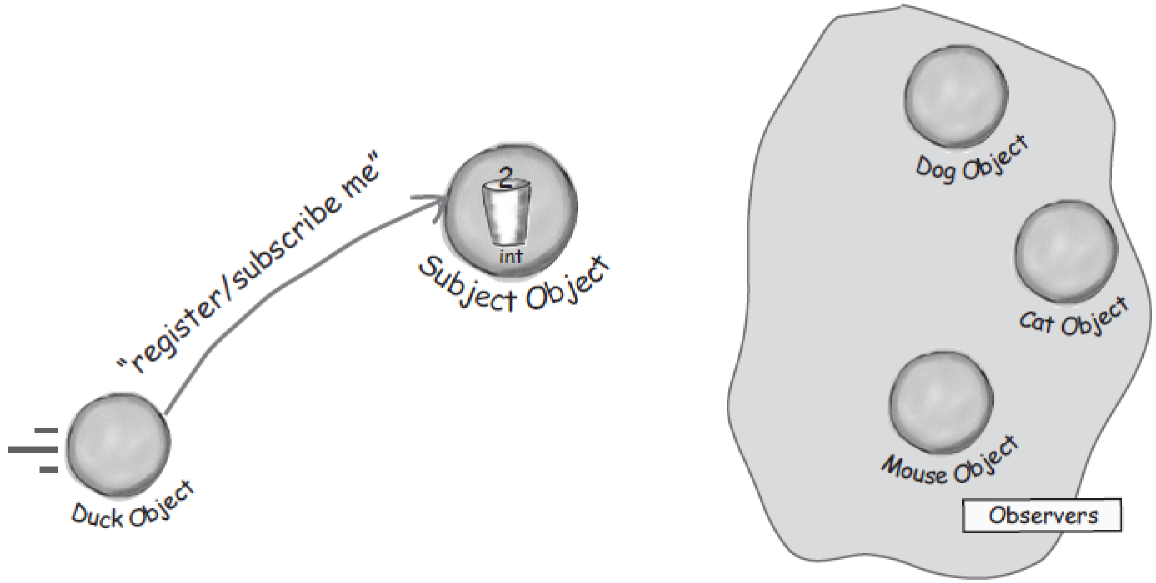
```java
class WeatherStation implements Observer {
        public WeatherStation(WeatherData obs) {
                obs.subscribe(this);
        }
        @Override
        public void update(Observable observable, Object object) {
                if (observable instanceof WeatherData
                                && object instanceof Integer) {
                        Integer temp = (Integer) object;
                        System.out.println("Station: temperature is " + temp);
                }
        }
}

public class Main {
        public static void main(String[] args) {
                WeatherData data = new WeatherData();

                WeatherStation station1 = new WeatherStation(data);
                WeatherStation station2 = new WeatherStation(data);
                data.update(40);

                data.unsubscribe(station1);
                System.out.println("deleted one");

                data.update(35);
        }
}
```

Publishers + Subscribers = Observer Pattern



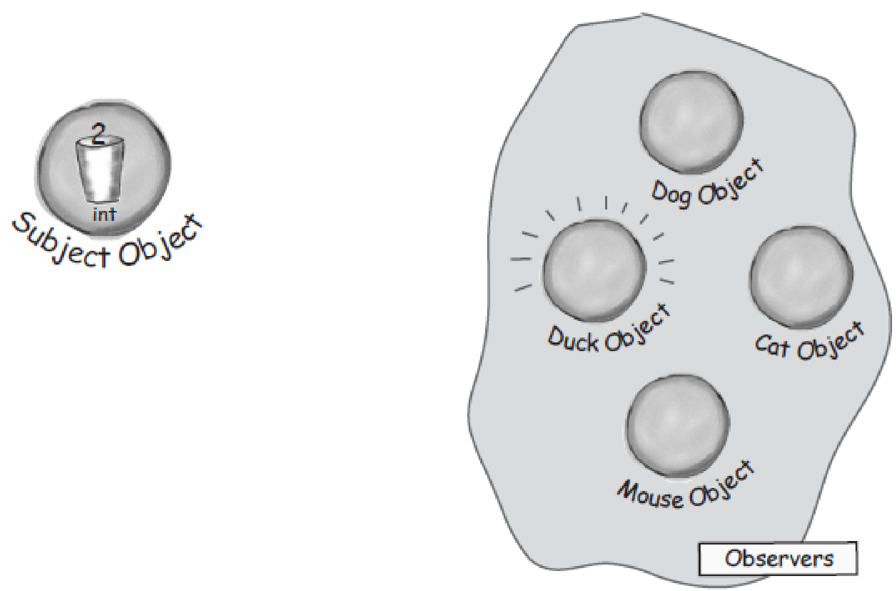
A Duck object comes along and tells the Subject that it wants to become an observer.

Duck really wants in on the action; those ints Subject is sending out whenever its state changes look pretty interesting...



The Duck object is now an official observer.

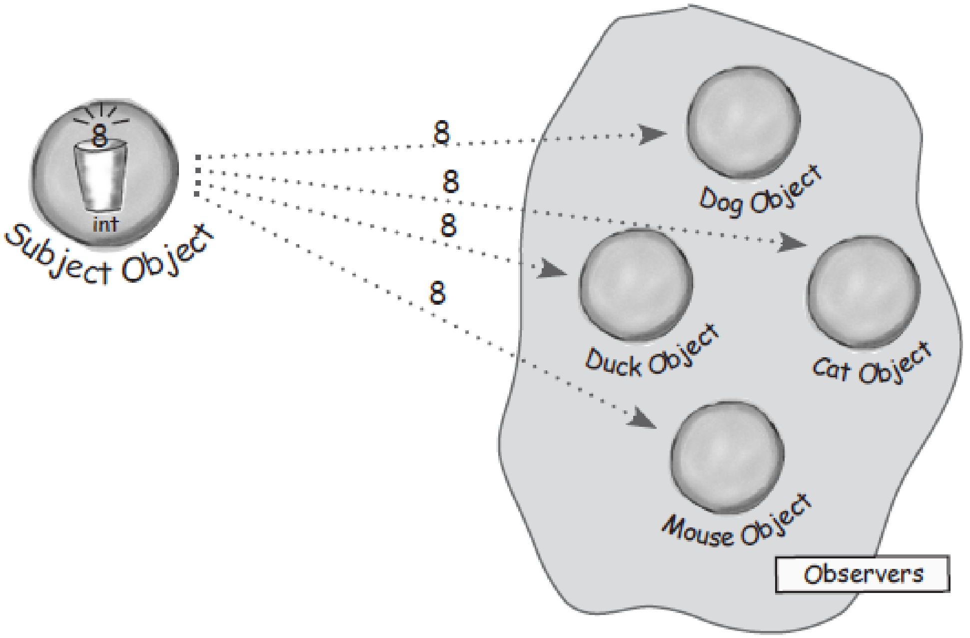
Duck is psyched... he's on the list and is waiting with great anticipation for the next notification so he can get an int.



