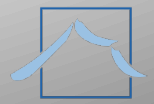
Informatics II for Engineering Sciences (MSE)

Chapter III – Design Patterns

Outline

1. **Introduction to Design Patterns**
2. **Adapter pattern:** Interfaces to existing systems (legacy systems)
3. **Composite Pattern:** Models dynamic aggregates
4. **Observer Pattern:** Provides consistency among objects

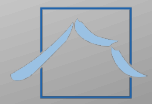


Introduction to Design Patterns

Definition:

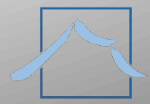
A pattern is a three-part rule, which expresses a relation between a certain *context*, a *problem*, and a *solution*

(Christopher Alexander)



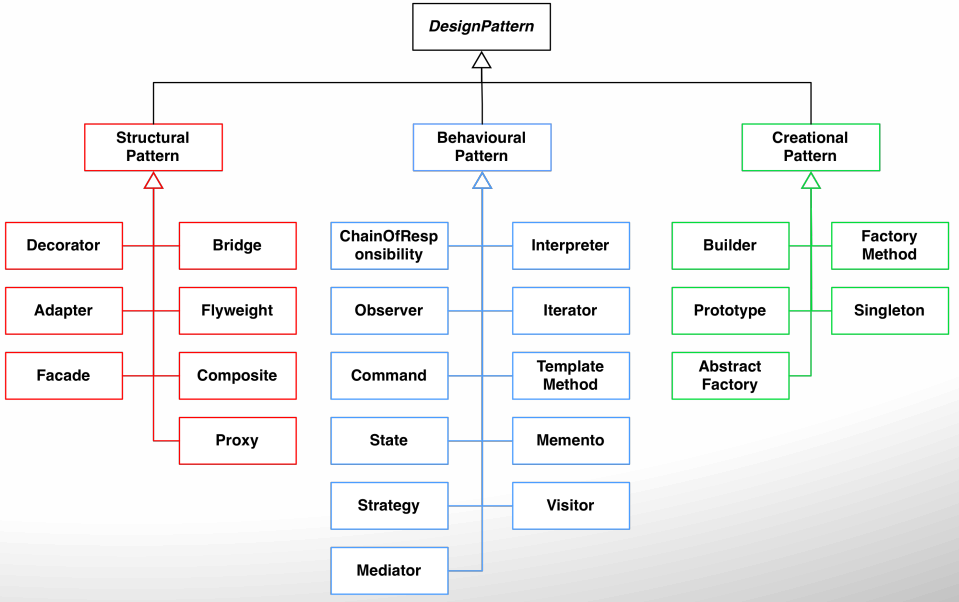
Why are Design Patterns Good?

- They are generalizations of detailed design knowledge from existing systems
- They provide a shared vocabulary to designers
- They provide examples of reusable designs
  - Polymorphism (Inheritance, sub-classing)
  - Delegation (or aggregation).



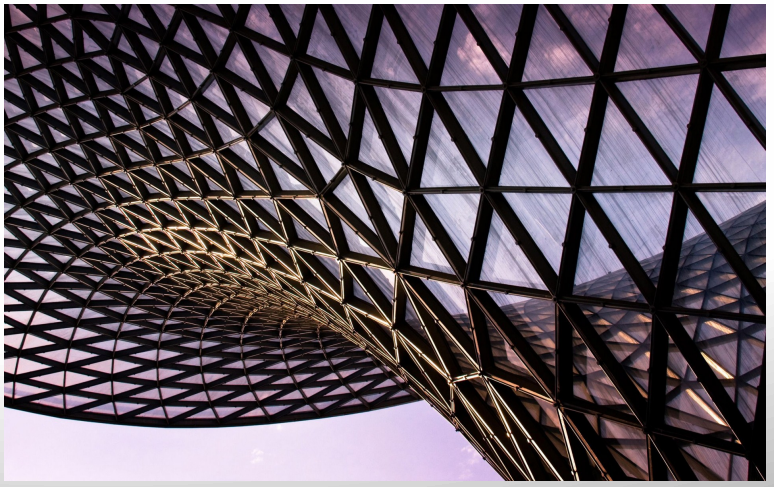
3 Types of Design Patterns

- ***Structural Patterns***
- ***Behavioral Patterns***
- ***Creational Patterns***



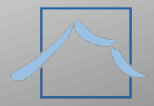
Structural Patterns

- **Structural Patterns**
  - Introduce an abstract class to enable future extensions
  - Reduce coupling between two or more classes
  - Encapsulate complex structures



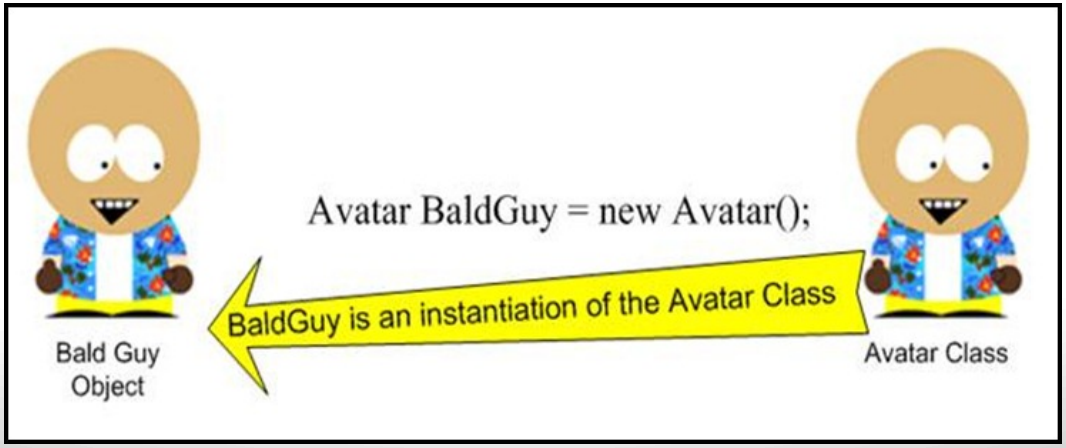
Behavioral Patterns

- **Behavioral Patterns**
  - Simplify complex control flows that are difficult to follow at runtime
  - Allow a choice between algorithms and the assignment of responsibilities to objects (“Who does what?”)

