CSc 133 Lecture Notes

7 - Design Patterns

Computer Science Department

California State University, Sacramento

Overview

- **Background**
- **Types of Design Patterns**
  - **Creational vs. Structural vs. Behavioral Patterns**
- **Specific Patterns**

Composite	***Singleton***
Iterator	***Observer***
Strategy	***Command***
Proxy	***Factory Method***

- **MVC Architecture**

Definition

In software engineering, a software design pattern is a **general reusable solution** to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code

Source: https://en.wikipedia.org/wiki/Software_design_pattern

Background

- A generic, clever, useful, or insightful solution to a set of recurring problems.
- Popularized by 1995 book: ***"Design Patterns: Elements of Reusable Object-Oriented Software"*** by Gamma et. al (the "gang of four").
  … identified the original set of 23 patterns.
- Original concept from architecture:
  ring road, circular staircase etc.

- Code frequently needs to do things been done before.

Types of Design Patterns

- **CREATIONAL**
  - Deal with process of object creation (i.e. **Singleton)**
- **STRUCTURAL**
  - Deal with structure of classes – how classes and objects can be combined to form larger structures
  - Design objects that satisfy constraints
  - Specify connections between objects
  - **i.e. Composite**
- **BEHAVIORAL**
  - Deal with interaction between objects
  - Encapsulate processes performed by objects
  - **i.e. Iterator**

Common Design Patterns

As defined in *Design Patterns: Elements of Reusable Object-Oriented Software*, by Gamma, Helm, Johnson, and Vlissides

Creational:

- Abstract Factory
- Builder
- **Factory Method***
- Prototype
- **Singleton***

Structural:

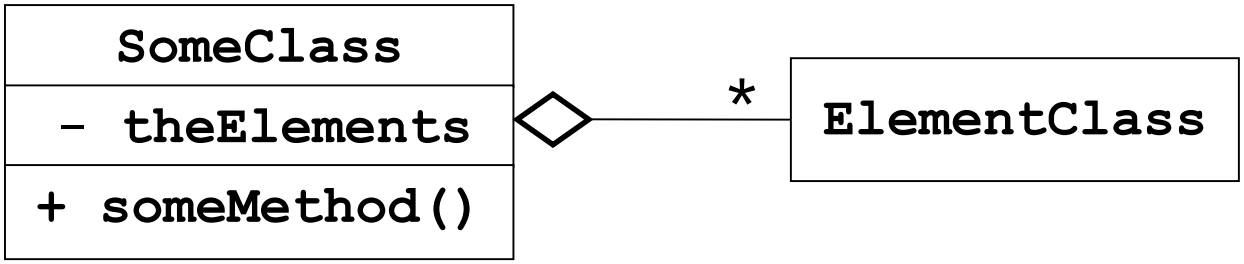
- Adapter
- Bridge
- **Composite***
- Decorator
- Façade
- Flyweight
- **Proxy***

Behavioral:

- Chain of Responsibility
- **Command***
- Interpreter
- **Iterator***
- Mediator
- Memento
- **Observer***
- State
- **Strategy***
- Template Method
- Visitor

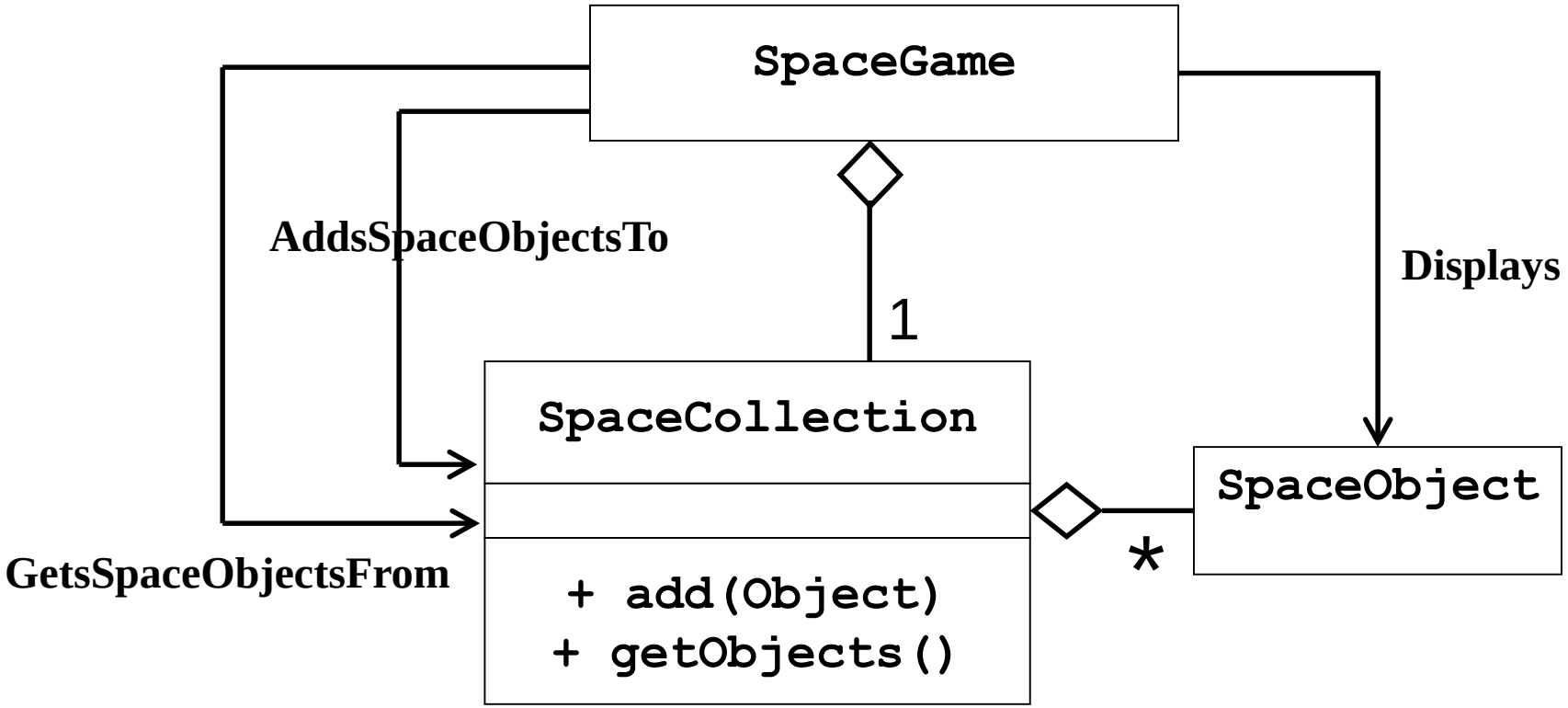
The Iterator Pattern

- MOTIVATION
  - An “aggregate” object contains “elements”
  - “Clients” need to access these elements
  - Aggregate shouldn’t expose internal structure



- Example
  - *GameWorld* has a set of game characters
  - *Screen view* needs to display the characters
  - *Screen view* does not need know the data structure

Collection Classes



SpaceGame Implementation

```
import java.util.Vector;
/** This class implements a game containing a collection of SpaceObjects.
 *  The class has knowledge of the underlying structure of the collection
*/
public class SpaceGame {

    private SpaceCollection theSpaceCollection ;

    public SpaceGame() {

        //create the collection
        theSpaceCollection = new SpaceCollection();

        //add some objects to the collection
        theSpaceCollection.add (new SpaceObject("Obj1"));
        theSpaceCollection.add (new SpaceObject("Obj2"));
        ...
    }

    //display the objects in the collection
    public void displayCollection() {
        Vector theObjects = theSpaceCollection.getObjects();
        for (int i=0; i<theObjects.size(); i++) {
            System.out.println (theObjects.elementAt(i));
        }
    }
}
```

Space Objects

```
/** This class implements a Space object.
 *  Each SpaceObject has a name and a location.
 */
public class SpaceObject {

    private String name;
    private Point location;

    public SpaceObject (String theName) {
      name = theName;
      location = new Point(0,0);
    }

    public String getName() {
      return name;
    }

    public Point getLocation() {

      return new Point (location);
    }

    public String toString() {
      return "SpaceObject " + name + " " + location.toString();
    }
}
```

SpaceCollection – Version #1

```
/**  This class implements a collection of SpaceObjects.
 *   It uses a Vector to hold the objects in the collection.
 */

public class SpaceCollection {

    private Vector theCollection ;

    public SpaceCollection() {
        theCollection = new Vector();
    }

    public void add(SpaceObject newObject) {
        theCollection.addElement(newObject);
    }

    public Vector getObjects() {
        return theCollection ;
    }
}
```

SpaceCollection – Version #2

```
/**  This class implements a collection of SpaceObjects.
 *   It uses a Hashtable to hold the objects in the collection.
 */

public class SpaceCollection {

    private Hashtable theCollection ;

    public SpaceCollection() {
        theCollection = new Hashtable();
    }

    public void add(SpaceObject newObject) {
        // use object's name as the hash key
        String hashKey = newObject.getName();
        theCollection.put(hashKey, newObject);
    }

    public Hashtable getObjects() {
        return theCollection ;
    }
}
```