The Observer Pattern

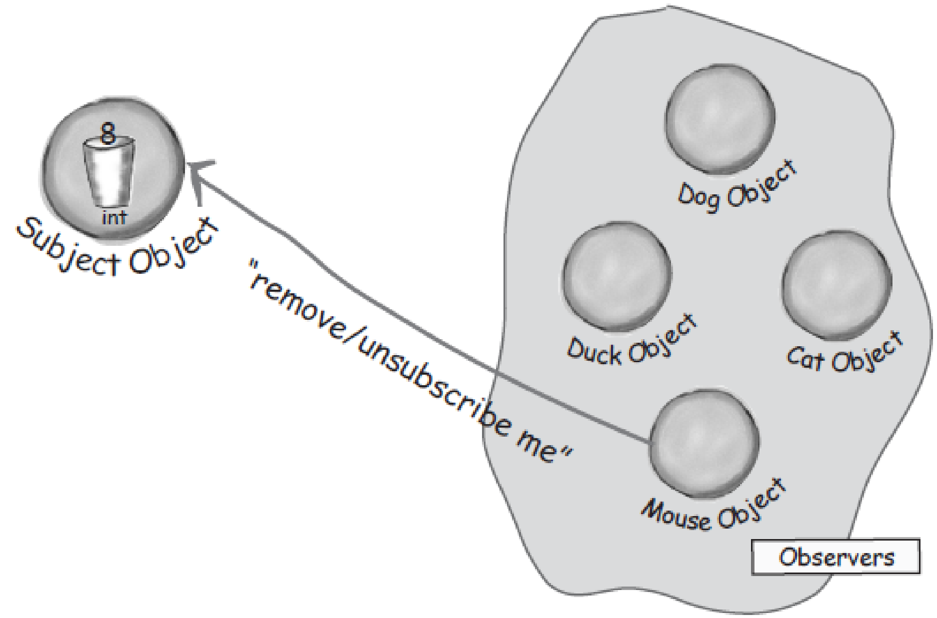
The Subject gets a new data value!

Now Duck and all the rest of the observers get a notification that the Subject has changed.



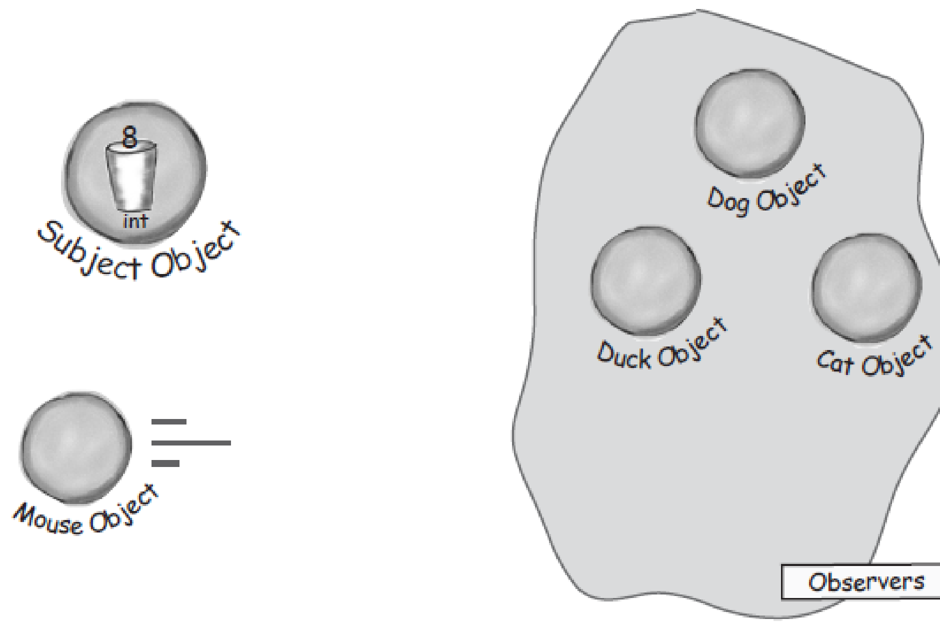
The Mouse object asks to be removed as an observer.

The Mouse object has been getting ints for ages and is tired of it, so it decides it's time to stop being an observer.



Mouse is outta here!

The Subject acknowledges the Mouse's request and removes it from the set of observers.

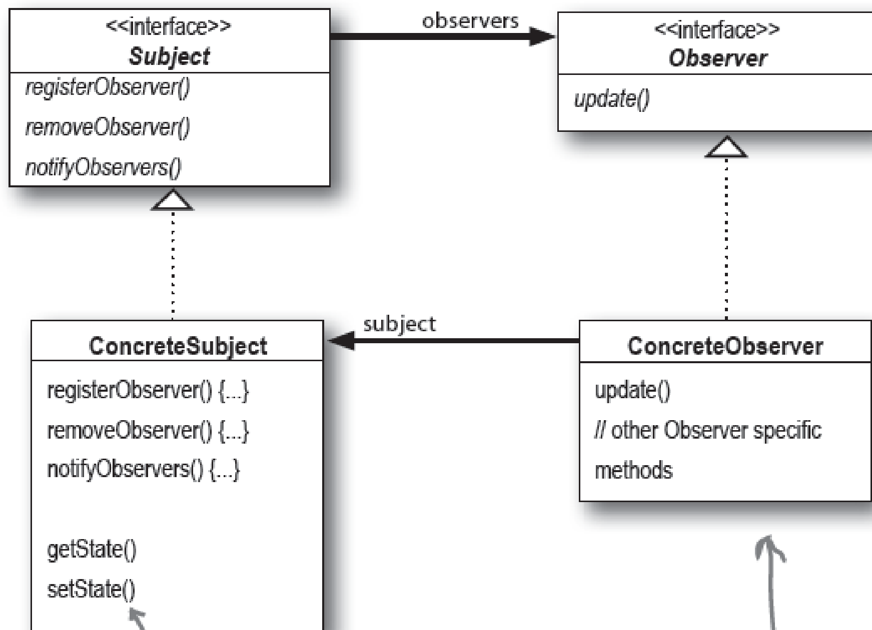


The Observer Pattern defined: the class diagram

Here's the Subject interface. Objects use this interface to register as observers and also to remove themselves from being observers.

Each subject can have many observers.

All potential observers need to implement the Observer interface. This interface just has one method, update(), that gets called when the Subject's state changes.