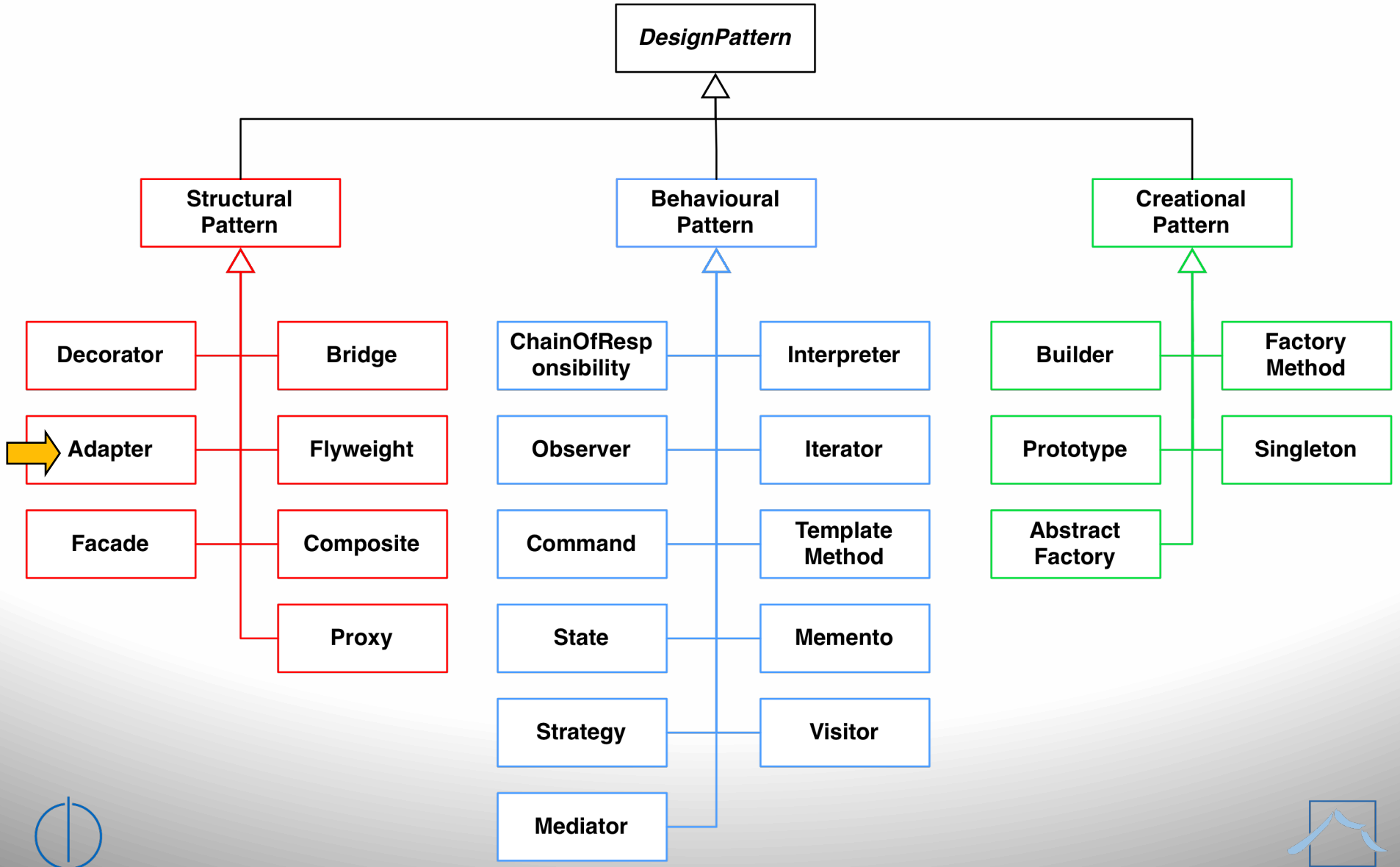


Creational Patterns

- **Creational Patterns**
  - Allow a simplified view from complex instantiation processes
  - Make the system independent from the way its objects are created, composed and represented.

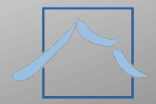


Design Patterns Taxonomy



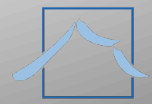
Adapter Pattern

- Connects incompatible components
  - Allows the simplified reuse of existing components
  - Useful in reengineering projects
    - Used to provide a new interface for (the ugly interface of) legacy systems
- Also known as a wrapper.

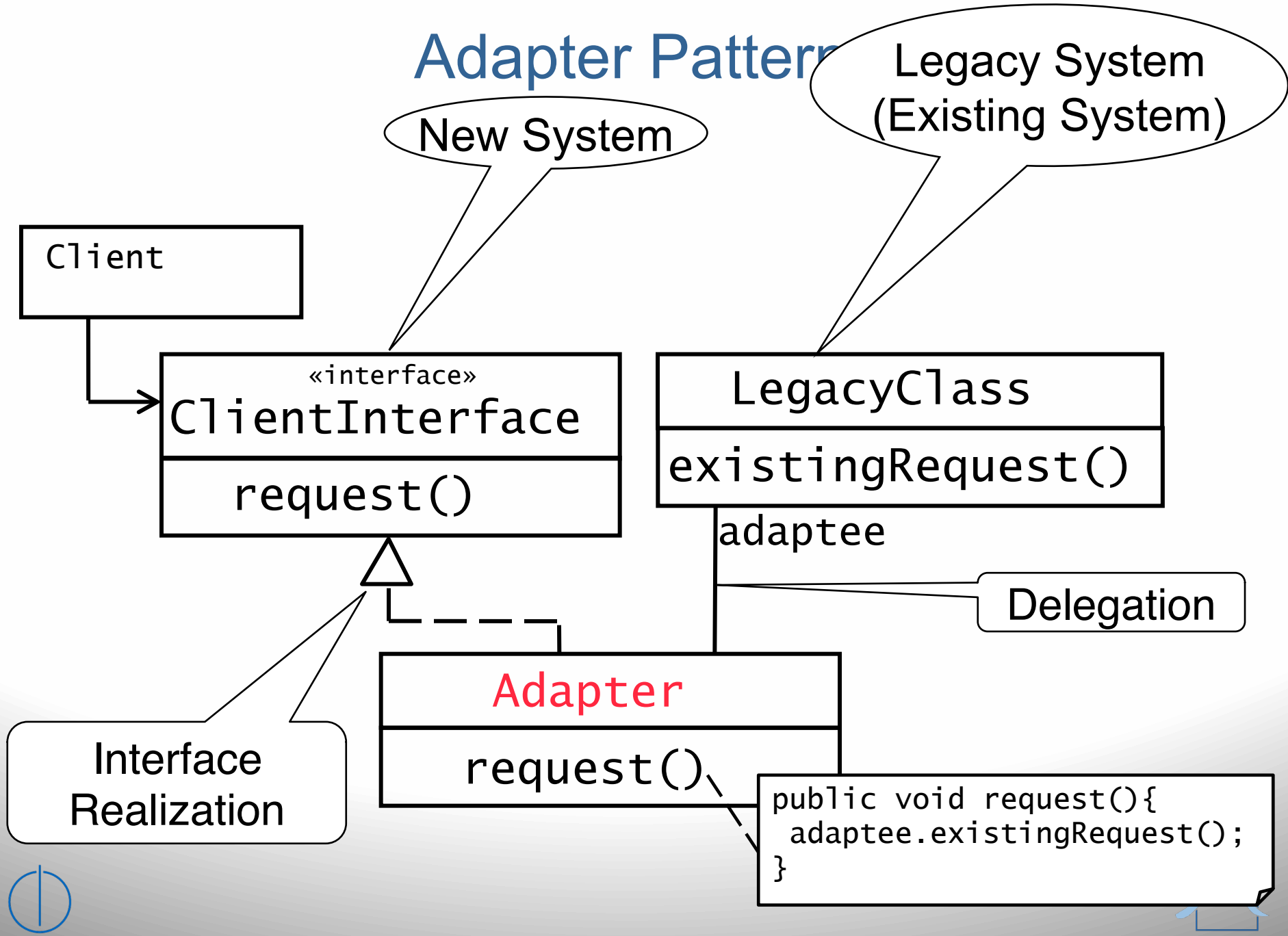


Using the Adapter Pattern

- **Main use:** Provide access to legacy systems
- **Legacy System:**
  - A legacy system is an old system that continues to be used, even though newer technology or more efficient methods are now available
- **Reasons for the continued use of a legacy system:**
  - System Cost: The system still makes money, the cost of designing a new system with the same functionality is too high
  - Poor Engineering (or Poor Management?): The system cannot not be changed because the compiler is no longer available or source code has been lost
  - Availability: The system requires 100% availability, it cannot simply be taken out of service and replaced with a new system
  - Pragmatism: The system is installed and working.