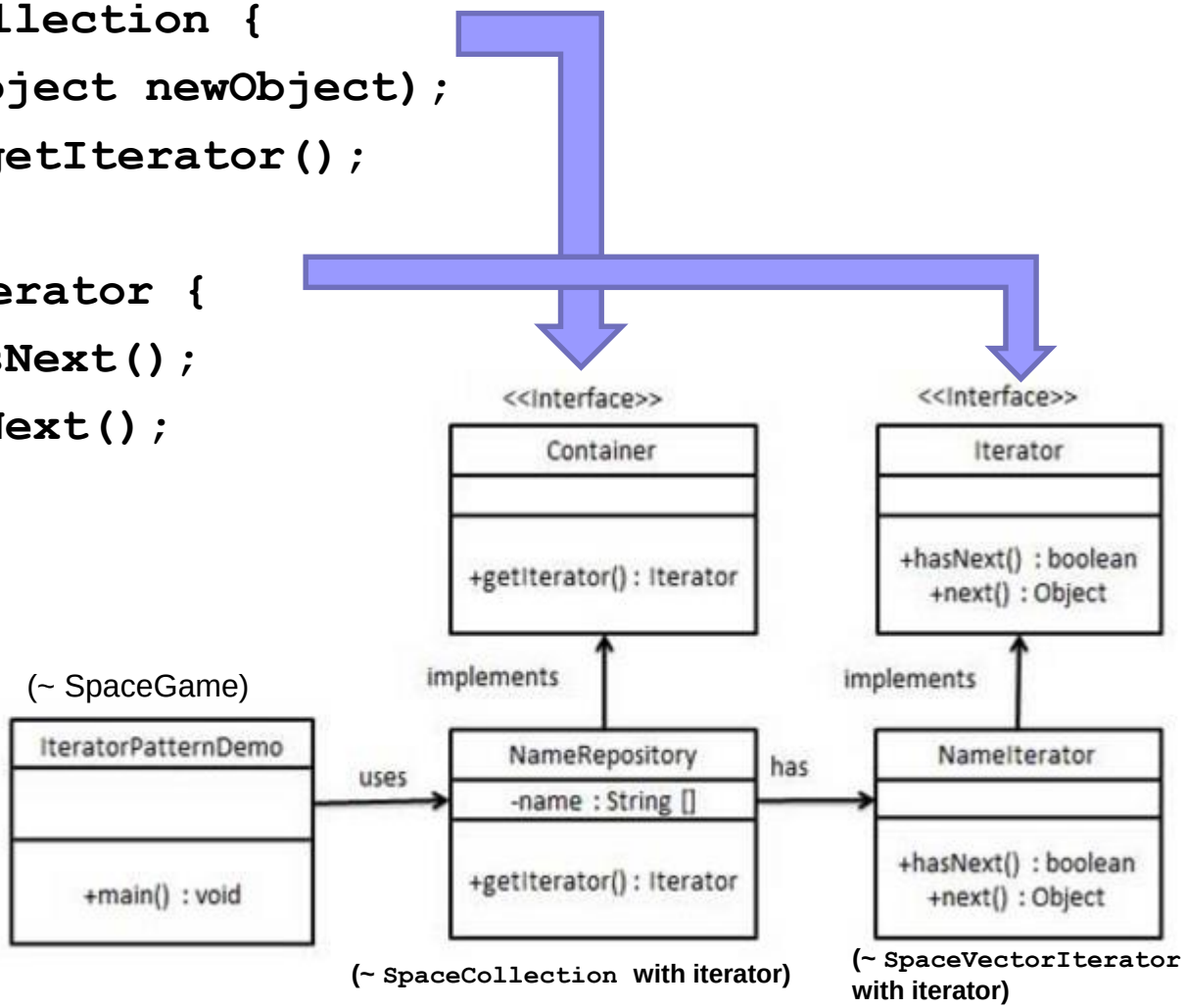
Collections and Iterators

```
public interface ICollection {
   public void add(Object newObject);
   public IIterator getIterator();
}
public interface IIterator {
   public boolean hasNext();
   public Object getNext();
}
```

(~ **`SpaceCollection`** with iterator)

Collections and Iterators

```
public interface ICollection {
   public void add(Object newObject);
   public IIterator getIterator();
}
public interface IIterator {
   public boolean hasNext();
   public Object getNext();
}
```



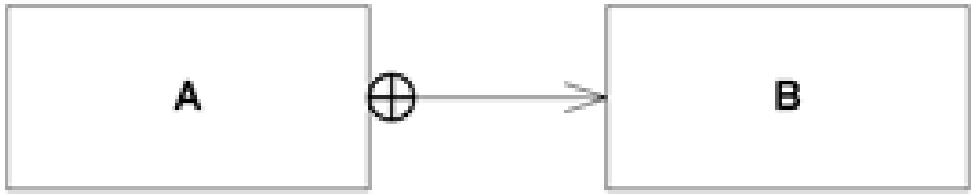
SpaceCollection With Iterator

```
/** This class implements a collection of SpaceObjects.
 *  It uses a Vector as the structure but does
 *  NOT expose the structure to other classes.
 *  It provides an iterator for accessing the
 *  objects in the collection.
 */

public class SpaceCollection implements ICollection {

    private Vector theCollection ;

    public SpaceCollection() {
        theCollection = new Vector ( );
    }

    public void add(Object newObject) {
        theCollection.addElement(newObject);
    }

    public IIterator getIterator() {
        return new SpaceVectorIterator ( ) ;
    }

    ...continued...
```

SpaceCollection With Iterator (cont.)

```
private class SpaceVectorIterator implements IIterator {
    private int currElementIndex;

    public SpaceVectorIterator() {
      currElementIndex = -1;
    }

    public boolean hasNext() {
        if (theCollection.size ( ) <= 0)  return false;
        if (currElementIndex == theCollection.size() - 1 )
          return false;
        return true;
    }

    public Object getNext ( ) {
        currElementIndex ++ ;
        return(theCollection.elementAt(currElementIndex));
    }
  } //end private iterator class
} //end SpaceCollection class
```

UML Notation for an Inner Class




```
public class A {
   private class B {
      ...
   }
}
```

Using An Iterator

```
/** This class implements a game containing a collection of SpaceObjects.
 *  The class assumes no knowledge of the underlying structure of the
 *  collection -- it uses an Iterator to access objects in the collection.
 */

public class SpaceGame {

    private SpaceCollection theSpaceCollection ;

    public SpaceGame() {

        //create the collection
        theSpaceCollection = new SpaceCollection();

        //add some objects to the collection
        theSpaceCollection.add (new SpaceObject("Obj1"));
        theSpaceCollection.add (new SpaceObject("Obj2"));
        ...
    }

    //display the objects in the collection
    public void displayCollection() {
        IIterator theElements = theSpaceCollection.getIterator() ;
        while ( theElements.hasNext() ) {
            SpaceObject spo = (SpaceObject) theElements.getNext() ;
            System.out.println ( spo ) ;
        }
    }
}
```

Quiz:
Which method get called here?

CN1's Iterator Interface

`boolean hasNext()`

Returns true if the collection has more elements.

`Object next()`

Returns the next element in the collection.

`void remove()`

Removes from the collection the last element returned by the iterator. Can only be called once after **`next()`** was called. Optional operation. Exception is thrown if not supported or **`next()`** is not properly called.

CN1's Collection Interface

boolean add(**Object** o) : Ensures that this collection contains the specified element

boolean addAll(**Collection** c) : Adds all of the elements in the specified collection to this collection

void clear() : Removes all of the elements from this collection

boolean contains(**Object** o) : Returns true if this collection contains the specified element.

boolean containsAll(**Collection** c) : Returns true if this collection contains all of the elements in the specified collection.

boolean equals(**Object** o) : Compares the specified object with this collection for equality.

int hashCode() : Returns the hash code value for this collection.

boolean isEmpty() : Returns true if this collection contains no elements.

Iterator iterator() : Returns an iterator over the elements in this collection.

boolean remove(**Object** o) : Removes a single instance of the specified element from this collection, if it is present

boolean removeAll(**Collection** c) : Removes all this collection's elements that are also contained in the specified collection

boolean retainAll(**Collection** c) : Retains only the elements in this collection that are contained in the specified collection

int size() : Returns the number of elements in this collection.

Object[] toArray() : Returns an array containing all of the elements in this collection.

Object[] toArray(**Object**[] a) : Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array.

CN1's Iterable Interface

- **Implementing this interface allows an object to be the target of the "*foreach*" statement…**

```
interface Iterable {
   public Iterator iterator();        // "getIterator()"
}
```

- **Example:**

```
public class SpaceCollection implements Iterable {
   ...
   public Iterator iterator() {
      return new SpaceVectorIterator(); //as before
   }
}
public class SpaceGame {
   ...
   public void displayCollection() {
      for (Object spo : theSpaceCollection){    //"foreach"
         System.out.println (((SpaceObject)spo).getName());
      }
   }
}
```

(CN1 `Collection` interface is a subinterface of CN1 `Iterable` interface.)

Recap: The Iterator Pattern

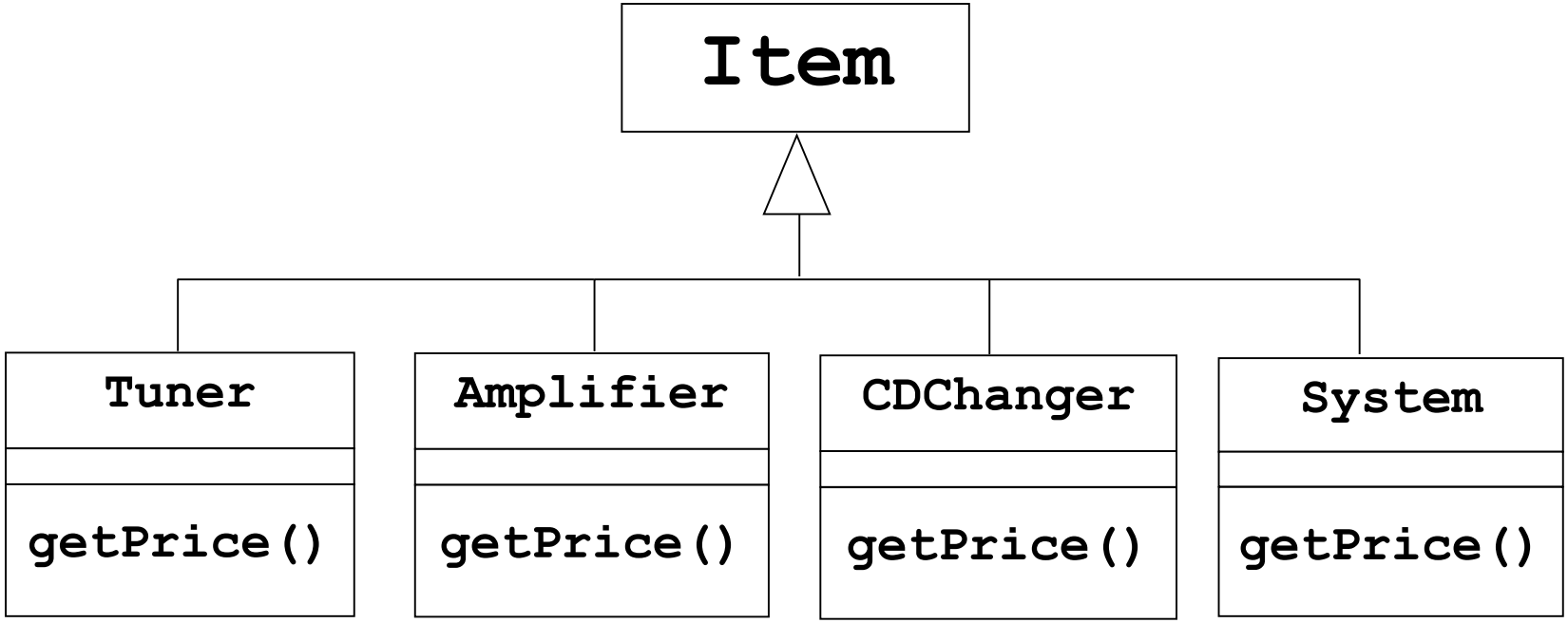
- This pattern is used to get a way to access the elements of a collection object in sequential manner without any need to know its underlying representation.
- Iterator pattern falls under behavioral pattern category.