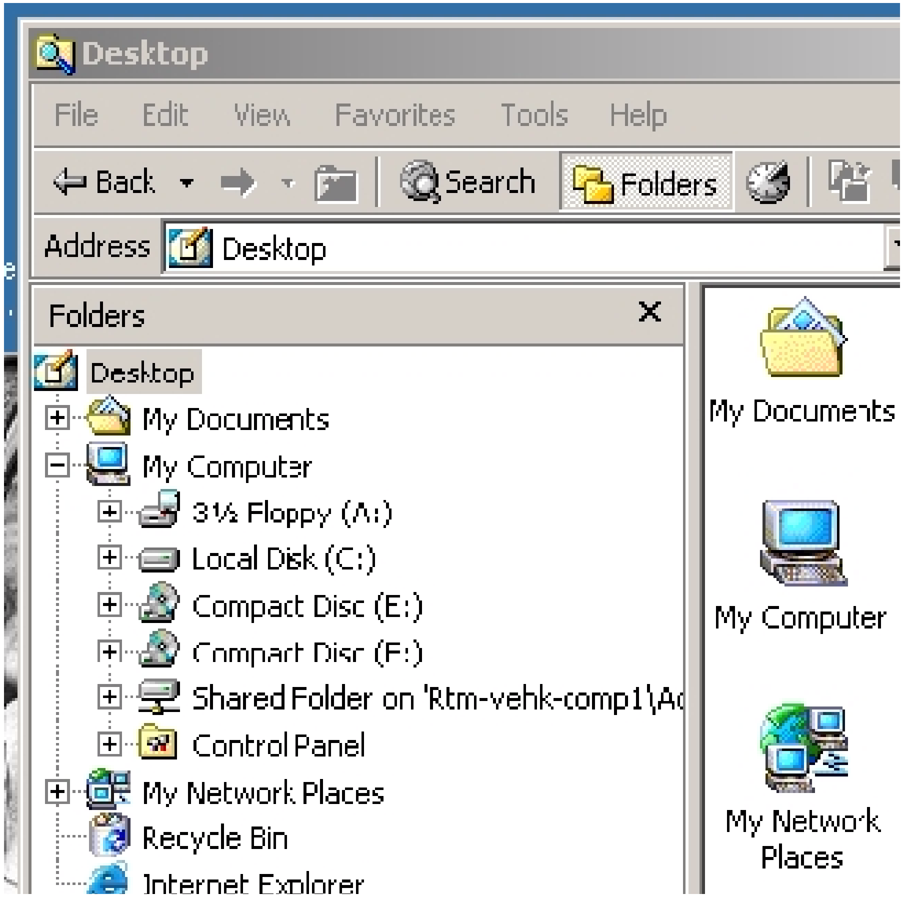
A concrete subject always implements the Subject interface. In addition to the register and remove methods, the concrete subject implements a notifyObservers() method that is used to update all the current observers whenever state changes.

The concrete subject may also have methods for setting and getting its state (more about this later).

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

JTrees

- Used to display a hierarchical structure
  - File structure, browsing history, etc…



Editing

- To edit the tree, you must go through the model:

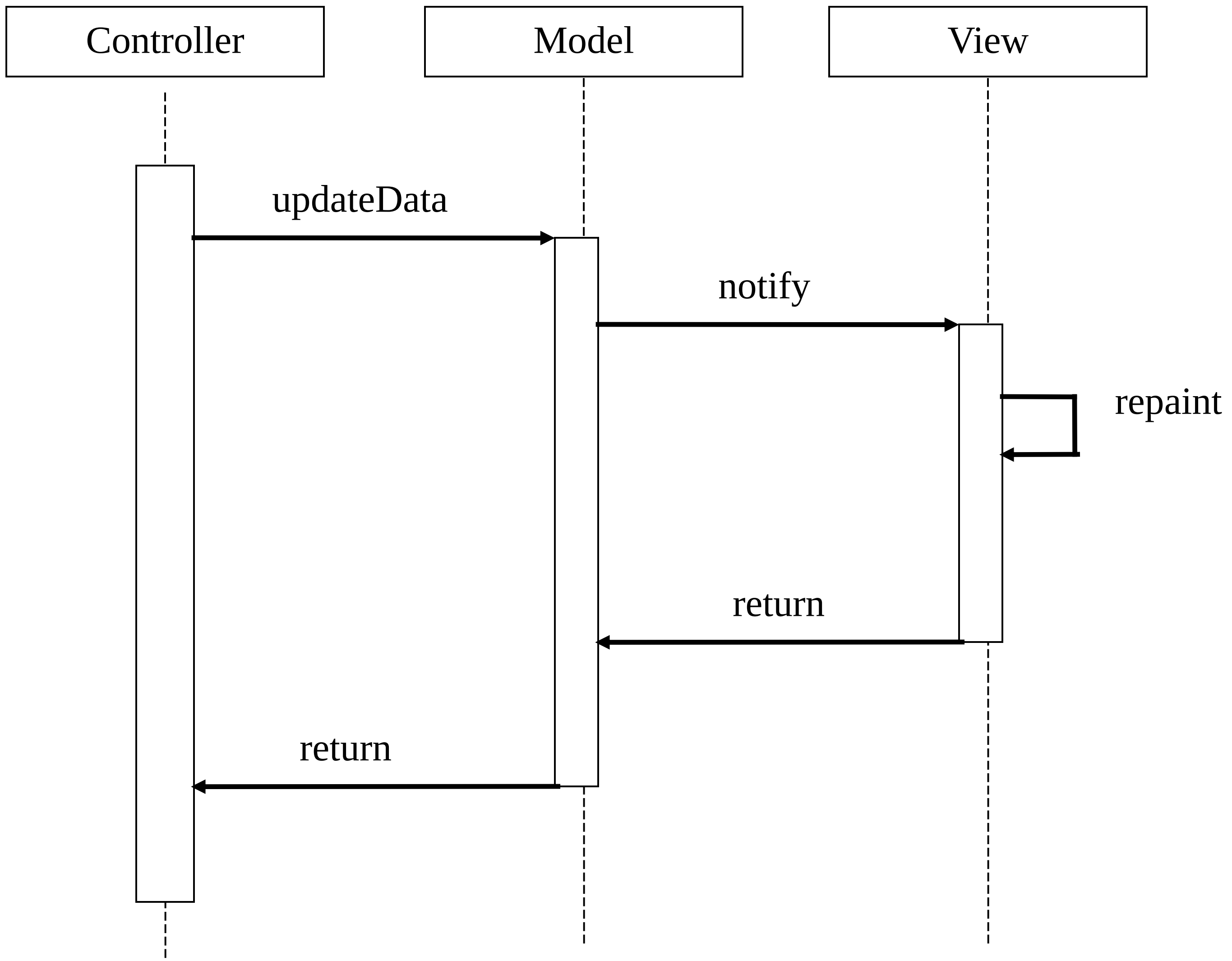
```
JTree tree = new JTree(…);
TreeModel model = tree.getModel();

// Insert Node
model.insertNodeInto(…

// Remove Node
model.removeNodeFromParent(…

// Change Node
model.nodeChanged(…

// UPDATING THE MODEL WILL NOW AUTOMATICALLY UPDATE
// THE VIEW (JTree) THANKS TO MVC!
```