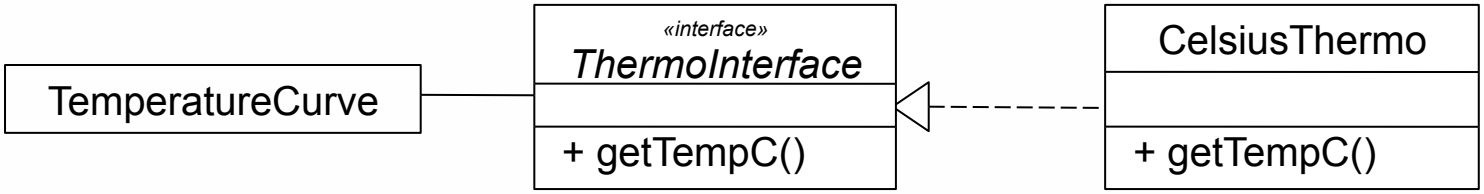
Adapter Pattern



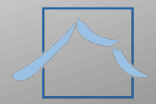
Exercise

Exercise #1: Adapter Pattern
Replacing a Broken Thermometer

- **Problem Statement**
  - You are on an expedition climbing Denali (6.193 m), one of the coldest mountains on earth. You need to reliably read the outside temperature for the last n hours (temperature curve) in Celsius
  - Inside the tent you are using a fancy digital thermometer with software implemented in Java. The program uses a *`ThermoInterface`* which provides the temperature in Celsius. It connects to the outside thermometer which runs software containing a class called `CelsiusThermo`



  - Somebody stepped on your outside Celsius thermometer (`CelsiusThermo`) and broke it
  - There is one more thermometer on the expedition, but this measures the temperature in Fahrenheit



Exercise #1 Adapter Pattern

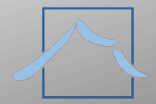
Your task: Write an adapter that solves the following problem

- Reuse the code from the Fahrenheit thermometer (`FahrenheitThermo`) while still providing temperatures in Celsius in `TemperatureCurve`

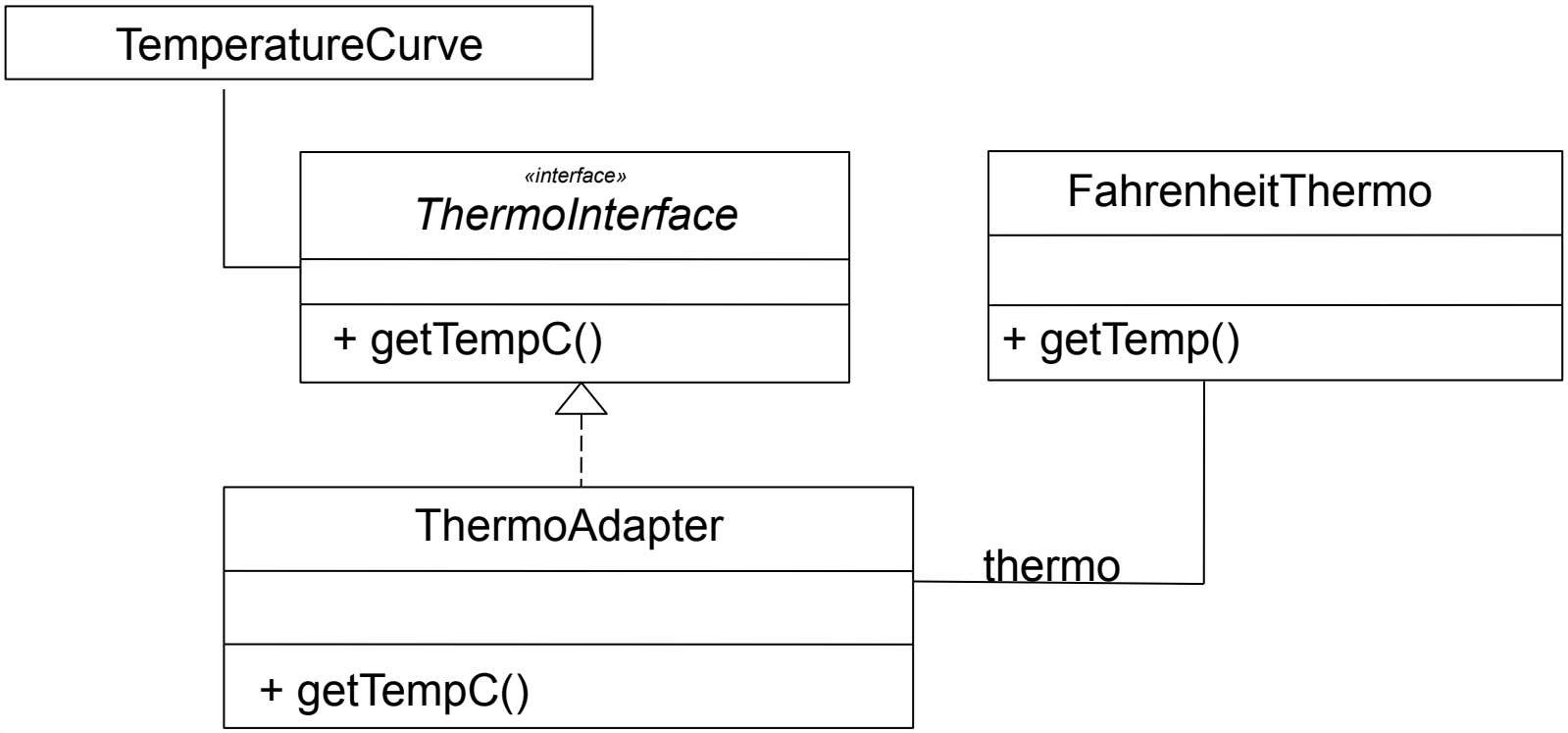
  tempCelsius = (tempFahrenheit -32.0) * (5.0/9.0)
- Constraint: The `TemperatureCurve` code should only be minimally changed
- Source code for the exercise is offered on Moodle

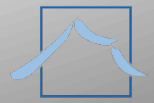
Upload your solution to Moodle: ➔ Exercise 1 – Student Solution Upload

You have 15 minutes!