The Composite Pattern

- **MOTIVATION:**
  - Objects organized in a hierarchical manner
  - Some objects are *groups* of the other objects
  - Individuals and groups need to be treated uniformly
- **Example:**
  - A store sells stereo component items:

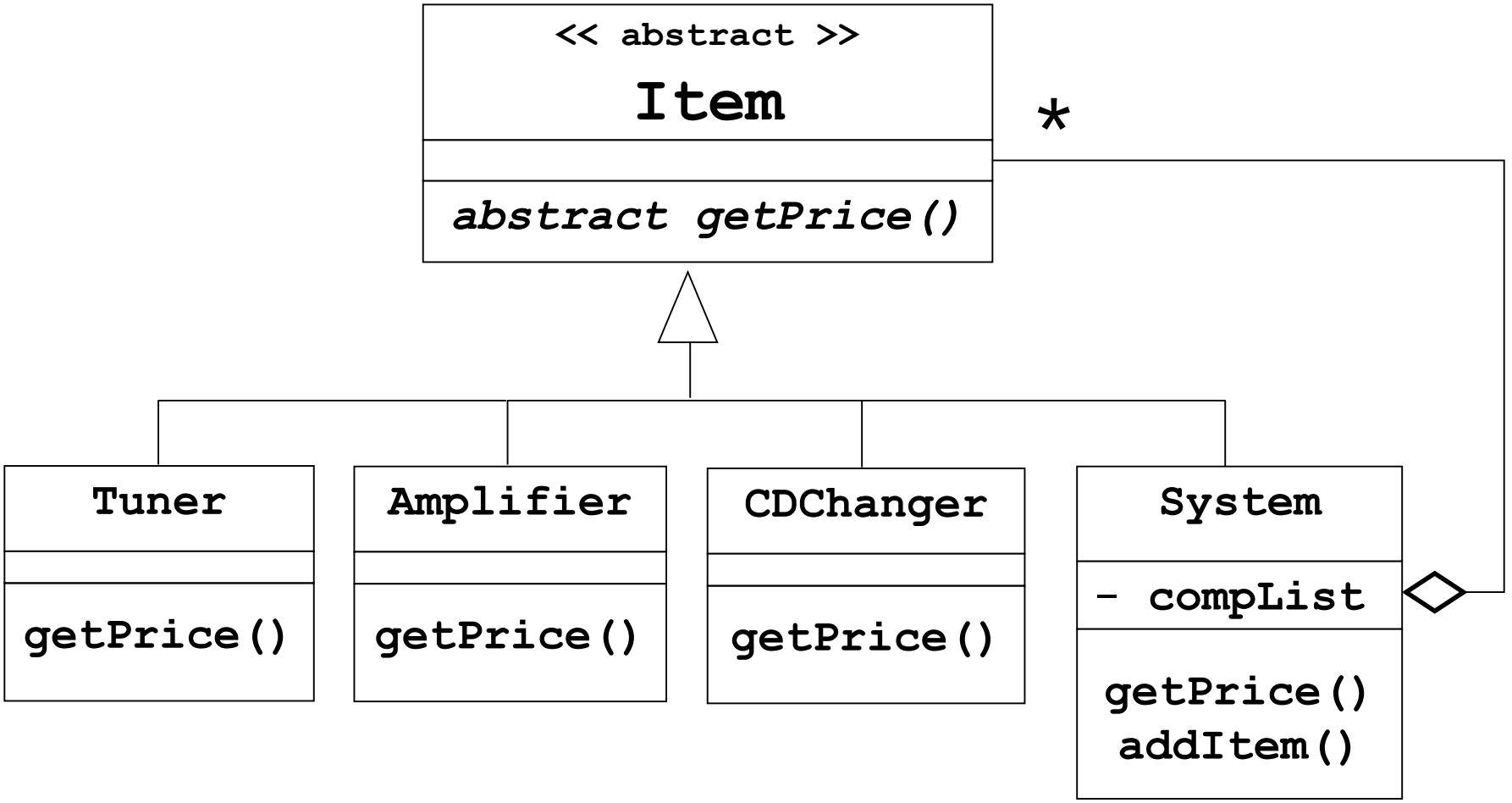
    Tuners, Amplifiers, CDChangers, etc.
    - Each item has a **`getPrice()`** method
  - The store also sells complete stereo systems
    - Systems also have a **`getPrice()`** method which returns a discounted sum of the prices.

Possible Class Organization

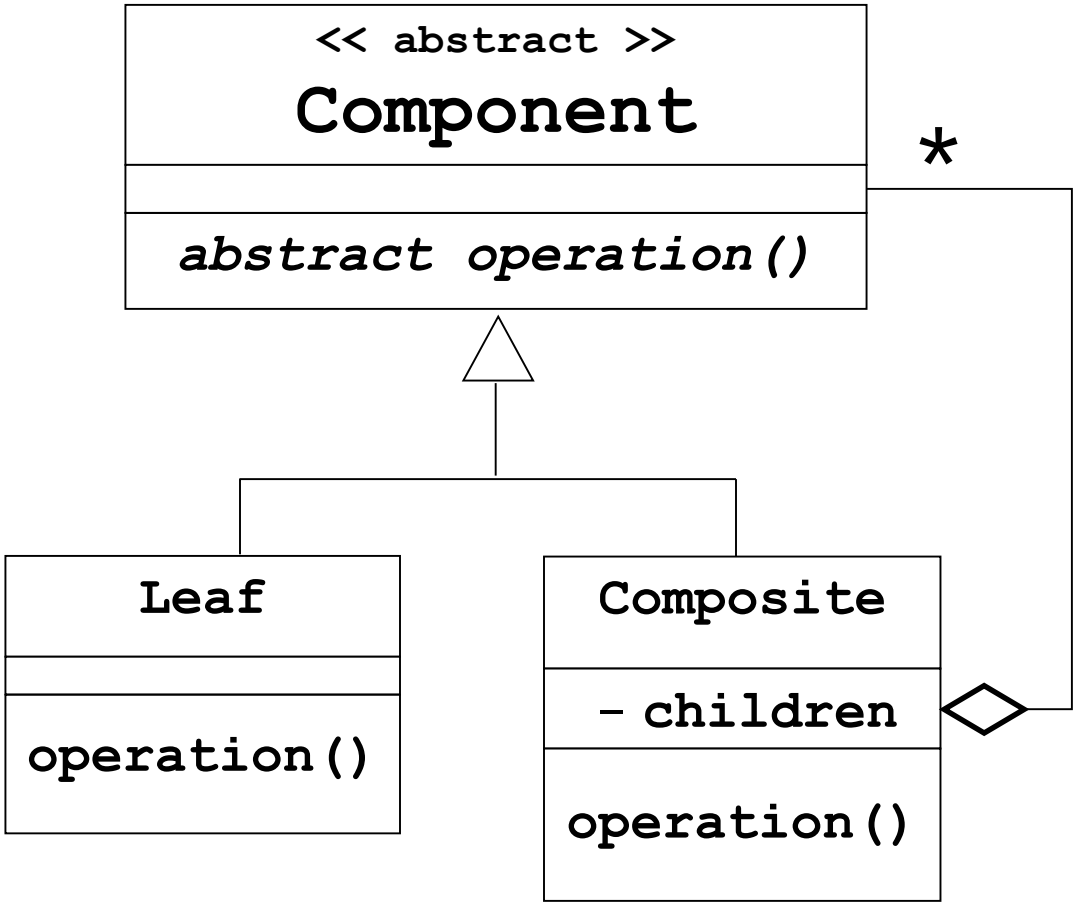


Problem ?

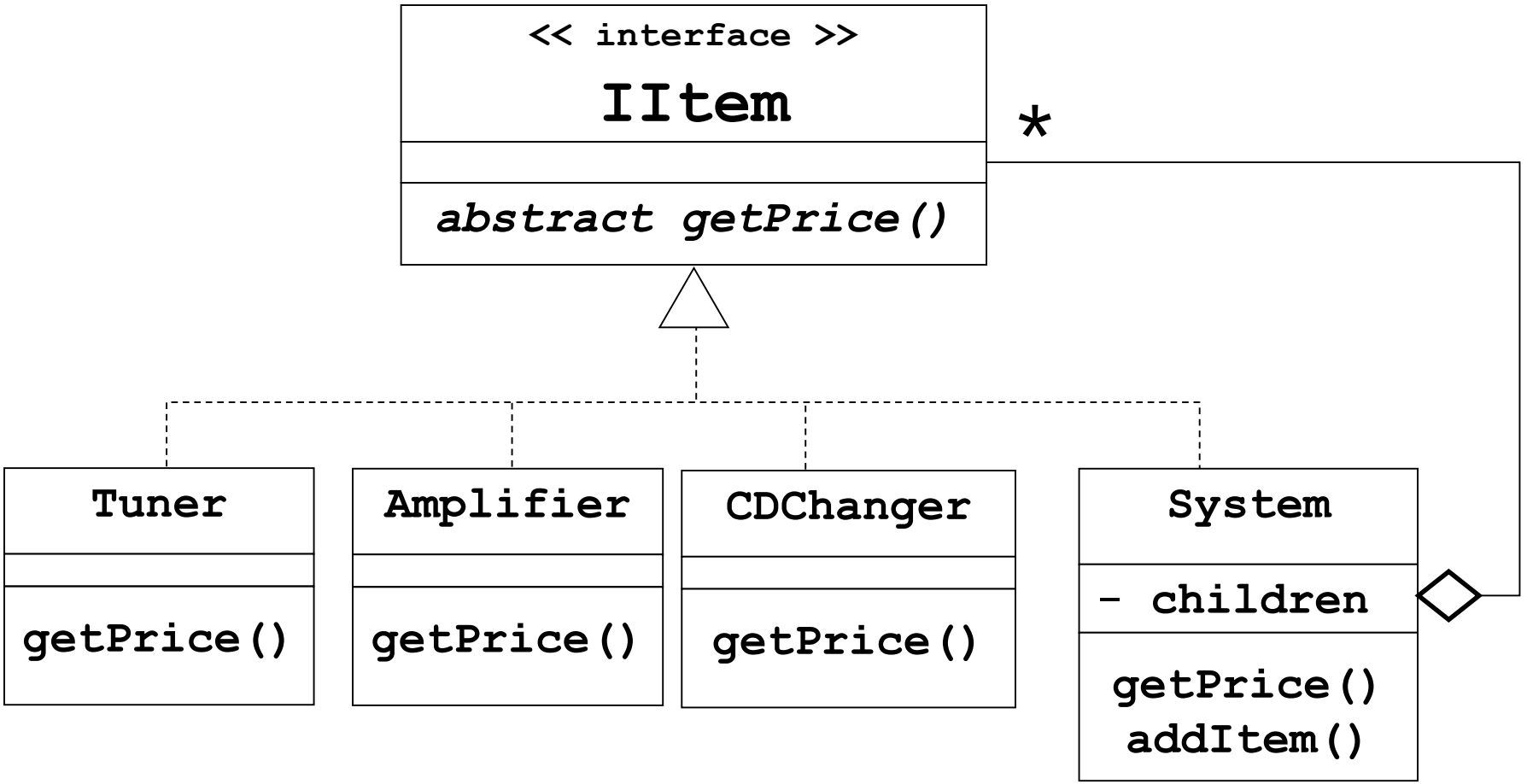
Solution


<< abstract >>
Item
abstract getPrice()
*
Tuner
getPrice()
Amplifier
getPrice()
CDChanger
getPrice()
System
- compList
getPrice()
addItem()