Complex Controls have their own States

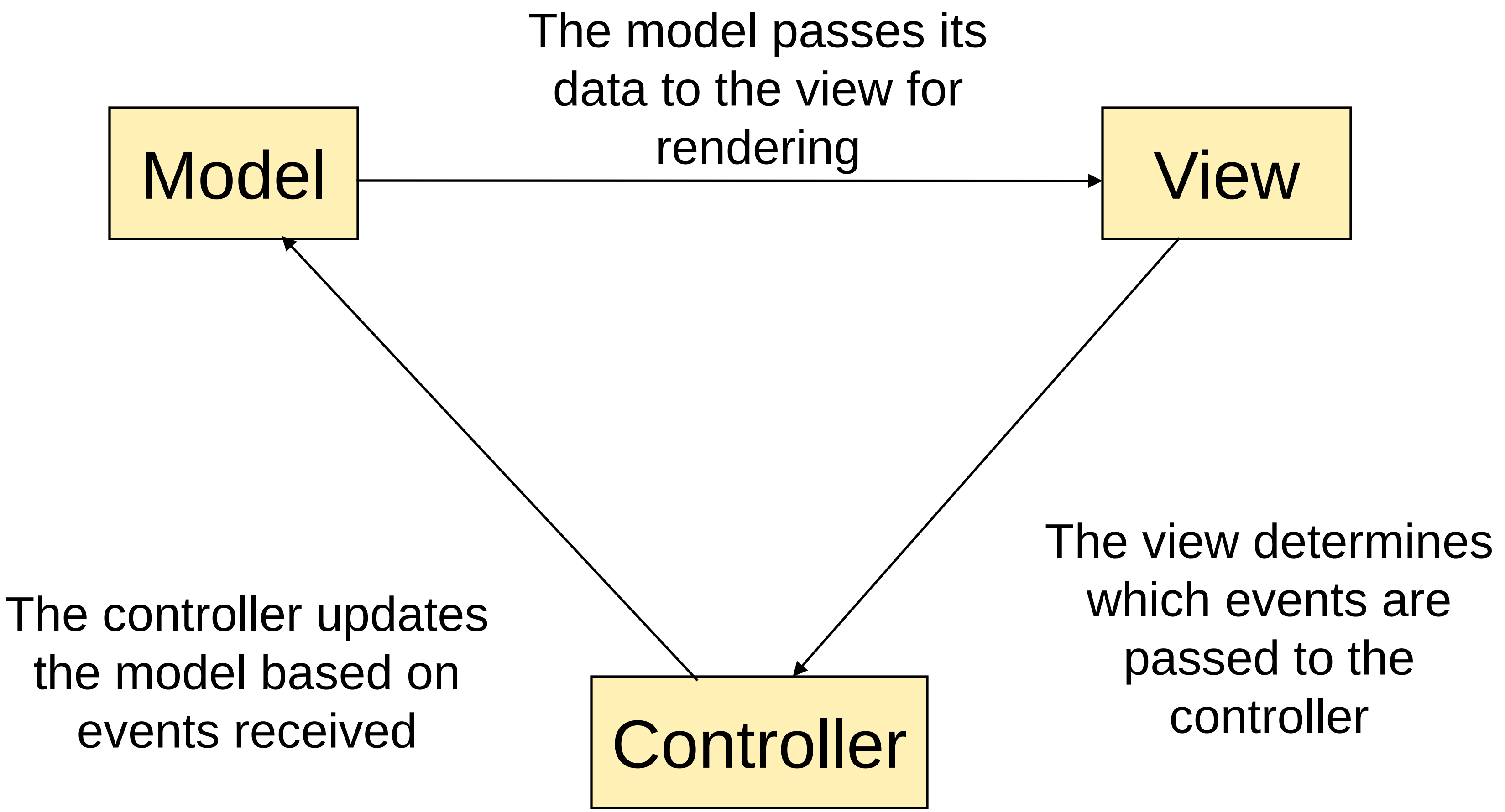
- Tables, trees, lists, combo boxes, etc.
  - data is managed separately from the view
  - when the state changes, the view is updated
- This is called MVC
  - Model
  - View
  - Controller
- MVC employs the Observer Pattern

MVC employs the Observer Pattern

- Model
  - data structure, no visual representation
  - notifies views when something interesting happens
- View / Observer
  - visual representation
  - views attach themselves to model in order to be notified
- Controller
  - event handler
  - listeners that are attached to view in order to be notified of user interaction (or otherwise)
- MVC Interaction
  - controller updates model
  - model tells view that data has changed
  - view redrawn


Controller
Model
View
updateData
notify
repaint
return
return

MVC Architecture



Swing loves MVC

```
   Model            View          Controller
• ComboBoxModel  - JComboBox - ItemListener
• ListModel      - JList     - ListSelectionListener
• TableModel     - JTable    - CellEditorListener
• TreeModel      - JTree     - TreeSelectionListener
…
```

- **Note:**
  - **all Swing components have loads of different listeners**

Common Design Patterns

Creational	Structural	Behavioral
• Factory	• Decorator	• Strategy
• Singleton	• Adapter	• Template
• Builder	• Facade	• Observer
• Prototype	• Flyweight	• Command
	• Bridge	• Iterator
		• State

Textbook: Head First Design Patterns

Command Abstraction

- For many GUIs, a single function may be triggered by many means (e.g., keystroke, menu, button, etc…)
  - we want to link all similar events to the same listener
- The information concerning the command can be abstracted to a separate command object
- Common Approach:
  - specify a String for each command
    - have listener respond to each command differently
  - ensure commands are handled in a uniform way
  - commands can be specified inside a text file
  - **The Command Pattern**

```java
interface Command {
        void execute();
}
class GarageDoor {
        private boolean open = false;
        public void open() {
                this.open = true;
        }
        public void close() {
                this.open = false;
        }
        public void showStatus() {
                System.out.println("The door is:"+this.open);
        }
}
class GarageDoorOpenCommand implements Command {
        private GarageDoor garageDoor;
        @Override
        public void execute() {
                garageDoor.open();
                garageDoor.showStatus();
        }
        public GarageDoorOpenCommand(GarageDoor door) {
                this.garageDoor = door;
        }
}
```

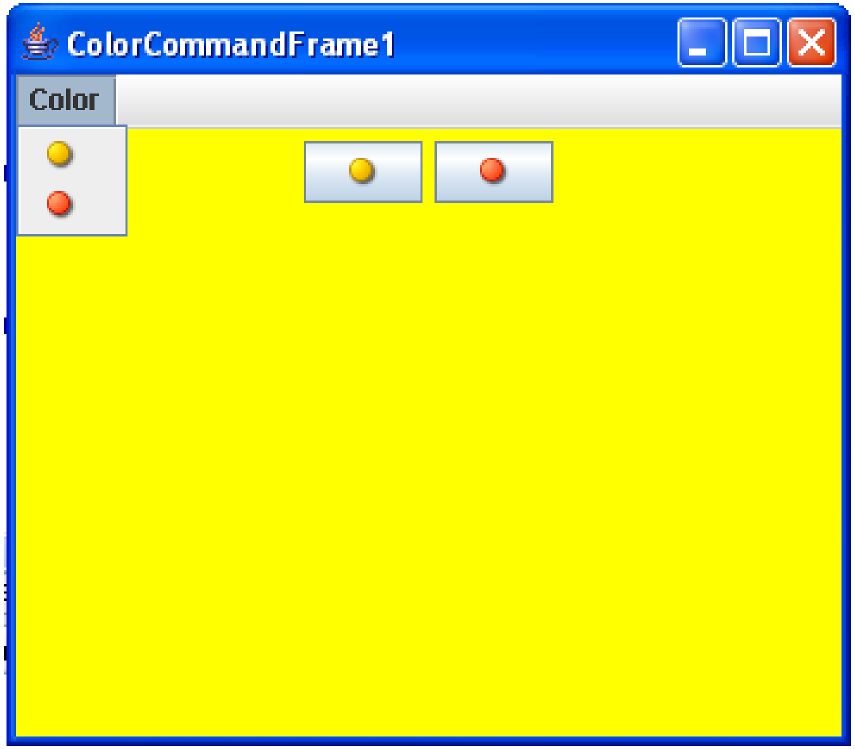
```java
class Light {
        private boolean on = false;
        public void on() {
                this.on = true;
        }
        public void off() {
                this.on = false;
        }
        public void toggle() {
                this.on = !this.on;
        }
        public void showStatus() {
                System.out.println("The light is "+this.on);
        }
}
public class LightOnCommand implements Command {
        private Light light;
        @Override
        public void execute() {
                light.on();
                light.showStatus();
        }
        public LightOnCommand(Light light) {
                this.light = light;
        }
}
```

```
class SimpleRemoteControl {
      private Command command;
      public void setCommand(Command command) {
            this.command = command;
      }
      public void buttonPressed() {
            command.execute();
      }
}
public class Main {
      public static void main(String[] args) {
            SimpleRemoteControl control =
                  new SimpleRemoteControl();
            control.setCommand(new LightOnCommand(
                  new Light()));
            control.buttonPressed();
            control.setCommand(new GarageDoorOpenCommand(
                  new GarageDoor()));
            control.buttonPressed();
      }
}
```