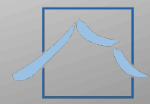
Exercise # 1 Adapter Pattern Solution



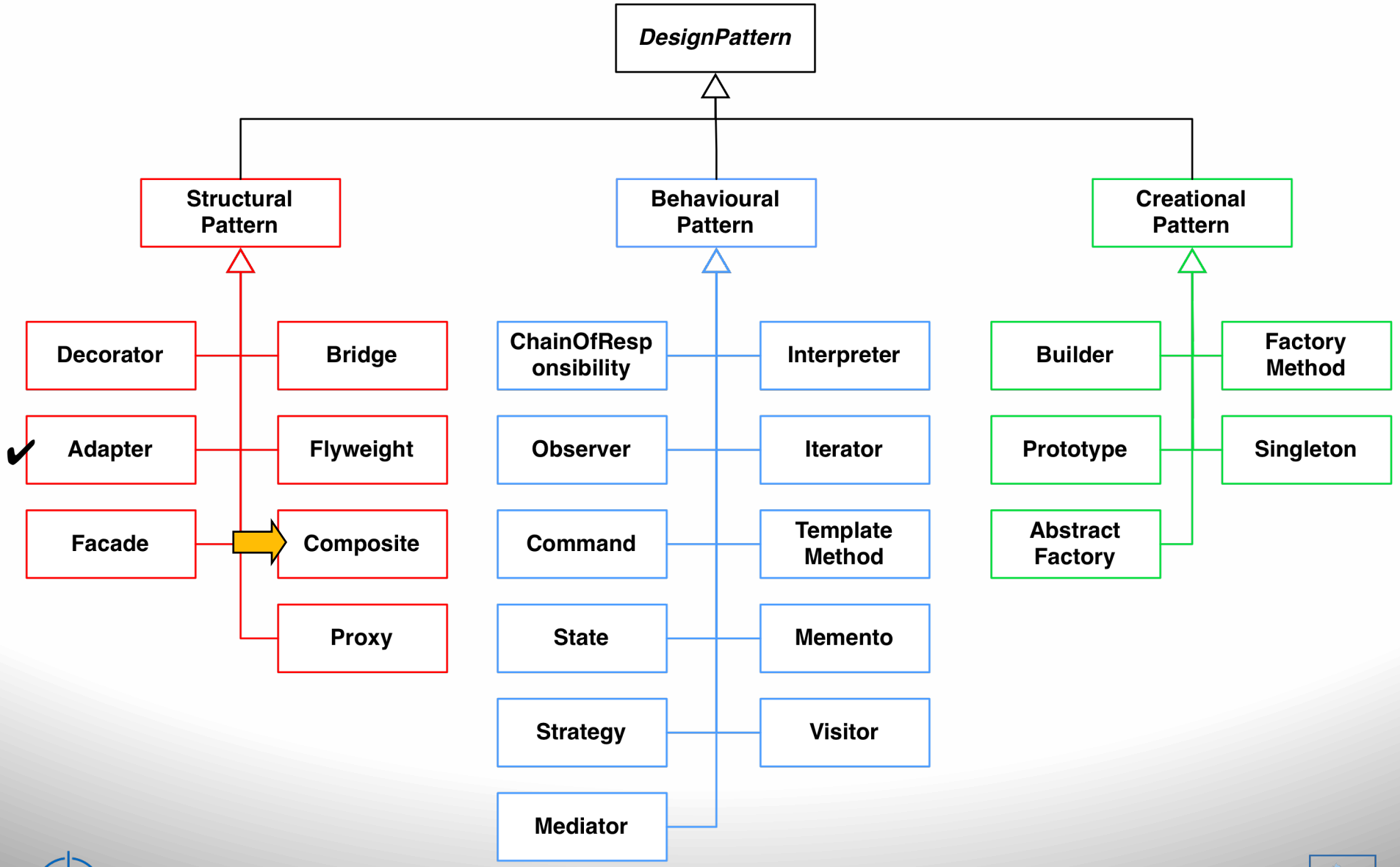


Thermometer Adapter – Solution (Ctd.)

```
public class ThermoAdapter implements ThermoInterface {

  private FahrenheitThermo thermo;

  public ThermoAdapter() {
    thermo = new FahrenheitThermo();
  }

  public double getTempC() {
    return (thermo.getTemp()-32.0) * (5.0/9.0);
  }
}
```

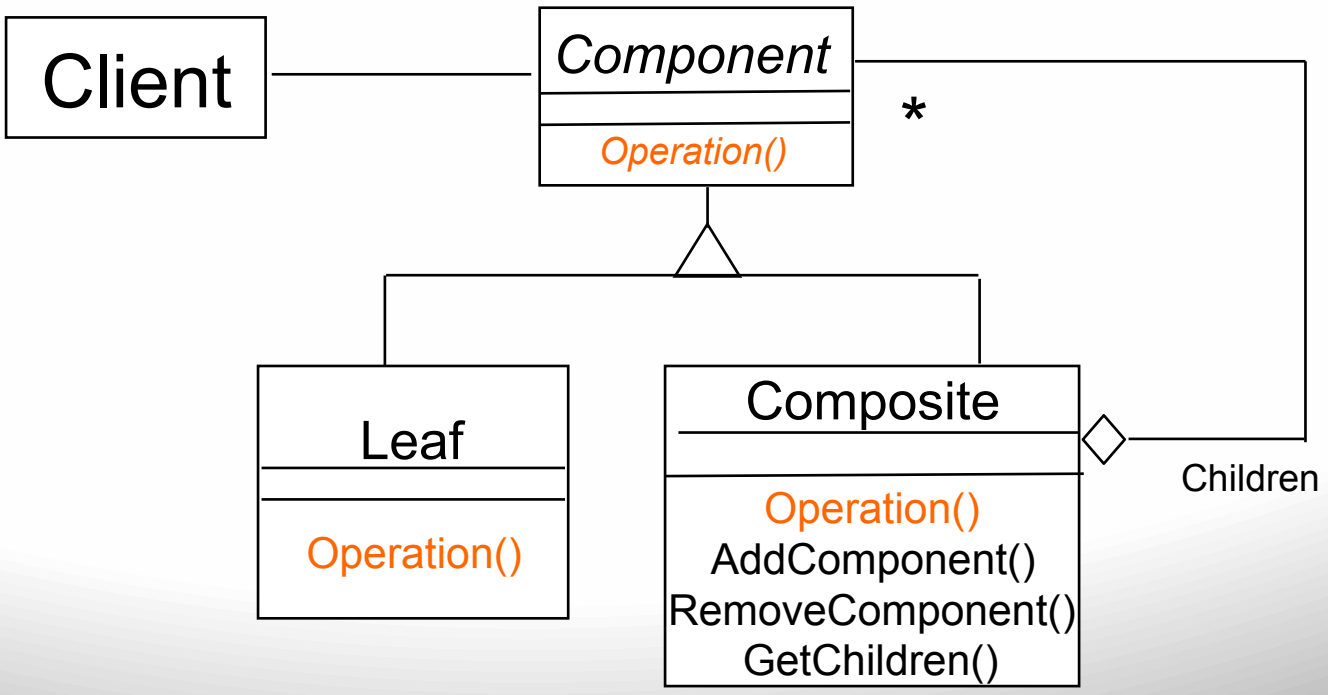


Design Patterns Taxonomy (23 Patterns)