Composite Pattern Organization



Composite Specified With *Interfaces*



Other Examples Of Composites

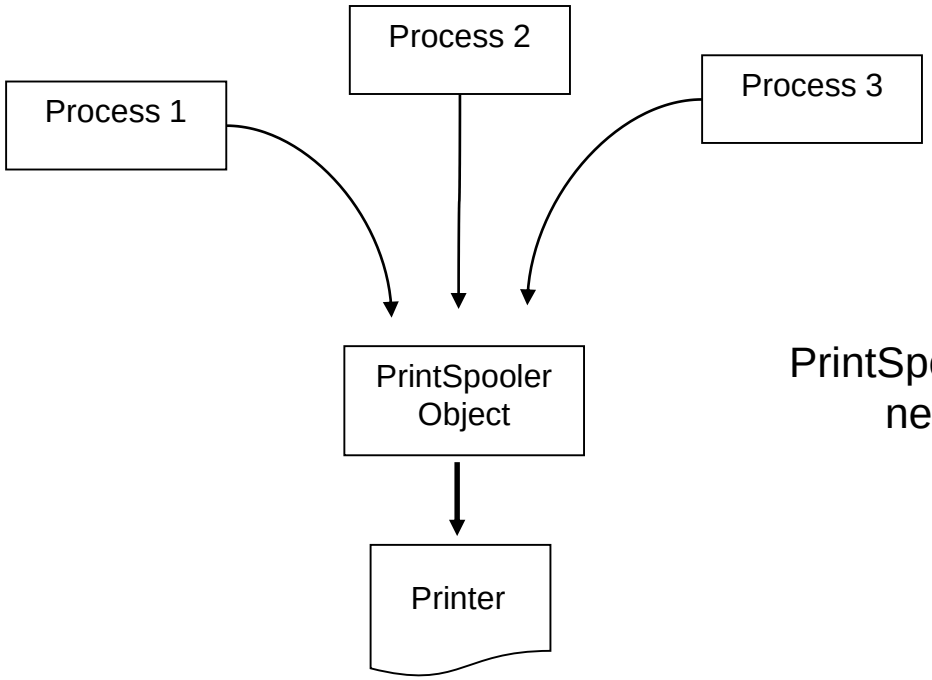
- Trees
  - Internal nodes (groups) and leaves
- Arithmetic expressions
  - `<exp> ::= <term> | <term> "+" <exp>`
- **Graphical Objects**
  - **Rectangles, lines, circles**
  - **Frames**
    - **can contain other graphical objects**

The Singleton Pattern

- **Definition**
  - The **singleton pattern** is a software design pattern that restricts the instantiation of a class to one object.
- **Motivation**
  - Insure a class never has more than one instance at a time
  - Provide public access to instance creation
  - Provide public access to current instance
- **Examples**
  - Print spooler
  - Audio player

PrintSpooler Example

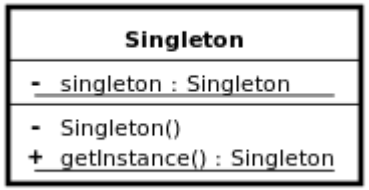
Multiple processes should not access a single printer simultaneously




```
PrintSpooler mySpooler =
    new PrintSpooler();
```

Singleton Implementation

```
public class PrintSpooler {

    // maintain a single global reference to the spooler
    private static PrintSpooler theSpooler;

    // insure that no one can construct a spooler directly
    private PrintSpooler()    { }

    // provide access to the spooler, creating it if necessary
    public static PrintSpooler getSpooler() {
      if (theSpooler == null)
            theSpooler =  new PrintSpooler();
      return theSpooler;
    }

    // accept a Document for printing
    public void addToPrintQueue (Document doc) {
      //code here to add the Document to a private queue ...
    }

    //private methods here to dequeue and print documents ...
}
```



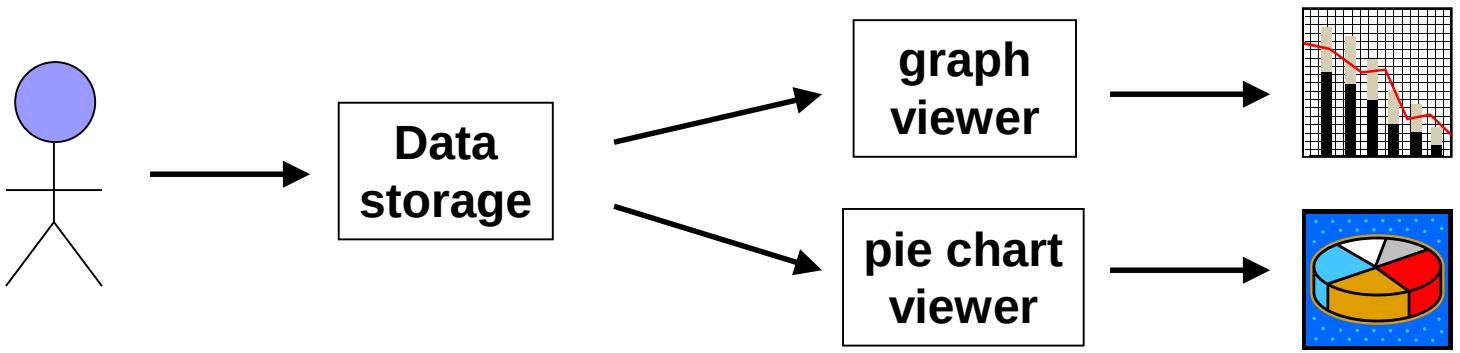
Class diagram exemplifying the singleton pattern.

Source: https://en.wikipedia.org/wiki/Singleton_pattern

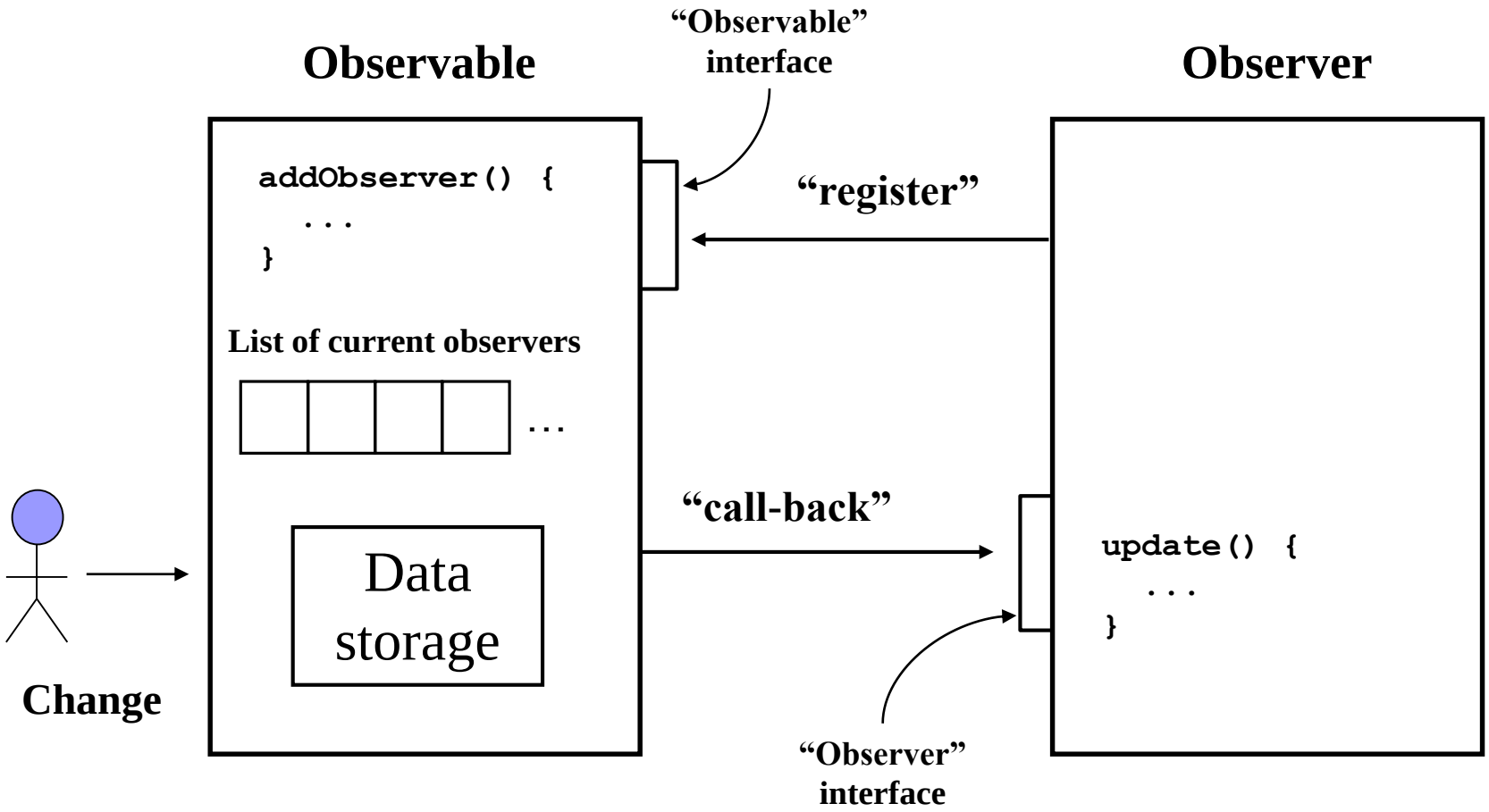
The Observer Pattern

Motivation

- An object stores data that changes regularly
- Various clients use the data in different ways
- Clients need to know when the data changes
- Code that is associated with the object that stores data should not need to change when new clients are added



The Observer Pattern (cont.)



Responsibilities

- Observables must
  - Provide a way for observers to “register”
  - Keep track of who is “observing” them
  - *Notify observers* when something changes
- Observers must
  - Tell observable it wants to be an observer *(“register”)*
  - Provide a method for the *callback*
  - Decide what to do when notified an observable has changed

Implementing Observer/Observable

```
public interface Observer { //build-in CN1 interface
   public void update (Observable o, Object arg);
}
```

```
public interface IObservable { //user-defined interface
   public void addObserver (Observer obs);
   public void notifyObservers();
}
```

OR...

```
public class Observable extends Object { //build-in CN1 class
   public void addObserver (Observer obs) {...}
   public void notifyObservers() {...}
   protected void setChanged() {...}
   ...
}
```

Implementing Observer/Observable (cont.)