Output:
The light is true
The door is:true

Example

- Suppose I wanted to create a simple GUI:
  - 1 colored panel
  - 2 buttons, yellow & red
  - 2 menu items, yellow & red
  - clicking on the buttons or menu items changes the color of the panel
- Since the buttons & the menu items both perform the same function, they should be tied to the same commands
  - I could even add popup menu items



Using Command Strings

```
public class ColorCommandFrame1 extends JFrame
                      implements ActionListener {
    private Toolkit tk = Toolkit.getDefaultToolkit();
    private ImageIcon yellowIcon
      = new ImageIcon(tk.getImage("yellow_bullet.bmp"));
    private ImageIcon redIcon
      = new ImageIcon(tk.getImage("red_bullet.bmp"));

    private JPanel coloredPanel = new JPanel();
    private JButton yellowButton = new JButton(yellowIcon);
    private JButton redButton = new JButton(redIcon);

    private JMenuBar menuBar = new JMenuBar();
    private JMenu colorMenu = new JMenu("Color");
    private JMenuItem yellowMenuItem = new JMenuItem(yellowIcon);
    private JMenuItem redMenuItem = new JMenuItem(redIcon);

    private JPopupMenu popupMenu = new JPopupMenu();
    private JMenuItem yellowPopupItem = new JMenuItem(yellowIcon);
    private JMenuItem redPopupItem = new JMenuItem(redIcon);

    private static final String YELLOW_COMMAND = "YELLOW_COMMAND";
    private static final String RED_COMMAND = "RED_COMMAND";
```

```java
public ColorCommandFrame1() {
    super("ColorCommandFrame1");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setExtendedState(JFrame.MAXIMIZED_BOTH);
    initButtons();
    initPopupMenu();
    initMenu();
}

public void initButtons() {
    yellowButton.setActionCommand(YELLOW_COMMAND);
    redButton.setActionCommand(RED_COMMAND);
    yellowButton.addActionListener(this);
    redButton.addActionListener(this);
    coloredPanel.add(yellowButton);
    coloredPanel.add(redButton);
    Container contentPane = getContentPane();
    contentPane.add(coloredPanel);
}
```

```java
public void initPopupMenu() {
   yellowPopupItem.setActionCommand(YELLOW_COMMAND);
   redPopupItem.setActionCommand(RED_COMMAND);
   yellowPopupItem.addActionListener(this);
   redPopupItem.addActionListener(this);
   popupMenu.add(yellowPopupItem);
   popupMenu.add(redPopupItem);
   coloredPanel.addMouseListener(new MouseAdapter() {
     public void mousePressed(MouseEvent e) {
       maybeShowPopup(e);
     }
     public void mouseReleased(MouseEvent e) {
       maybeShowPopup(e);
     }
     private void maybeShowPopup(MouseEvent e) {
       if (e.isPopupTrigger()) {
         popupMenu.show(e.getComponent(),e.getX(),e.getY());
       }
     }
   });
}
```

```java
  public void initMenu() {
     yellowMenuItem.setActionCommand(YELLOW_COMMAND);
     redMenuItem.setActionCommand(RED_COMMAND);
     yellowMenuItem.addActionListener(this);
     redMenuItem.addActionListener(this);
     colorMenu.add(yellowMenuItem);
     colorMenu.add(redMenuItem);
     menuBar.add(colorMenu);
     setJMenuBar(menuBar);
  }
  public void actionPerformed(ActionEvent ae) {
      String command = ae.getActionCommand();
      if (command.equals(YELLOW_COMMAND))
             coloredPanel.setBackground(Color.YELLOW);
      else if (command.equals(RED_COMMAND))
             coloredPanel.setBackground(Color.RED);
   }
}
```

Common Design Patterns

Creational	Structural	Behavioral
• Factory • Singleton • Builder • Prototype	• Decorator • Adapter • Facade • Flyweight • Bridge	• Strategy • Template • Observer • Command • Iterator • State

Textbook: Head First Design Patterns

The Iterator Pattern

- The iterator pattern is a design pattern in which an iterator is used to traverse a container and access the container's elements: "*you have to perform some operation on a sequence of elements in a given data structure*"
  - it decouples algorithms from containers,
  - the iterator object will maintain the state of the iteration, keeping track of the current item and having a way of identifying what elements are next to be iterated.

The Iterator Pattern

- An **`Iterator`** produces proper elements for processing
- Defining an Iterator may be complex
- Using an Iterator must be simple
  - they're all used in the same way
- E.g. **`update()`** all elements of **`List list`**:

```
Iterator it;
for (it=list.listIterator(); it.hasNext(); )
        it.next().update();
```

- Makes iteration through elements of a set "higher level"
- Separates the *production* of elements for iteration from the *operation* at each step in the iteration.

The Iterator Pattern

- Iterator is a design pattern that is encountered very often.
  - Problem: Mechanism to operate on every element of a set.
  - Context: The set is represented in some data structure (list, array, hashtable, etc.)
  - Solution: Provide a way to iterate through every element.
- Common Classes using Iterators in Java API
  - StringTokenizer
  - Vector, ArrayList, etc …
  - Even I/O streams work like Iterators

Iterator (in Java)

```
public interface Iterator {
  // Returns true if there are more
  // elements to iterate over; false
  // otherwise
  public boolean hasNext();

  // If there are more elements to
  // iterate over, returns the next one.
  // Modifies the state “this” to record
  // that it has returned the element.
  // If no elements remain, throw
  // NoSuchElementException.
  public Object next()
       throws NoSuchElementException;

  public void remove();
}
```

Iterator vs. **Enumeration**

- Java provides another interface **`Enumeration`** for iterating over a collection.
- **`Iterator`** is
  - newer (since JDK 1.2)
  - has shorter method names
  - has a **`remove`**() method to remove elements from a collection during iteration

Iterator	**Enumeration**
`hasNext()`	**`hasMoreElements()`**
`next()`	**`nextElement()`**
`remove()`	**-**

- **`Iterator`** is recommended for new implementations.