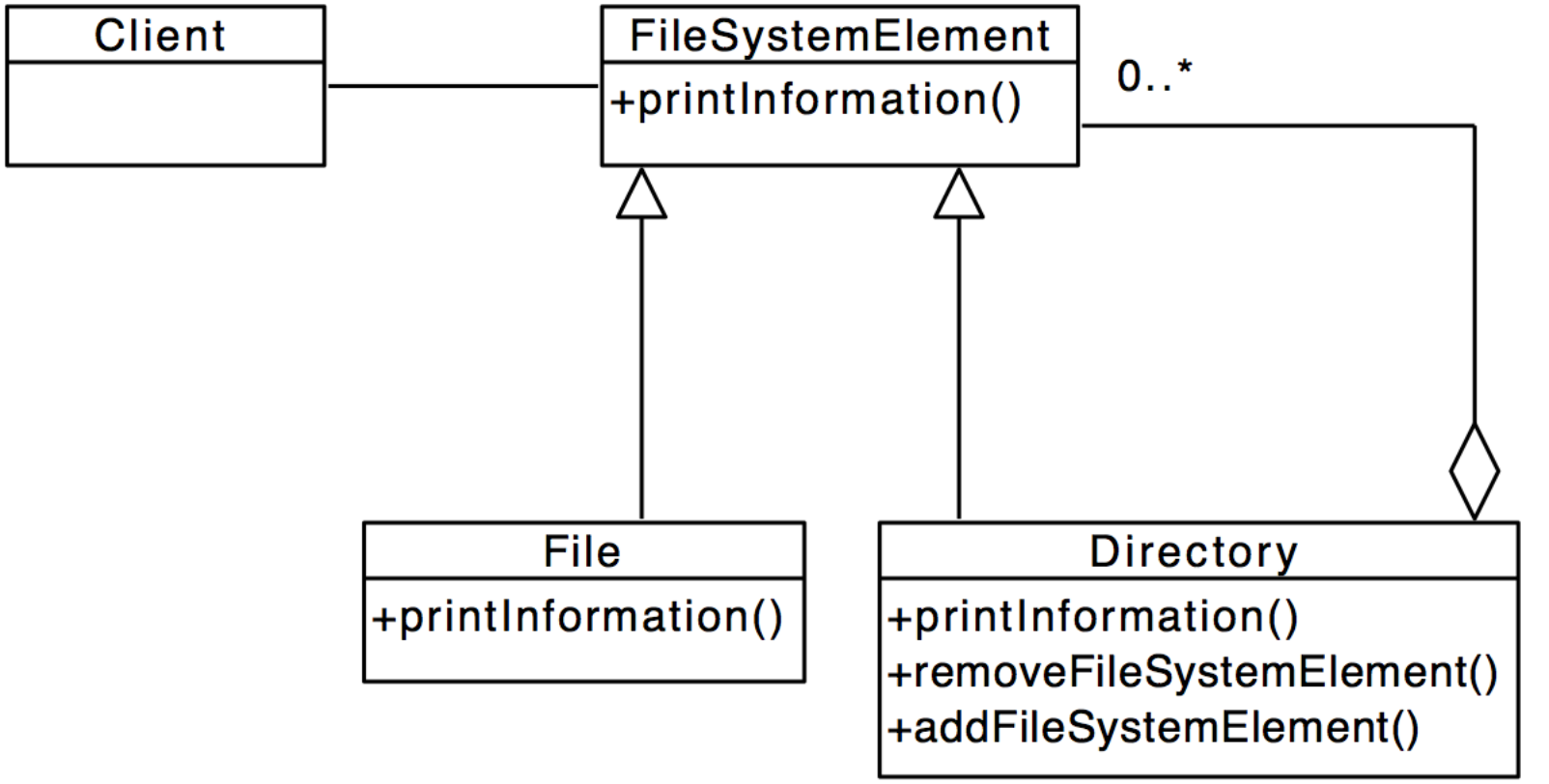
Introducing the Composite Pattern

- **Observation:**
  - Tree structures representing part-whole hierarchies with arbitrary depth and width can be used in the solution of many problems
- **Solution:**
  - The Composite Pattern lets a client treat individual objects and compositions of these objects **uniformly**



Composite Pattern Example: File System

- In a given file system a file system element can be either a file or a directory. A directory can have one or more file system elements.

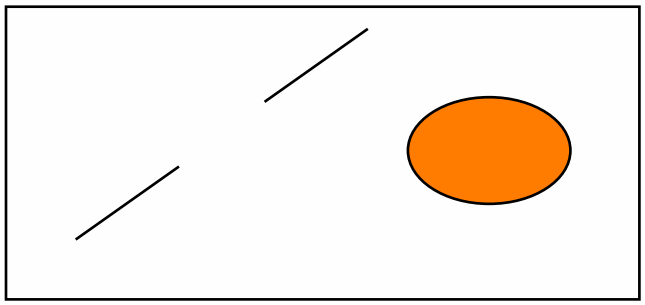


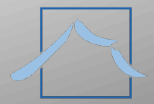
Exercise

Exercise #2 Composite Pattern

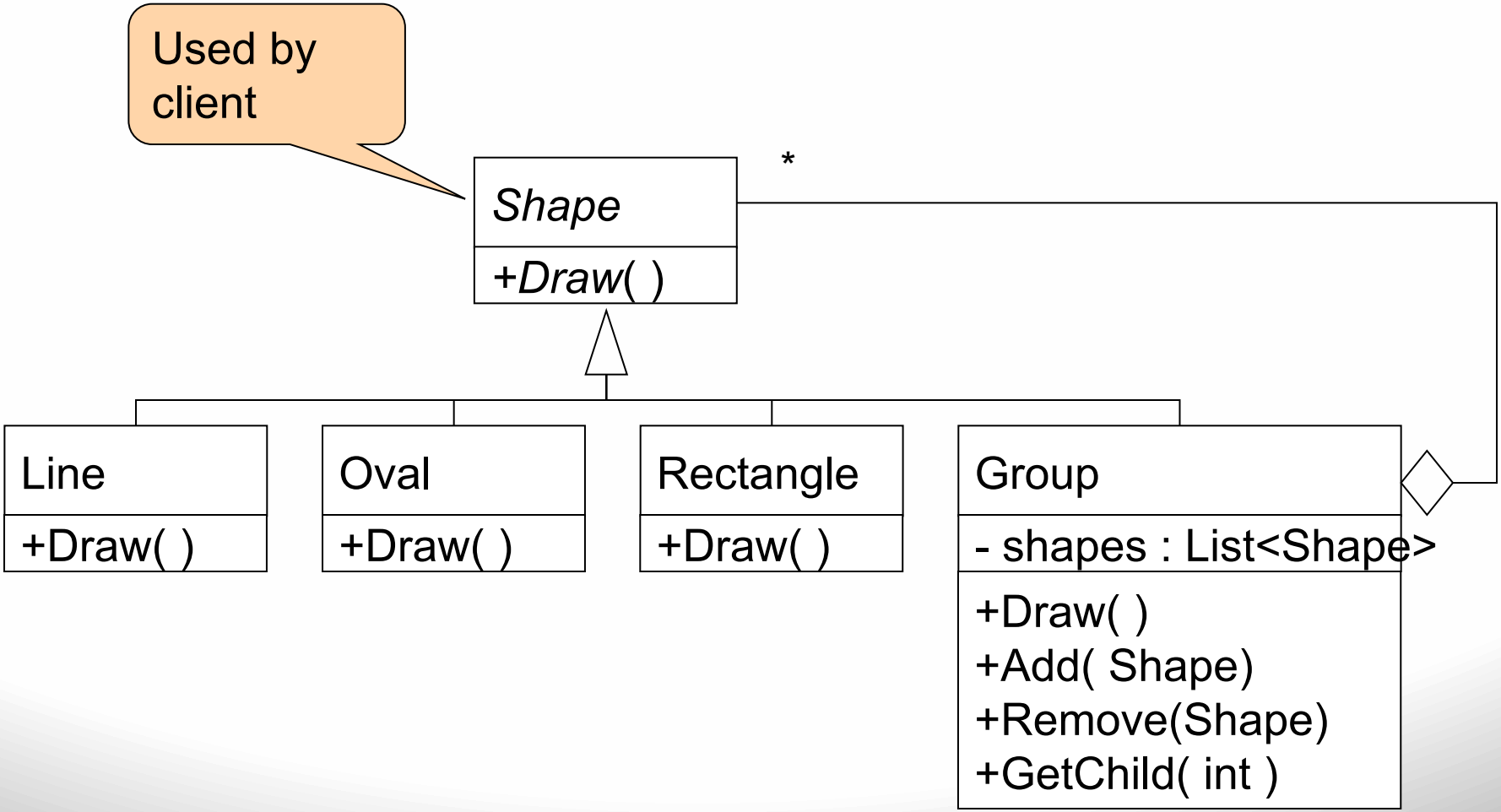
Composite Pattern

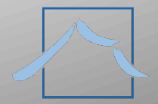
Implement the composite pattern to represent grouped graphical objects.





Exercise #2 Composite Pattern UML

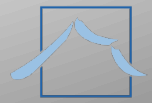




Exercise #2 Composite Pattern

- Get sample exercise from Moodle
- Implement rest of the code as described in UML
- To make it easier, just write the name of the shape in the Draw method. e.g "I draw oval"
- Fill colors for your drawing
- E.g. Drawing with one red Oval, & one blue Oval, green Rectangle.
- For the exercise purpose, just print the name of the color.

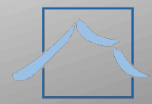
You have 20 minutes!