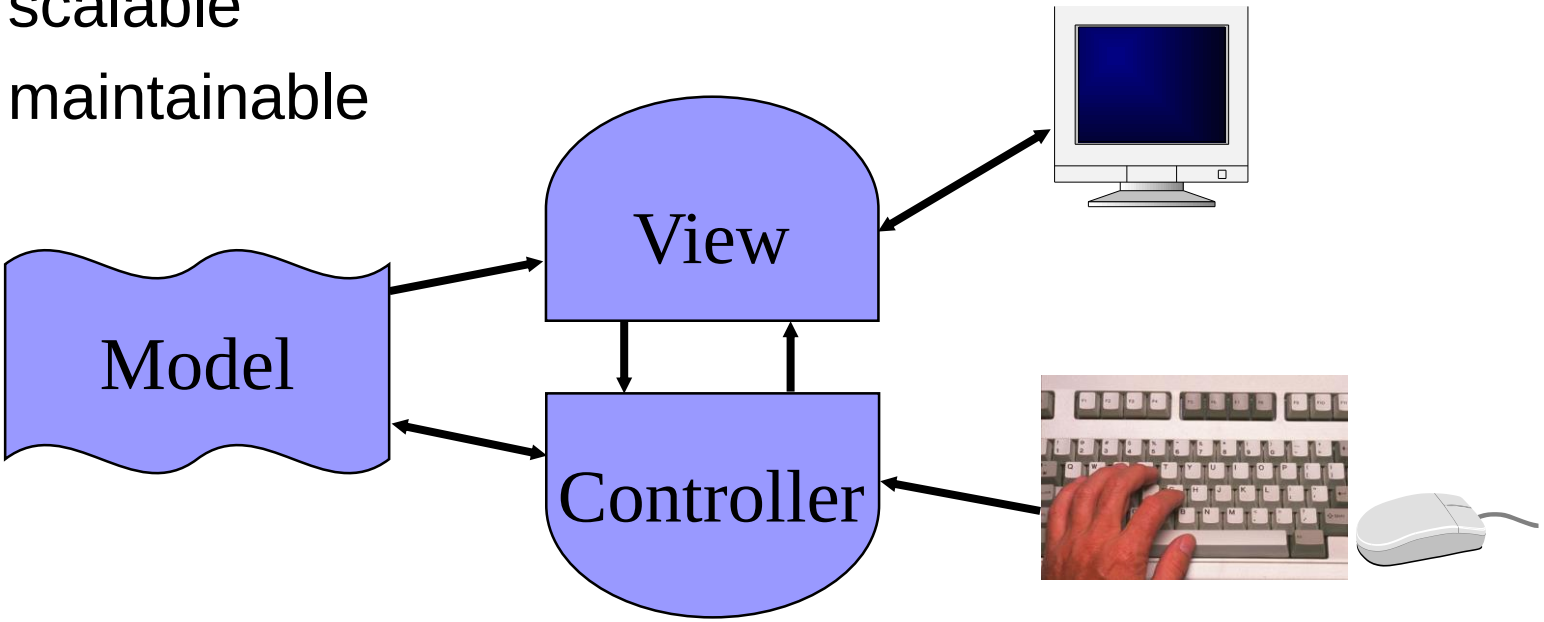
About extending from a build-in Observable class:

```
public class Observable extends Object { //build-in CN1 class
   public void addObserver (Observer obs) {...}
   public void notifyObservers() {...}
   protected void setChanged() {...}
   ...
}
```

- Advantage: Provides code for **`notifyObservers()`** and **`addObserver()`**
- Disadvantage: You cannot extend from another class
- Make sure you call **`setChanged()`** before calling **`notifyObservers()`**
- **`notifyObservers()`** automatically calls **`update()`** on the list of observers that is created by **`addObserver()`**

Model-View-Controller design pattern

- Architecture for interactive apps
  - introduced by Smalltalk developers at PARC
- Partitions application so that it is
  - scalable
  - maintainable



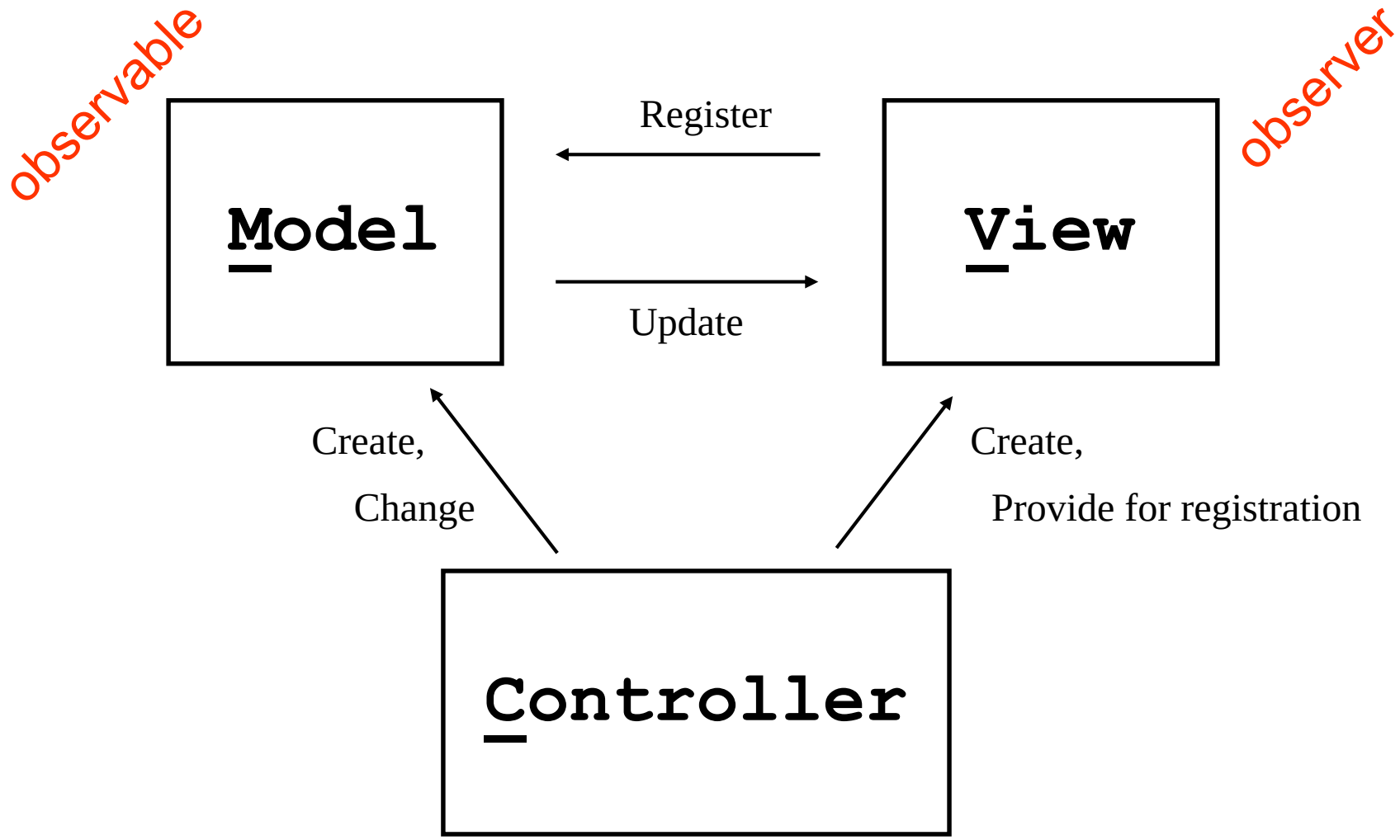
Model-View-Controller design pattern (Cont)

Component	Purpose	Description
Model	Maintain data	Business logic plus one or more data sources such as a database.
View	Display all or a portion of the data	The user interface that displays information about the model to the user.
Controller	Handle events that affect the model or view	The flow-control mechanism means by which the user interacts with the application.

Model-View-Controller design pattern (Cont)

Component	In our assignment # 1 context (Fall 2017)
Model	A game in turn contains several components, including (1) a GameWorld which **holds** a collection of **game objects** and other **state variables**, and (2) a play() method to accept and execute user commands. Later, we will learn that a component such as **GameWorld** that holds the program's data is often called a model.
View	In this first version of the program the top-level Game class will also be responsible for **displaying** information about the state of the game. In future assignments we will learn about a separate kind of component called a **view** which will assume that responsibility.
Controller	The **top-level Game class** also manages the flow of control in the game (such a class is therefore sometimes called a controller). The controller enforces rules such as what actions a player may take and what happens as a result. This class accepts input in the form of keyboard commands from the human player and invokes appropriate methods in the game world to perform the requested commands – that is, to manipulate data in the game model.

MVC Architecture



Skeleton Code

```
public class Controller {
   private Model model;
   private View v1;
   private View v2;
   public Controller () {
      model = new Model();    // create "Observable" model
      v1 = new View(model);   // create an "Observer" that registers itself
      v2 = new View();        // create another "Observer"
      model.addObserver(v2); // register the observer
   }
   // methods here to invoke changes in the model
}

public class Model extends Observable { // OR implements IObservable {
   // declarations here for all model data...
   // methods here to manipulate model data, etc.
   // if implementing IObservable, also provide methods that handle observer
   // registration and invoke observer callbacks
}

public class View implements Observer {
   public View(Observable myModel) {   // this constructor also
      myModel.addObserver(this);       // registers itself as an Observer
   }
   public View ()
   { } // this constructor assumes 3rd-party Observer registration
   public update (Observable o, Object arg) {
      // code here to output a view based on the data in the Observable
   }
}
```