Example Loop controlled by `next()`

```
private Payroll payroll = new
  Payroll();
...
public void decreasePayroll() {
  Iterator it = payroll.getIterator();
   while (it.hasNext()) {
     Employee e = (Employee)it.next();
     double salary = e.getSalary();
     e.setSalary(salary*.9);
   }
}
```

Implementing an `Iterator`

```
public class Payroll {
  private Employee[] employees;
  private int num_employees;
  ...
  // An iterator to loop through all Employees
  public Iterator getIterator() {
     return new EmplGen();
  }
  ...
  private class EmplGen implements Iterator {
  // see next slide
  ...
  }
}
```

Implementing an Iterator

```
private class EmplGen implements Iterator {
   private int n = 0;

   public boolean hasNext() {
      return n < num_employees;
   }

   public Object next() throws NoSuchElementException {
      Object obj;
      if (n < num_employees) {
         obj = employees[n];
         n++;
         return obj;
      }
      else throw new NoSuchElementException
      ("No More Employees");
   }
```

state of iteration captured by index n

returns true if there is an element left to iterate over

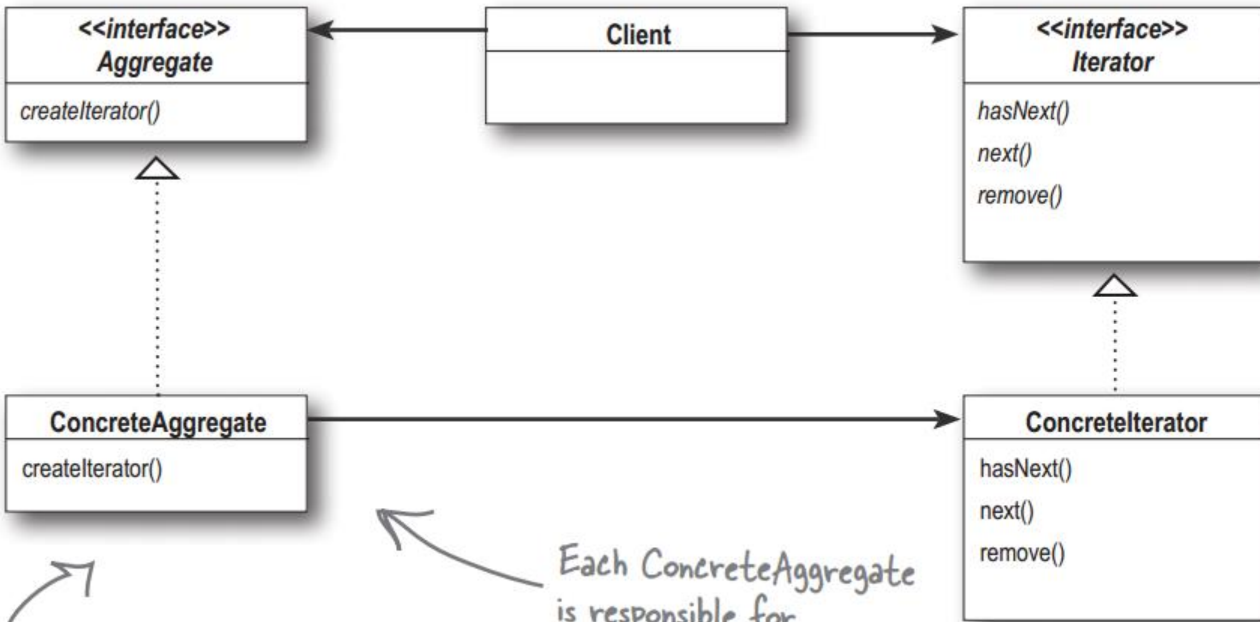
returns the next element in the iteration sequence

Implementing an Iterator

```
public Object remove(){
    Object obj;
    if (n < num_employees) {
        obj = employees[n];
        // shift left all the elements from n+1…num_employees
        for(int i=n+1; i<num_employees; i++)
            employees[i-1] = employees[i];
        num_employees--;
        return obj;
    }
    else throw new NoSuchElementException
    ("No More Employees");
  }
}
```

Having a common interface for your aggregates is handy for your client; it decouples your client from the implementation of your collection of objects.

The Iterator interface provides the interface that all iterators must implement, and a set of methods for traversing over elements of a collection. Here we're using the java.util.Iterator. If you don't want to use Java's Iterator interface, you can always create your own.



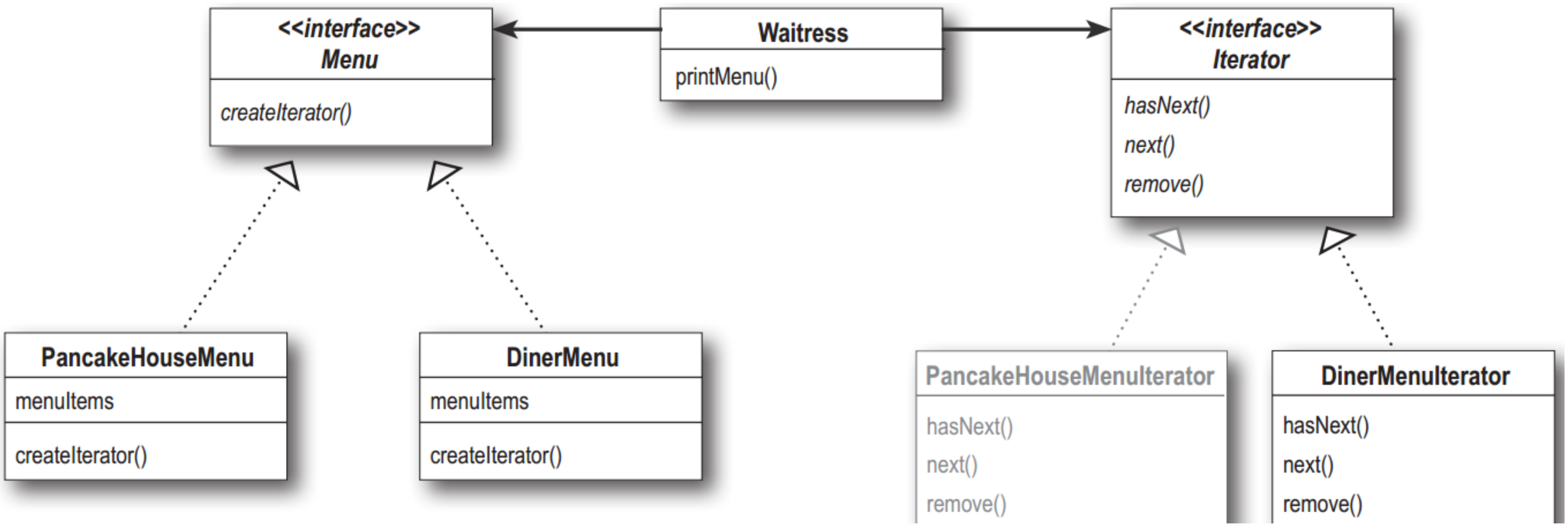
Each ConcreteAggregate is responsible for instantiating a ConcreteIterator that can iterate over its collection of objects.

The ConcreteAggregate has a collection of objects and implements the method that returns an Iterator for its collection.

The ConcreteIterator is responsible for managing the current position of the iteration.