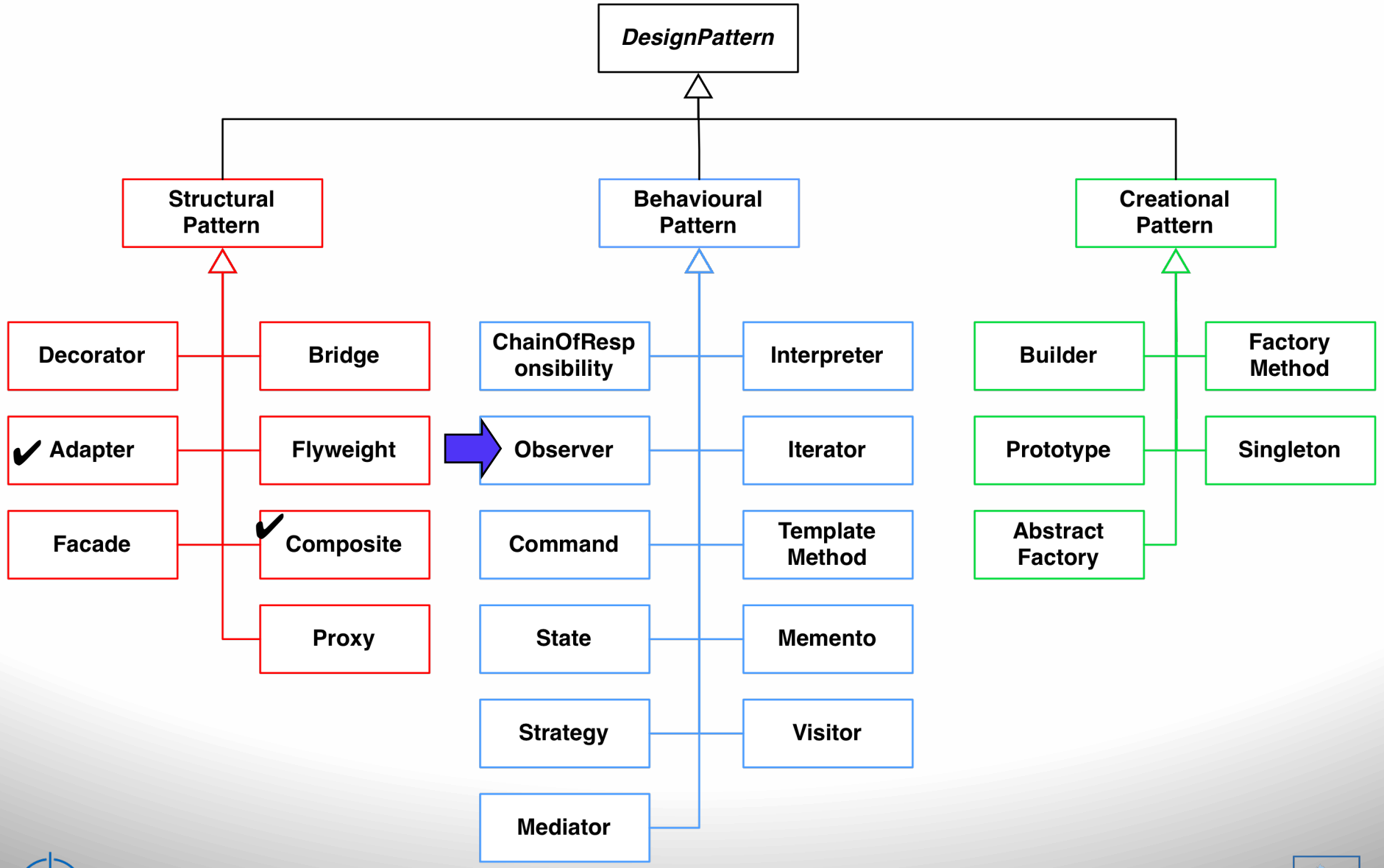
Exercise #2 Composite Pattern Solution

```
public class Group extends Graphic{

        protected ArrayList<Graphic> graphicList = new ArrayList<Graphic>();

        public void addGraphic(Graphic graphic) {
                graphicList.add(graphic);
        }

        public void removeGraphic(Graphic graphic) {
                graphicList.remove(graphic);
        }

        public void draw(){
                System.out.println("Drawing elements of group");
                for(Graphic graphic : graphicList)
                        graphic.draw();
                System.out.println("Finished drawing elements of group");
        }
}
```

For colors, define color attribute in toDraw() method of abstract class Shape.



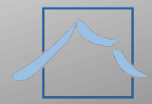
Design Patterns Taxonomy (23 Patterns)



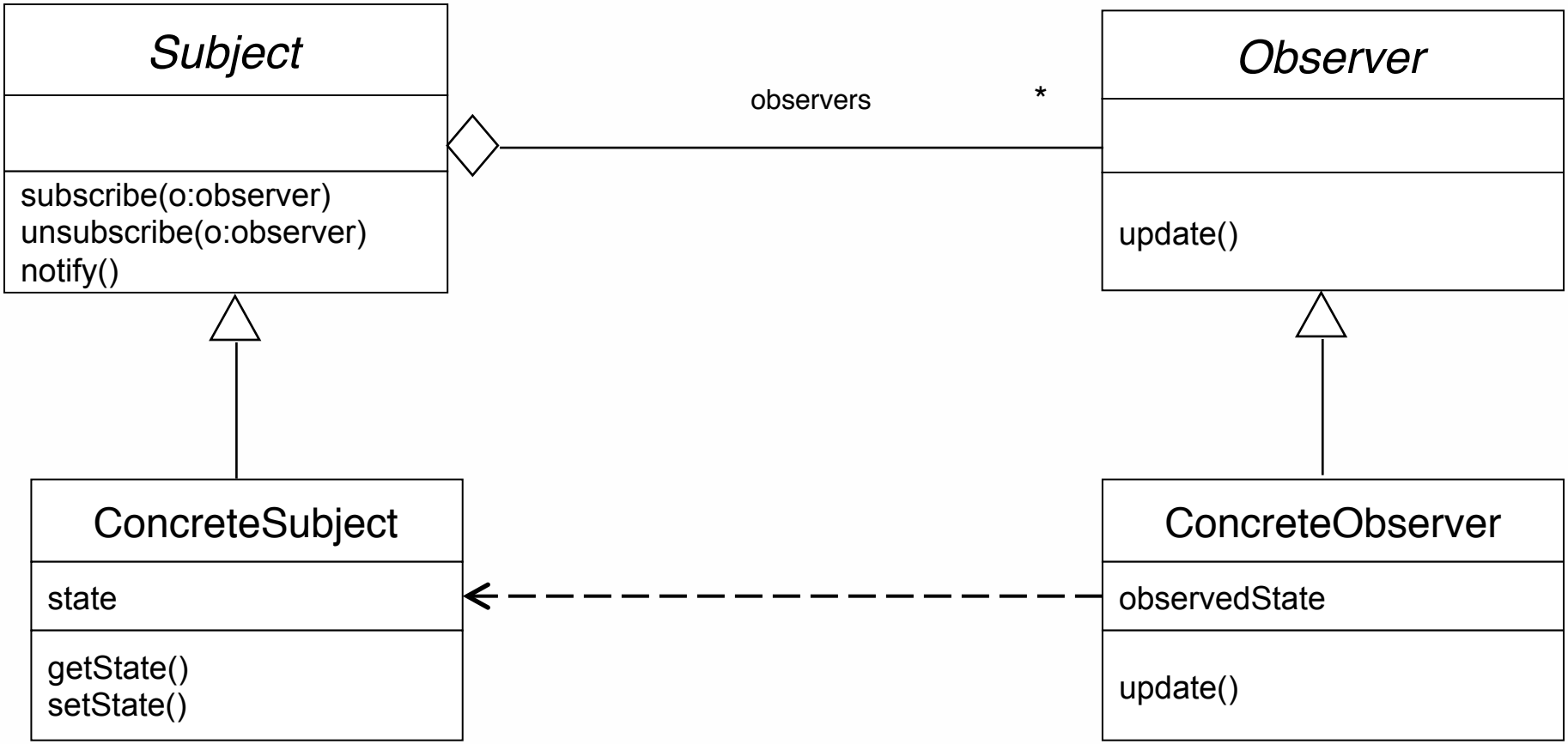
Observer Pattern

Problem:

- We have an *object* that changes its state quite often ***(subject)***
- We have *other objects* that want to be aware of this change ***(observer)***
- The system should maintain consistency across the (redundant) *observers*, whenever the state of the *subject* changes
- The system design should be highly extensible
  - It should be possible to add new *observers,* without having to recompile the observed object or the existing views.