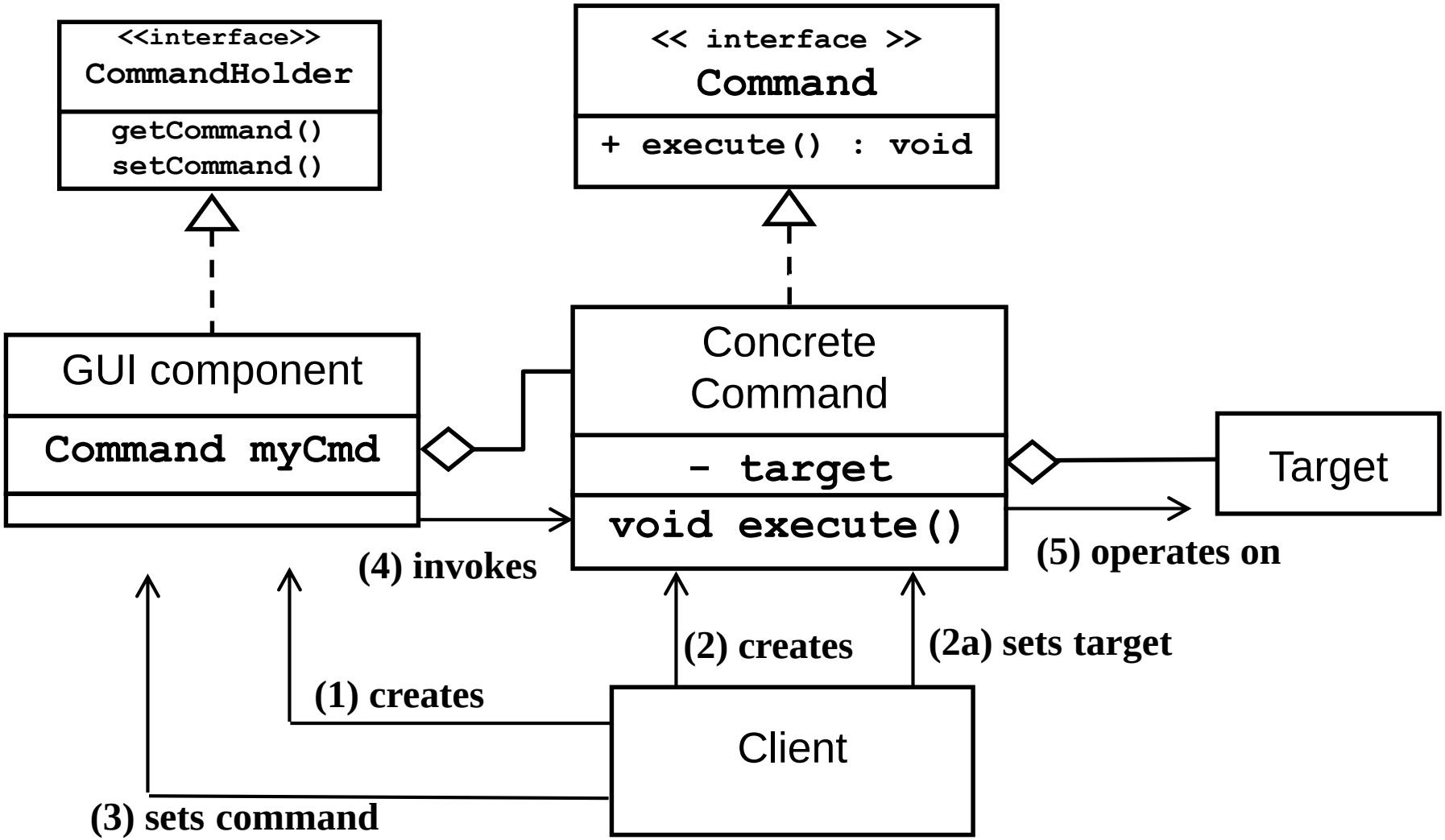
Part II – Design Pattern

The Command Pattern

Motivation

- Need to avoid having multiple copies of the code that performs the same operation invoked from different sources
- Desire to separate code implementing a command from the object which invokes it
- Need for maintaining *state information* about the command
  - Enabled or disabled?
  - Other data – e.g. invocation *count*

Command Pattern Organization



CN1 Command Class

- Implements **ActionListener** interface.
  - Provides empty body implementation for: **actionPerformed()** == "**execute()**"
  - We need to extend from **Command** and override **actionPerformed()** to perform the operation we would like to execute. In the constructor, do not forget to call super("command name")
- Also defines methods like: **isEnabled(), setEnabled(), getCommandName()**
- You can add a command object as a listener to a component using one of its **addXXXListener()** methods which takes **ActionListener** as a parameter (e.g. **addPointerPressedListener()** in **Component**, **addActionListener()** in **Button, addKeyListener()** in **Form**)
- When activated (button pushed, pointer/key pressed etc), component calls **actionPerformed()** method of its listener/command

CN1 `Command` Class (cont.)

Using the **`addKeyListener()`** of **`Form`**, we can attach a listener (an object of a listener class which implements **`ActionListener`** or an object of sub-class of **`Command`**) to a certain key.

This is called **key binding**: we are binding the listener/command (more specifically: the operation defined in its **`actionPerformed()`** method) to the key stroke, e.g:

```
/* Code for a form that uses key binding
//... [create a listener object called myCutCommand]
addKeyListener('c', myCutCommand);
//[when the 'c' key is hit, actionPerformed() method of CutCommand is called]
```

CN1 `Button` Class

`Button` is a "command holder"

- Defines methods like: **`setCommand(), getCommand()`**
- If you use **`setCommand()`** you do not need to also call **`addActionListener()`** since the command is *automatically added as listeners on the button*
- **`setCommand()`** changes the label of the button to the "command name" specified in command's construction

To use the command design pattern properly on buttons, add the command object to the button using **`setCommand()`** (instead of **`addActionListener()`**).

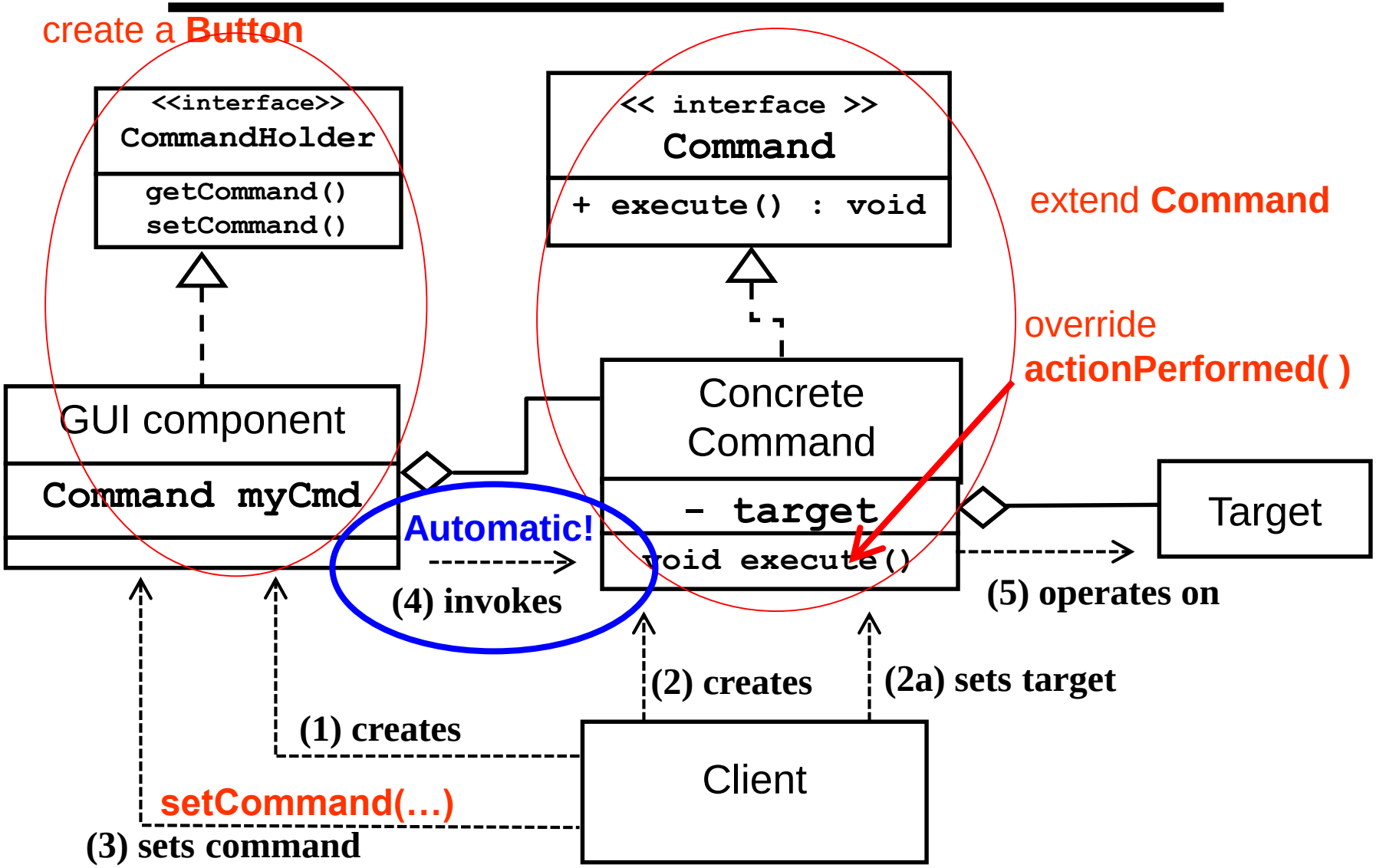
Remember **`CheckBox`** is-a **`Button`** too!

Adding Commands to Title Bar

When you add a regular command (e.g., without specifying a "SideComponent" property) to the title bar area using **`Toolbar`**'s **`addCommandToXXX()`** methods:

- an item (side/overflow menu item or a title bar area item) is automatically generated and added to the title bar area
- The command automatically becomes the listener of the item

Command Pattern – CN1



Summary of Implementing Command Design Pattern in CN1

- **Define your command classes:**
  - Extend **`Command`** (which implements **`ActionListener`** interface and provides empty body implementation of **`actionPerformed()`**)
  - Override **`actionPerformed()`**
- **Add a Toolbar and buttons to your form**
- **Instantiate command objects in your form**
- **Add command objects to various entities:**
  - buttons w/ **`setCommand()`**, title bar area items w/ **`Toolbar's addCommandToXXX()`** methods, key strokes w/ **`Form`**'s **`addKeyListener()`**

Implementing Command Design Pattern in CN1

```
/** This class instantiates several command objects, creates several GUI
 * components (button, side menu item, title bar item), and attaches the command objects
 * to the GUI components and keys.  The command objects then automatically get invoked
 * when the GUI component or the key is activated.
 */

public class CommandPatternForm extends Form {
  public CommandPatternForm () {
    //...[set a Toolbar to form]
   Button buttonOne = new Button("Button One");
   Button buttonTwo = new Button("Button Two");
    //...[style and add two bottons to the form]
   //create command objects and set them to buttons, notice that labels of buttons
   //are set to command names
   CutCommand myCutCommand = new CutCommand();
   DeleteCommand myDeleteCommand = new DeleteCommand();
   buttonOne.setCommand(myCutCommand);
   buttonTwo.setCommand(myDeleteCommand);
    //add cut commnand to the right side of title bar area
   myToolbar.addCommandToRightBar(myCutCommand);
   //add delete command to the side menu
   myToolbar.addCommandToSideMenu(myDeleteCommand);
    //bind 'c' ket to cut command and 'd' key to delete command
   addKeyListener('c', myCutCommand);
   addKeyListener('d', myDeleteCommand);
   show();
   }
}
```

Implementing Command Design Pattern in CN1 (cont.)

```
/** These classes define a Command which perform "cut" and "delete" operations.
 *  The commands are implemented as a subclass of Command, allowing it
 *  to be added to any object supporting attachment of Commands.
 *  This example does not show how the "Target" of the command is specified.
 */

public class CutCommand extends Command{
  public CutCommand() {
    super("Cut"); //do not forget to call parent constructor with command_name
  }
  @Override //do not forget @Override, makes sure you are overriding parent method
  //invoked to perform the 'cut' operation
  public void actionPerformed(ActionEvent ev){
    System.out.println("Cut command is invoked...");
  }
}
public class DeleteCommand extends Command{
  public DeleteCommand() {
    super("Delete");
  }
  @Override
  public void actionPerformed(ActionEvent e){
   System.out.println("Delete command is invoked...");
  }
}
```

The Strategy Pattern

Motivation

- A variety of algorithms exists to perform a particular operation
- The client needs to be able to select/change the choice of algorithm *at run-time.*

The Strategy Pattern (cont.)