The Iterator Pattern

Example: Pancake House and Diner Merger (they use different internal data structures: array vs. ArrayList



```java
import java.util.ArrayList;
import java.util.List;
import java.util.Iterator;
interface Menu<E> {
  public Iterator<E> getIterator();
}
class PancakeHouseMenu<E> implements Menu<E> {
  private List<E> food;
  public PancakeHouseMenu() {
     food = new ArrayList<E>();
  }
  public void addFood(E food) {
     this.food.add(food);
  }
  public PancakeHouseMenuIterator<E> getIterator(){
     return new PancakeHouseMenuIterator<E>(food);
  }
}
```

```java
class PancakeHouseMenuIterator<E> implements Iterator<E> {
   private List<E> food;
   private int position;
   public PancakeHouseMenuIterator(List<E> food) {
       this.food = food;
   }
   @Override
   public boolean hasNext() {
       if(position < food.size() && this.food.get(position)!=null){
           return true;
       } else {
           return false;
       }
   }
   @Override
   public E next() {
       return food.get(position++);
   }
   @Override
   public void remove() {
       this.food.remove(position);
   }
}
```

```
public class Waitress {
    Menu pancakeHouseMenu;
    Menu dinerMenu;
    public Waitress(Menu pancakeHouseMenu, Menu dinerMenu) {
        this.pancakeHouseMenu = pancakeHouseMenu;
        this.dinerMenu = dinerMenu;
    }
    public void printMenu() {
        Iterator pancakeIterator = pancakeHouseMenu.createIterator();
        Iterator dinerIterator = dinerMenu.createIterator();
        System.out.println("MENU\n----\nBREAKFAST");
        printMenu(pancakeIterator);
        System.out.println("\nLUNCH");
        printMenu(dinerIterator);
    }
    private void printMenu(Iterator iterator) {
        while (iterator.hasNext()) {
            MenuItem menuItem = (MenuItem)iterator.next();
            System.out.print(menuItem.getName() + ", ");
            System.out.print(menuItem.getPrice() + " -- ");
            System.out.println(menuItem.getDescription());
        }
}
```

```java
   public static void main(String args[]) {
      PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
      DinerMenu dinerMenu = new DinerMenu();
      Waitress waitress = new Waitress(pancakeHouseMenu, dinerMenu);
      waitress.printMenu();
   }
}
class MenuItem {
   String name;
   String description;
   boolean vegetarian;
   double price;
   public MenuItem(String name, String description,
          boolean vegetarian, double price) {
      this.name = name;
      this.description = description;
      this.vegetarian = vegetarian;
      this.price = price;
   }
   public String getName() {
      return name;
   }
...
```

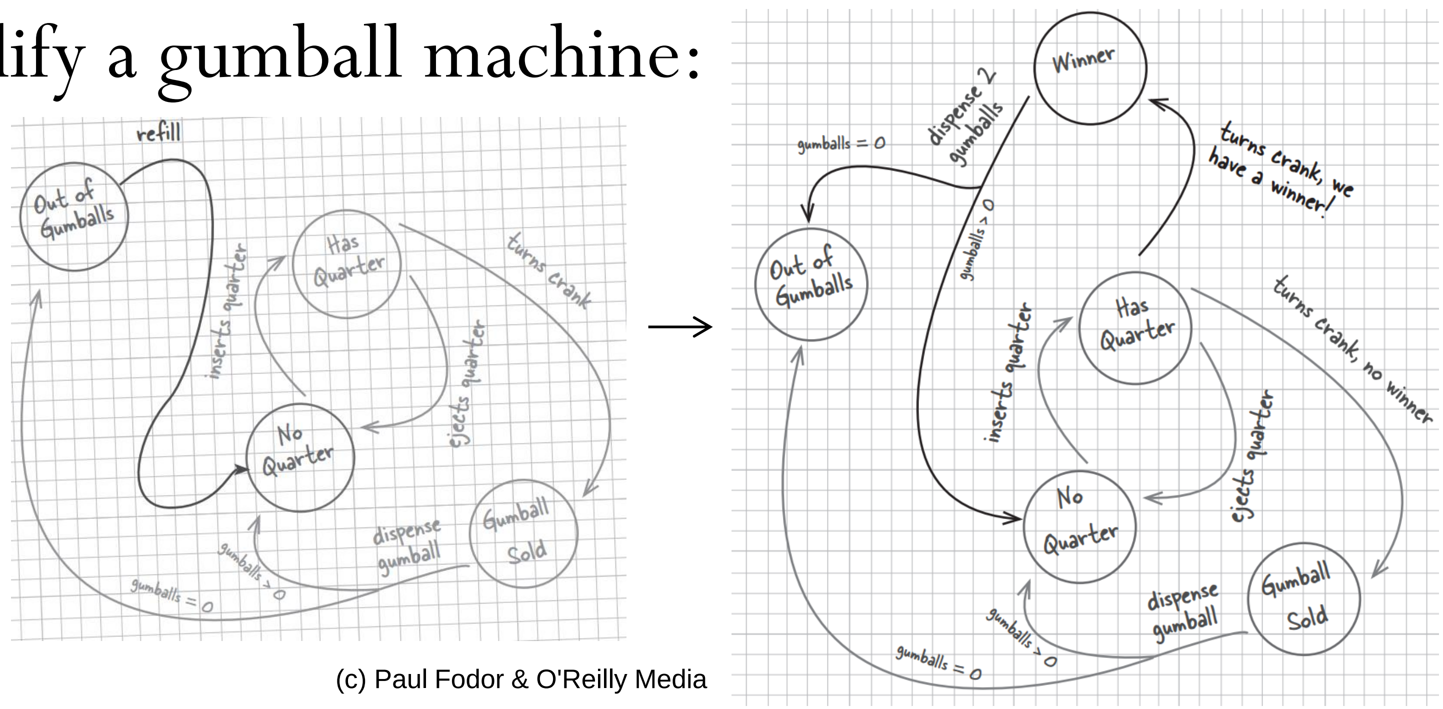
Common Design Patterns

Creational	Structural	Behavioral
• Factory	• Decorator	• Strategy
• Singleton	• Adapter	• Template
• Builder	• Facade	• Observer
• Prototype	• Flyweight	• Command
	• Bridge	• Iterator
		• State