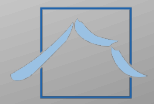
Observer Pattern: Decouples an Abstraction from its Views



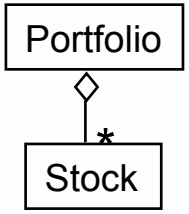
- The Subject represents the observed object
- Observers attach to the Subject by calling subscribe()
- The state is contained in the subclass ConcreteSubject
- The state can be obtained and set by subclasses of type ConcreteObserver.

Observer Pattern

- Models a 1-to-many dependency between objects
  - Connects the state of an observed object, the subject with many observing objects, the observers
- Usage:
  - Maintaining consistency across redundant objects (observers)
  - Optimizing a batch of changes to maintain consistency
- Three variants for maintaining the consistency:
  - Push Notification: Every time the state of the subject changes, *all* the observers are notified of the change
    - Push-Update Notification: The subject also sends the state that has been changed to the observers
  - Pull Notification: An observer inquires about the state the of the subject
- Also called **Publish and Subscribe.**

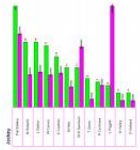
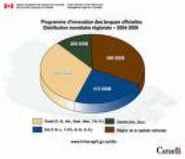


Observer Pattern: A Stock Exchange Example



- We have an *object* that changes its state quite often ***(subject)***
  A Portfolio of Stocks

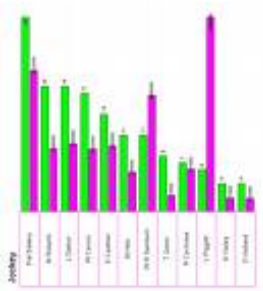
- We have *other objects* that want to be aware of this change ***(observer)***
  Different Views of the Stock Values (Bar Chart, Pie Chart, etc.)

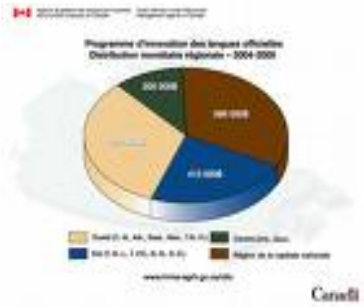




- The system should maintain consistency across the (redundant) *observers*, whenever the state of the *subject* changes
  Whenever the stock values change the stock values should be updated

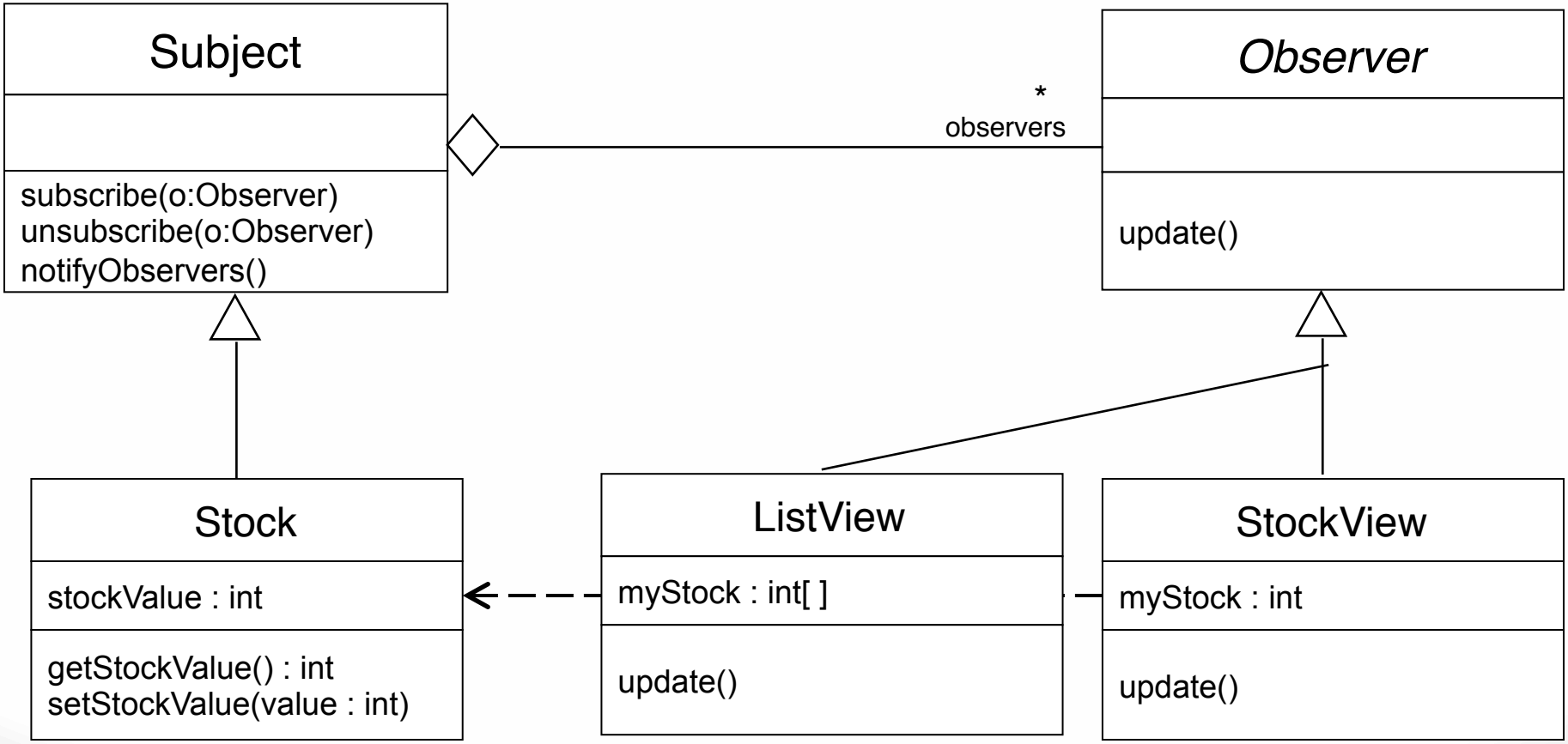
- The system design should be highly extensible
  - It should be possible to add new *observers,* without having to recompile the observed object or the existing views.
    An alarm whenever a value changes

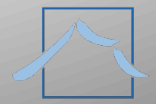
Observer Example Stock Application





Observer Pattern: Stock Exchange Example





Observer Pattern in Java

```
public class Subject {
    private List<Observer> observers
                = new ArrayList<Observer>();
    public void subscribe(Observer o){
           this.observers.add(o);
           o.observedSubject = this;
    }
    public void unsubscribe(Observer o){
           this.observers.remove(o);
    }
    public void notifyObservers(){
           for (Observer o : observers) {
               o.update();
           }
    }
}
```

```
public class Stock extends Subject {
    private int stockValue = 0;

    public int getStockValue() {
     return stockValue;
    }
public void setStockValue(int value) {
   this.stockValue = value;
   notifyObservers();
 }
}
```

```
public abstract class Observer {
     Subject observedSubject;
     public abstract void update();
}
```

```
public class StockView extends Observer {
     int value;
     public void update() {
        value = ((Stock) observedSubject).getStockValue();
        System.out.println ("New value: "+ value);
     }
}
```