Examples where different *strategies* might be used:

- Save a file in different formats (plain text, PDF, PostScript…)
- Compress a file using different compression algorithms
- Sort data using different sorting algorithms
- Capture video data using different encoding algorithms
- Plot the same data in different forms (bar graph, table, …)
- Have a game's non-player character (NPC) change its AI
- Arrange components in an on-screen window using different layout algorithms

Example: NPC AI Algorithms

Typical client code sequence:

```
void attack() {

        switch (characterType) {
        case WARRIOR:   fight();                break;
        case HUNTER:    fireWeapon();           break;
        case PRIEST:    castDisablingSpell();   break;
        case SHAMAN:    castMagicSpell();       break;
        }
}
```

Problem with this approach?

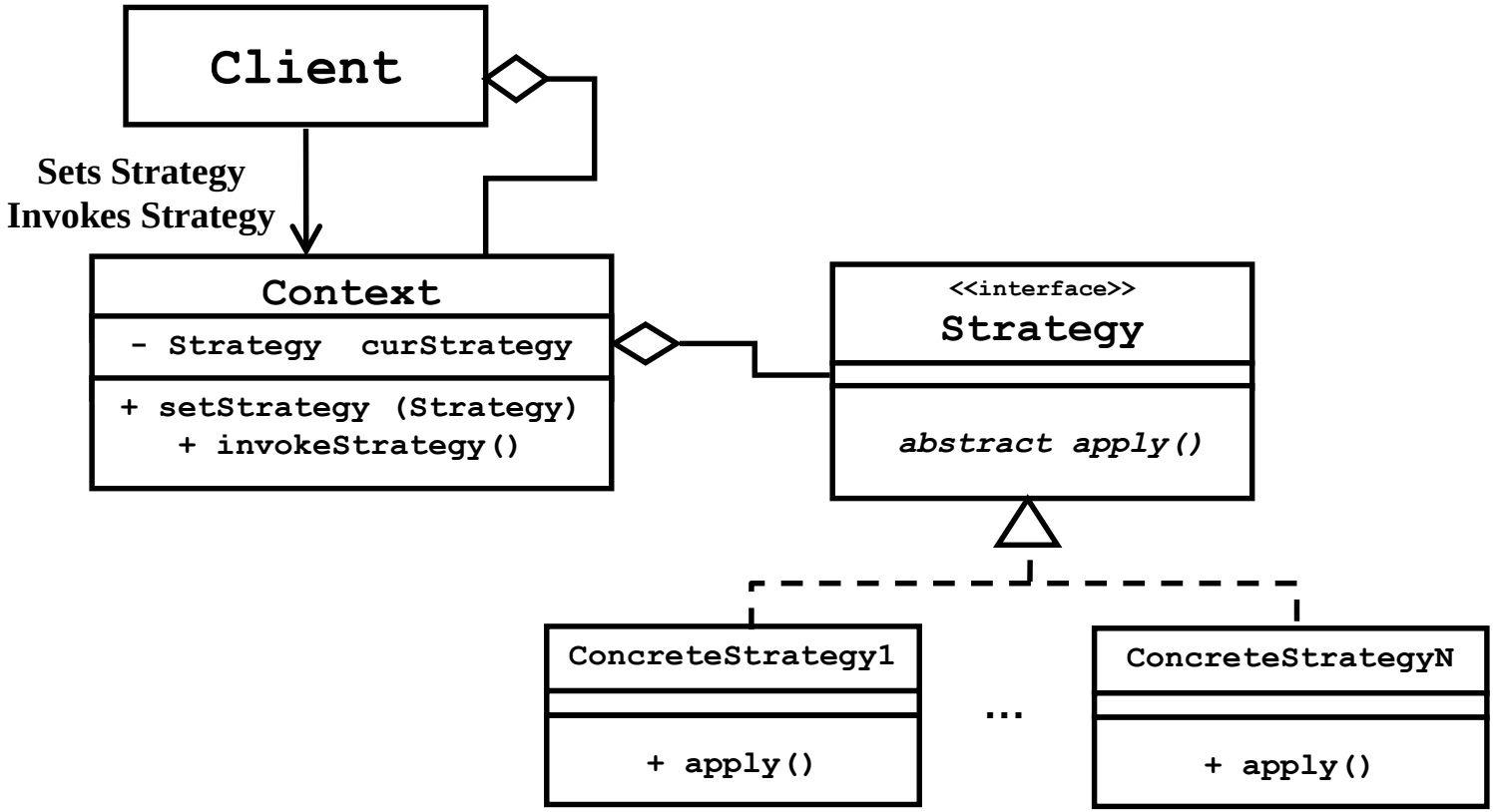
Changing or adding a plan requires changing the client!

Solution Approach

- Provide various objects that know how to "apply strategy" (e.g. apply fight, fireWeapon, or castMagicSpell strategies)
  - Each in a different way, but with a uniform interface
- The context (e.g. NPC) maintains a "current strategy" object
- Provide a mechanism for the client (e.g. Game) to *change* and *invoke* the current strategy object of a context

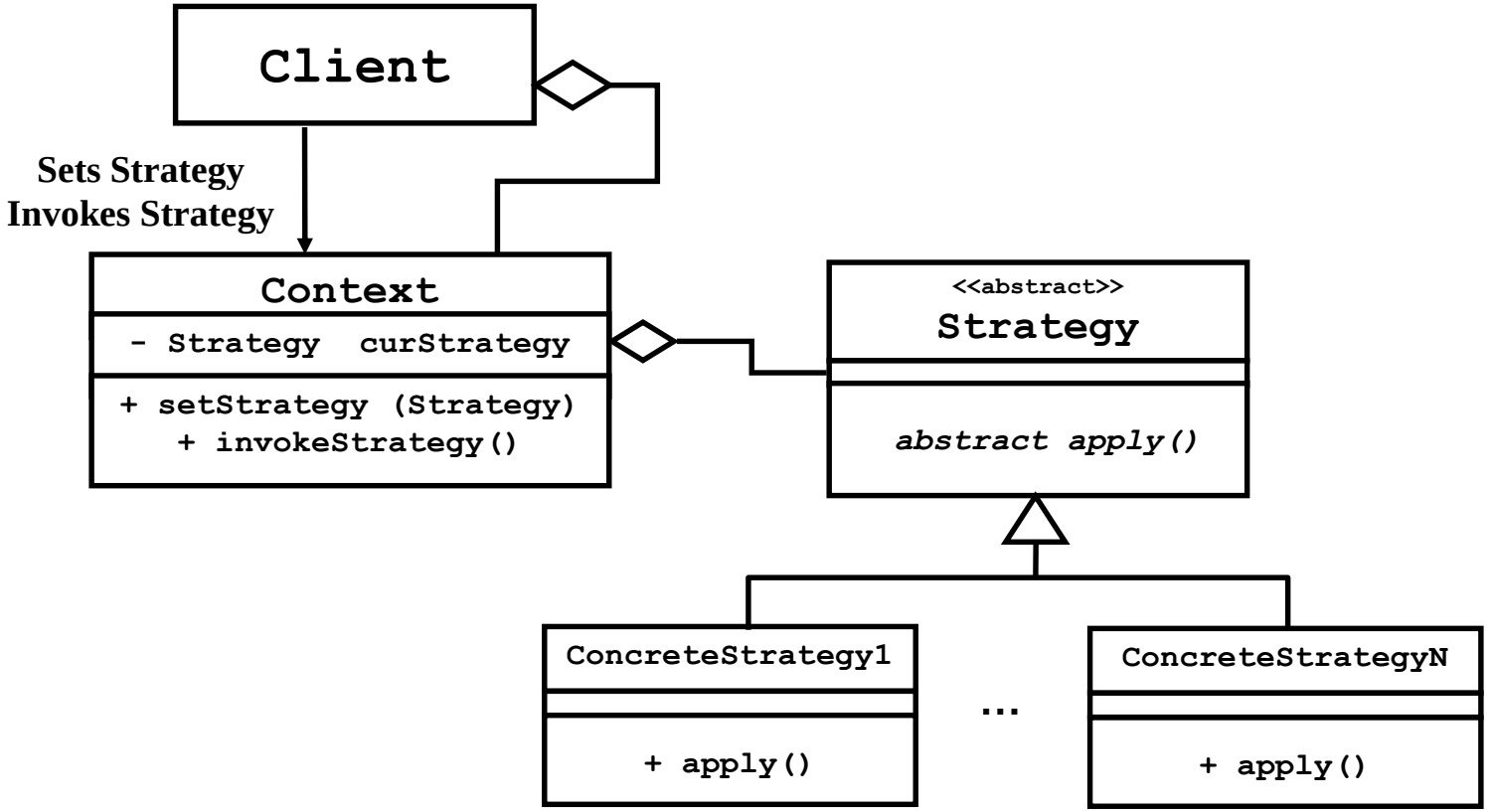
Strategy Pattern Organization

- **Using Interfaces**



Strategy Pattern Organization (cont.)

- **Using subclassing**



Example: NPC's in a Game

```
public interface Strategy {
    public void apply();
}

public class FightStrategy implements Strategy {
    public void apply() {
        //code here to do "fighting"
    }
}

public class FireWeaponStrategy implements Strategy {
    private Hunter hunter;
    public FireWeaponStrategy(Hunter h) {
        this.hunter = h;   //record the hunter to which this strategy applies
    }
    public void apply() {
        //tell the hunter to fire a burst of 10 shots
        for (int i=0; i<10; i++) {
            hunter.fireWeapon();
        }
    }
}

public class CastMagicSpellStrategy implements Strategy {
    public void apply() {
        //code here to cast a magic spell
    }
}
```

NPC's in a Game (cont.)