Textbook: Head First Design Patterns

The State Pattern

- It is used to encapsulate varying behavior for the same routine based on an object's state object.
- If we model the state machine with constants, it is difficult to change it later.
- E.g., modify a gumball machine:



```java
interface State {
   void insertQuarter();
   void ejectQuarter();
   void turnCrank();
   void dispense();
   void refill(int balls);
}
class HasQuarterState implements State {
   private GumballMachine machine;
   public HasQuarterState(GumballMachine machine) {
       this.machine = machine;
   }
   @Override
   public void insertQuarter() {
       System.out.println("You already inserted a quarter");
   }
   @Override
   public void ejectQuarter() {
       System.out.println("Quarter ejected");
       this.machine.setState(machine.getNoQuarterState());
   }
   @Override
   public void turnCrank() {
       System.out.println("You turned crank!");
       machine.setState(machine.getSoldState());
       machine.dispense();
   }
```

```
    @Override
    public void dispense() {
        System.out.println("You have to turn the crank first!");
    }
    @Override
    public void refill(int balls) {
        System.out.println("Can only refill if sold out.");
    }
}
class SoldState implements State {
    private GumballMachine machine;
    public SoldState(GumballMachine machine) {
        this.machine = machine;
    }
    @Override
    public void insertQuarter() {
        System.out.println("Wait a second ... you are getting a gumball.");
    }
    @Override
    public void ejectQuarter() {
        System.out.println("Sorry, you already turned the crank.");
    }
    @Override
    public void turnCrank() {
        System.out.println("You alread turned the crank. ");
    }
```

```java
    @Override
    public void dispense() {
        this.machine.releaseBall();
        if(machine.getCount() > 0) {
            this.machine.setState(machine.getNoQuarterState());
        } else {
            System.out.println("Oops ... out of gumballs");
            machine.setState(machine.getSoldOutState());
        }
    }
    @Override
    public void refill(int balls) {
        System.out.println("Can only refill if sold out.");
    }
}
class SoldOutState implements State {
    private GumballMachine machine;
    public SoldOutState(GumballMachine machine) {
        this.machine = machine;
    }
    @Override
    public void insertQuarter() {
        System.out.println("Machine sold out. You cannot insert a quarter");
    }
    @Override
    public void ejectQuarter() {
    }
```

```java
    @Override
    public void turnCrank() {
    }
    @Override
    public void dispense() {
    }
    @Override
    public void refill(int balls) {
        this.machine.setCount(balls);
        this.machine.setState(this.machine.getNoQuarterState());
    }
}
class NoQuarterState implements State {
    private GumballMachine machine;
    public NoQuarterState(GumballMachine machine) {
        this.machine = machine;
    }
    @Override
    public void insertQuarter() {
        System.out.println("You inserted a quarter");
        this.machine.setState(machine.getHasQuarterState());
    }
    @Override
    public void ejectQuarter() {
        System.out.println("You haven't inserted a quarter!");
    }
```

```java
    @Override
    public void turnCrank() {
        System.out.println("Please insert quarter first!");
    }
    @Override
    public void dispense() {
    }
    @Override
    public void refill(int balls) {
        System.out.println("Can only refill if sold out.");
    }
}
public class GumballMachine {
    private State hasQuarterState;
    private State noQuarterState;
    private State soldOutState;
    private State soldState;
    private State state;
    private int count = 0;
    public GumballMachine(int initialGumballs) {
        hasQuarterState = new HasQuarterState(this);
        noQuarterState = new NoQuarterState(this);
        soldOutState = new SoldOutState(this);
        soldState = new SoldState(this);
        if(initialGumballs > 0) {
            this.count = initialGumballs;
```

```
            this.state = noQuarterState;
        }
    }
    public void setState(State state) {
        this.state = state;
    }
    public State getHasQuarterState() {
        return hasQuarterState;
    }
    public State getNoQuarterState() {
        return noQuarterState;
    }
    public State getSoldOutState() {
        return soldOutState;
    }
    public State getSoldState() {
        return soldState;
    }
    public State getState() {
        return state;
    }
    public int getCount() {
        return count;
    }
    public void insertQuarter() {
        state.insertQuarter();
    }
```

```java
public void setCount(int count) {
    if(count >= 0) {
        this.count = count;
    }
}
public void ejectQuarter() {
    state.ejectQuarter();
}
public void turnCrank() {
    state.turnCrank();
    state.dispense();
}
public void dispense() {
    state.dispense();
}
public void releaseBall() {
    if(this.count > 0) {
        System.out.println("A gumball comes rolling down the slot!");
        this.count --;
    }
}
public void refill(int balls) {
    this.state.refill(balls);
}
```

```java
public static void main(String[] args) {
    GumballMachine machine = new GumballMachine(3);
    machine.insertQuarter();
    machine.turnCrank();

    machine.turnCrank();
    machine.insertQuarter();
    machine.dispense();
    machine.turnCrank();

    machine.insertQuarter();
    machine.turnCrank();

    machine.insertQuarter();
    machine.refill(2);
    machine.insertQuarter();
    System.out.println(machine.getCount());
    machine.turnCrank();
    System.out.println(machine.getCount());
}
```

```java
interface State {                              // Traffic light
                                               // Example
    void switchGreen();
    void switchOrange();
    void switchRed();
}
class GreenState implements State {
    private TrafficLight light;
    public GreenState(TrafficLight light) {
        this.light = light;
    }
    @Override
    public void switchGreen() {
        System.out.println("Light is already green!");
    }
    @Override
    public void switchOrange() {
        light.setState(this.light.getOrangeState());
        System.out.println("Light switched to orange!");
    }
    @Override
    public void switchRed() {
        System.out.println("First switch the light to orange");

    }
}
```

```
class OrangeState implements State {
  private TrafficLight light;
  public OrangeState(TrafficLight light) {
     this.light = light;
  }
  @Override
  public void switchGreen() {
     System.out.println("The light has to become red first!")
  }
  @Override
  public void switchOrange() {
     System.out.println("Light is already orange!");
  }
  @Override
  public void switchRed() {
     System.out.println("Light switched to red.");
     this.light.setState(this.light.getRedState());

  }
}
```

```java
class RedState implements State {
  private TrafficLight light;
  public RedState(TrafficLight light) {
     this.light = light;
  }
  @Override
  public void switchGreen() {
     System.out.println("Light switched to green");
     this.light.setState(this.light.getGreenState());
  }
  @Override
  public void switchOrange() {
     System.out.println("First switch the light to green");
  }
  @Override
  public void switchRed() {
     System.out.println("Light is already red.");
  }
}
```

```java
public class TrafficLight  {
   private State redState;
   private State orangeState;
   private State greenState;
   private State state;
   public TrafficLight() {
       redState = new RedState(this);
       orangeState = new OrangeState(this);
       greenState = new GreenState(this);
       state = this.getRedState();
   }
   public void switchGreen() {
       state.switchGreen();
   }
   public void switchOrange() {
       state.switchOrange();
   }
   public void switchRed() {
       state.switchRed();
   }
   public State getRedState() {
       return redState;
   }
   public State getOrangeState() {
       return orangeState;
   }
```

```java
    public State getGreenState() {
        return greenState;
    }

    public void setState(State state) {
        this.state = state;
    }

    public static void main(String[] args) {
        TrafficLight light = new TrafficLight();

        light.switchOrange();
        light.switchGreen();
        light.switchRed();
        light.switchOrange();
        light.switchOrange();
        light.switchRed();
    }

}
```

Design Patterns

- Many other design patterns:
  - Concurrency patterns:
    - Monitor object: An object whose methods are subject to mutual exclusion, thus preventing multiple objects from erroneously trying to use it at the same time.
    - Reactor: A reactor object provides an asynchronous interface to resources that must be handled synchronously.
    - Read-write lock: Allows concurrent read access to an object, but requires exclusive access for write operations.
    - Scheduler: Explicitly control when threads may execute single-threaded code.

    Active object, Balking, Event-based asynchronous, Guarded suspension, Join, Lock, Monitor, Proactor, Read write lock, Thread Pool, Thread-local storage
  - Architectural patterns: n-tier, Specification, Publish-subscribe, Service, Locator…
- Use the design patterns, BUT inappropriate use of patterns may unnecessarily increased complexity.
- Other languages have other patterns, e.g.: many patterns are simplified or eliminated in Lisp.