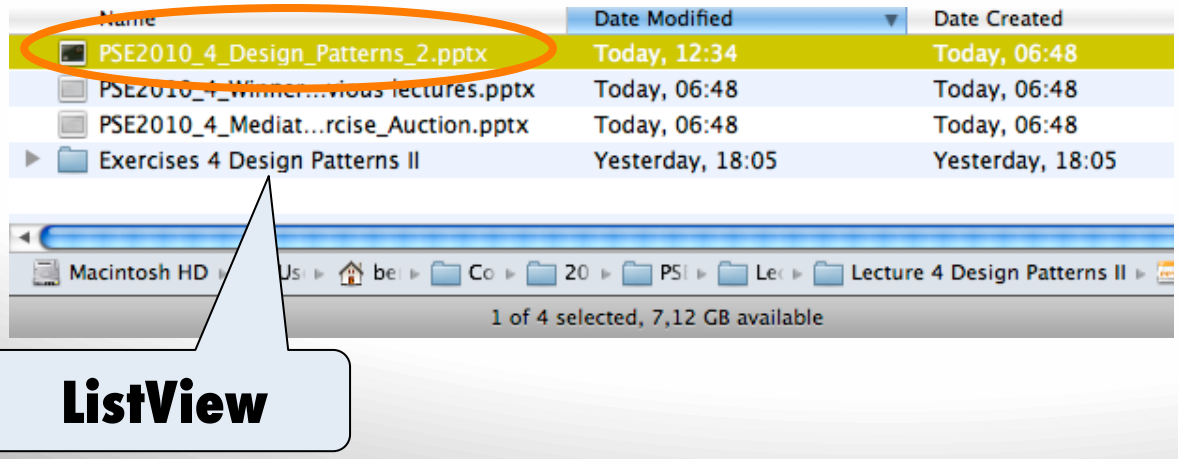
```
public class ListView extends Observer {
     List<Integer> values = new ArrayList<Integer>();

     public void update() {
        newValue = ((Stock) observedSubject).getStockValue();
        values.add(newValue);
        System.out.println("New Value: "+ newValue);     }
}
```

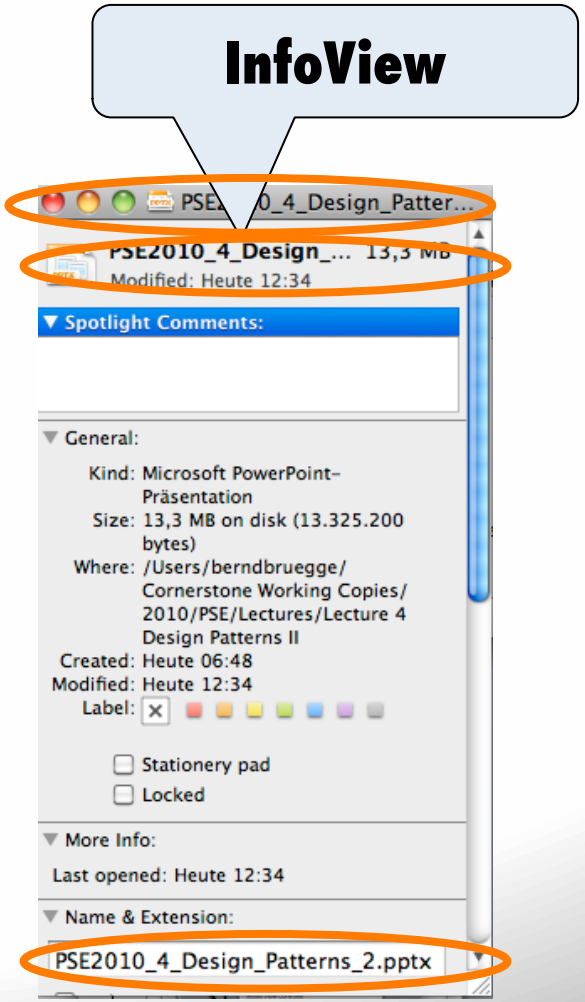
Exercise

Exercise # 3 Observer Pattern

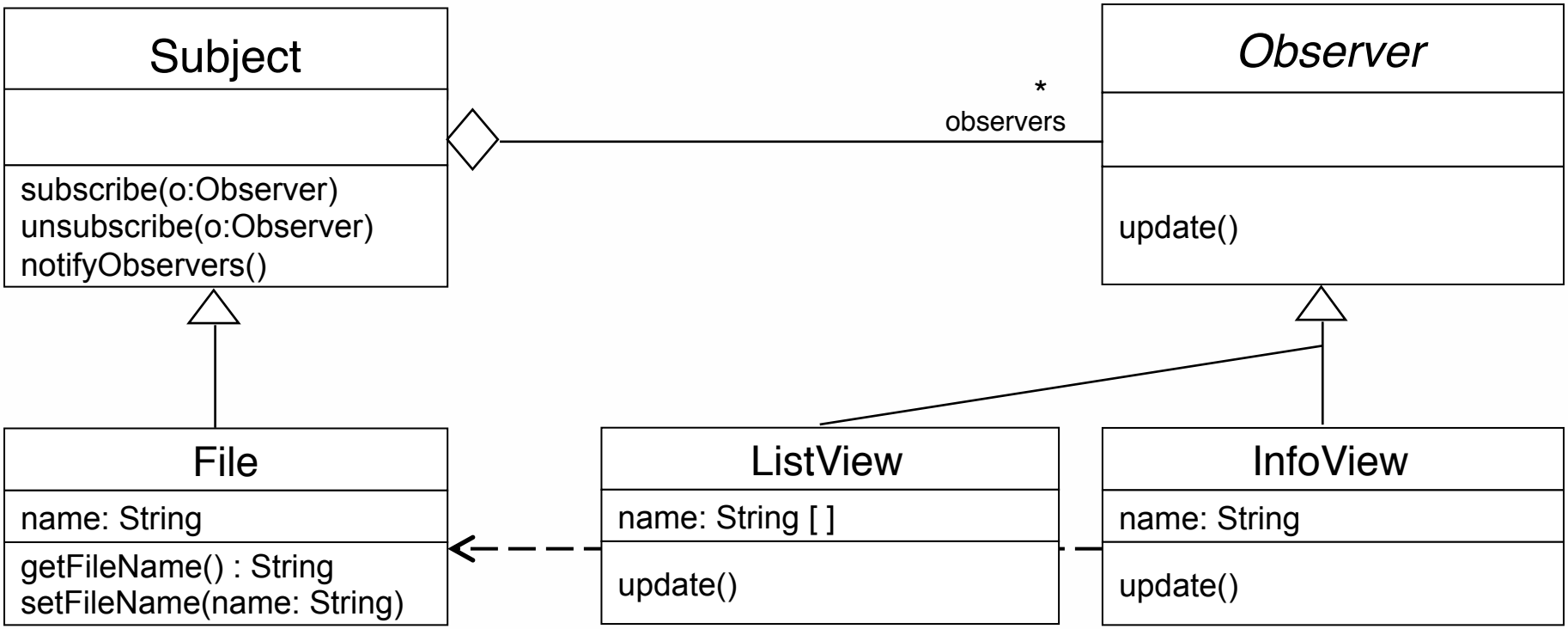
Implement the File class and the following two views using the observer pattern.

When changing file names in different views in an operating system every other view which is observing the file name, should be updated.





Exercise # 3 Observer Pattern: UML



Implementation tips:
When the name of the file is changed the change is reflected on **all views**.
The views should print a message each time the name change notification is received:

- INFOVIEW received an update for the file: NEW NAME 1
- LISTVIEW received an update for the file: NEW NAME 1

Exercise # 3 Observer Pattern: Solution

```
public class Subject {
    private List<Observer> observers  = new
ArrayList<Observer>();
    public void subscribe(Observer o) {
           this.observers.add(o);
           o.observedSubject = this;
    }
    public void unsubscribe(Observer o){
           this.observers.remove(o);
    }
    public void notifyObservers(){
           for (Observer o : observers) {
               o.update();
           }
    }}
```

observes

```
public abstract class Observer {
      Subject observedSubject;
      public abstract void update();
}
```

```
public class ListView extends Observer {
      public void update() {
          String newFilename = ((File)
observedSubject).getFilename();
          System.out.println (" LISTVIEW
received an update for the file: "+
newFilename );
      }
}

public class InfoView extends Observer {
      public void update() {
          String newFilename = ((File)
observedSubject).getFilename();
          System.out.println (" INFOVIEW
received an update for the file: "+
newFilename );
      }
}
```

```
public class File extends Subject {
    private String filename= "";

    public String getFilename() {
           return filename;
    }
public void setFileName(String filename) {
           this.filename = filename;
           notifyObservers();
 }}
```