"Contexts" :

```
public class Character {
    private Strategy curStrategy;
    public void setStrategy(Strategy s) {
        curStrategy = s;
    }
    public void invokeStrategy() {
        curStrategy.apply();
    }
}
```

```
public class Warrior extends Character {

//code here for Warrior specific methods

}
```

```
public class Shaman extends Character {

//code here for Shaman specific methods

}
```

```
public class Hunter extends Character {
    private int bulletCount ;

    public boolean isOutOfAmmo() {
        if (bulletCount <= 0) return true;
        else return false;
    }
    public void fireWeapon() {
        bulletCount -- ;
    }

//code here for other Hunter specific
//methods

}
```

Assigning / Changing Strategies

```
/** This Game class demonstrates the use of the Strategy Design Pattern
 *  by assigning attack response strategies to each of several game characters.
 */
public class Game {
    //the list of non-player characters in the game
    ArrayList<Character> npcList = new ArrayList<Character>();
    public Game() {    //construct some characters, assigning each a starting strategy
        Warrior w1 = new Warrior();
        w1.setStrategy(new FightStrategy());
        npcList.add(w1);
        Hunter h1 = new Hunter();
        h1.setStrategy(new FireWeaponStrategy(h1));
        npcList.add(h1);
        Shaman s1 = new Shaman();
        s1.setStrategy(new CastSpellStrategy());
        npcList.add(s1);
    }
    public void attack() {    //force each character to execute its attack response
        for (Character c : npcList) {
            c.invokeStrategy();
        }
    }
    public void updateCharacters() {  //update any strategies that need changing
        for (Character c : npcList) {
            if(c instanceof Hunter) {
                if ( ((Hunter)c).isOutOfAmmo() ) {
                    //change the character's strategy
                    c.setStrategy(new FightStrategy());
                }
            }
        }
    }
}
```

CN1 Layouts

- Strategy abstract super class:

  `Layout`

- Client is the **`Form`**
- Context: **`Container`** (e.g., **`ContentPane`** of **`Form`**)
- Context methods:

```
public void setLayout (Layout lout)
public void revalidate()
```

- Concrete strategies (**`extends Layout`**):

```
class FlowLayout()
class BorderLayout()
class GridLayout()
...
```

- "Apply" method (declared in the **`Layout`** super class):

  abstract **`void layoutContainer(Container parent)`**

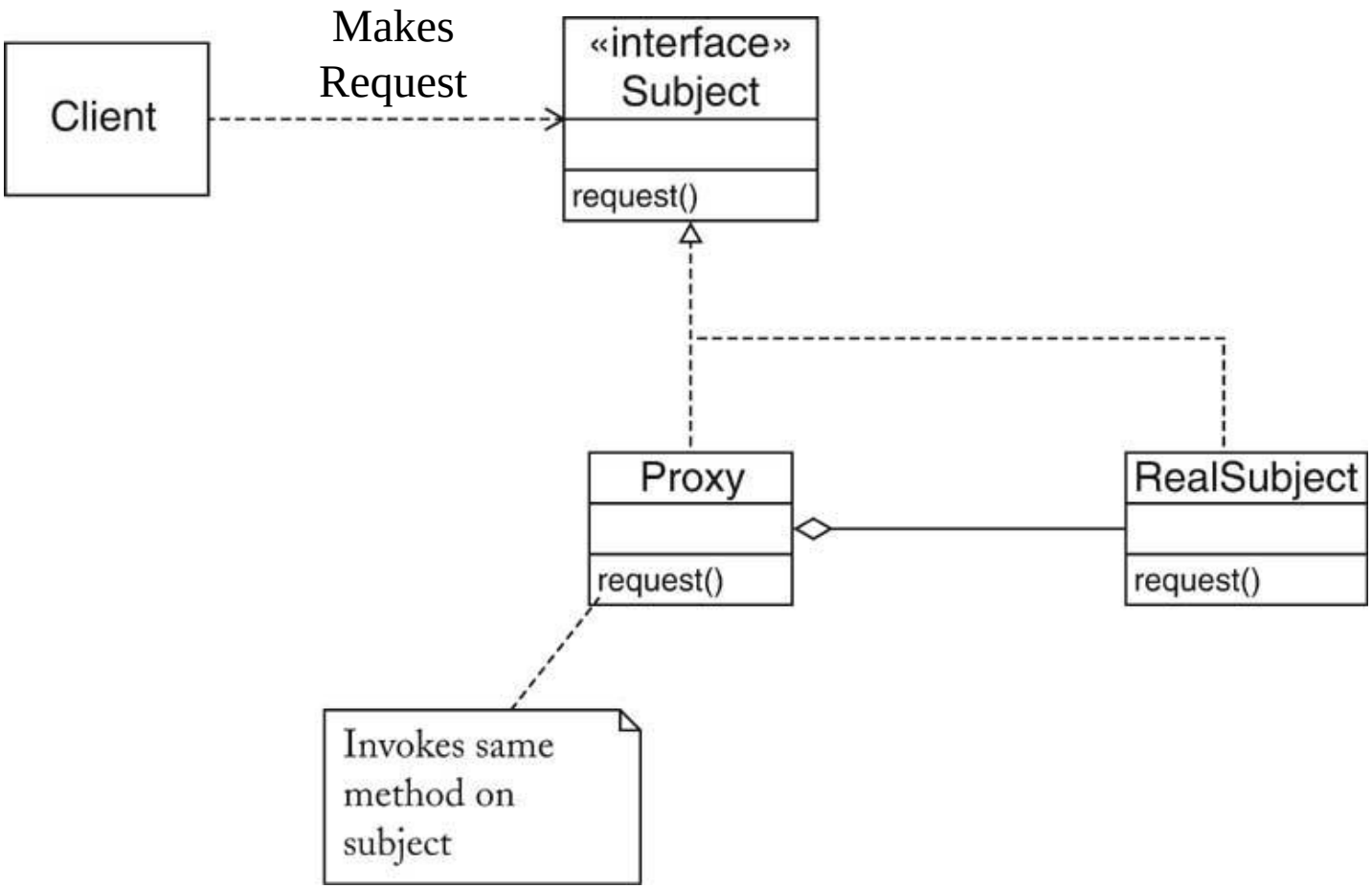
The Proxy Pattern

- Motivation
  - Undesirable target object manipulation
    - Access required, but not to all operations
  - Expensive target object manipulation
    - Lengthy image load time
    - Significant object creation time
    - Large object size
  - Inaccessible target object
    - Resides in a different address space
      - E.g. another JVM or a machine on a network

Proxy Types

- **Protection Proxy – controls access**
- **Virtual Proxy – acts as a stand-in**
- **Remote Proxy – local stand-in for object in another address space**

Proxy Pattern Organization



Proxy Example

```
interface IGameWorld {
     Iterator getIterator();
     void addGameObject(GameObject o);
     boolean removeGameObject (GameObject o);
}


/**A proxy which prohibits removal of GameWorldObjects from the GameWorld*/
public class GameWorldProxy implements IObservable, IGameWorld {
     private GameWorld realGameWorld ;
     public GameWorldProxy (GameWorld gw)
       { realGameWorld = gw; }
     public Iterator getIterator ()
       { return realGameWorld.getIterator(); }
     public void addGameObject(GameObject o)
       {  realGameWorld.addGameObject(o) ;  }
     public boolean removeGameObject (GameObject o)
       {  return false ; }
     //...[also has methods implementing IObservable]
}
```

Proxy Example (cont.)

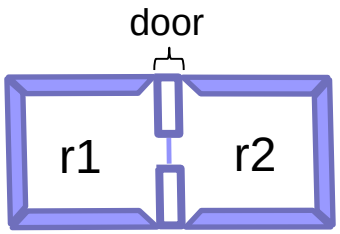
```
/** This class defines a Game containing a GameWorld with a ScoreView Observer. */
public class Game {
    public Game() {
        GameWorld gw = new GameWorld();   //construct a GameWorld
        ScoreView sv = new ScoreView();   //construct a ScoreView
        gw.addObserver(sv);               //register ScoreView as a GameWorld Observer
    }
}
-------------------------------------------------------------------------------
/** This class defines a GameWorld which is an Observable and maintains a list of
 *  Observers; when the GameWorld needs to notify its Observers of changes it does so
 *  by passing a GameWorldProxy to the Observers. */
public class GameWorld implements IObservable, IGameWorld {
    private Vector<GameObject> myGameObjectList = new Vector<GameObject>();
    private Vector<IObserver> myObserverList = new Vector<IObserver>();
    public Iterator<GameObject> getIterator() { ... }
    public void addGameObject(GameObject o) { ... }
    public boolean removeGameObject(GameObject o) {
        //code here to remove the specified GameObject from the GameWorld...
    }
    public void addObserver(IObserver o) { myObserverList.add(o); }

    //Pass a GameWorldProxy to Observers, thus prohibiting Observer removal of GameObjects
    public void notifyObservers() {
        GameWorldProxy proxy = new GameWorldProxy(this);
        for (IObserver o : myObserverList) {
            o.update((IObservable)proxy, null);
        }
    }
}
```

The Factory Method Pattern

- **Motivation**
  - **Sometimes a class can't anticipate the class of objects it must create**
  - **It is sometimes better to delegate specification of object types to subclasses**
  - **It is frequently desirable to avoid binding application-specific classes into a set of code**

Example: Maze Game

```
public class MazeGame {

    // This method creates a maze for the game, using a hard-coded structure for the
    // maze (specifically, it constructs a maze with two rooms connected by a door).
    public Maze createMaze () {

        Maze theMaze = new Maze() ;   //construct an (empty) maze

        Room r1 = new Room(1) ;       //construct components for the maze
        Room r2 = new Room(2) ;
        Door theDoor = new Door(r1, r2);

        r1.setSide(NORTH, new Wall()); //set wall properties for the rooms
        r1.setSide(EAST,  theDoor);
        r1.setSide(SOUTH, new Wall());
        r1.setSide(WEST,  new Wall());

        r2.setSide(NORTH, new Wall());
        r2.setSide(EAST,  new Wall());
        r2.setSide(SOUTH, new Wall());
        r2.setSide(WEST,  theDoor);

        theMaze.addRoom(r1); //add the rooms to the maze
        theMaze.addRoom(r2);

        return theMaze ;
    }

    //other MazeGame methods here (e.g. a main program which calls createMaze())...
}
```



Problems with `createMaze()`

- Inflexibility; lack of "reusability"
- Reason: it "hardcodes" the maze types
  - Suppose we want to create a maze with (e.g.)
    - Magic Doors
    - Enchanted Rooms
  - Possible solutions:
    - Subclass MazeGame and override **`createMaze()`** (i.e., create a whole new version with new types)
    - Hack **`createMaze()`** apart, changing pieces as needed

createMaze() Factory Methods

```
public class MazeGame {

    //factory methods - each returns a MazeComponent of a given type
    public Maze makeMaze()         { return new Maze() ; }
    public Room makeRoom(int id)   { return new Room(id) ; }
    public Wall makeWall()         { return new Wall() ; }
    public Door makeDoor(Room r1, Room r2) { return new Door(r1,r2) ; }

    // Create a maze for the game using factory methods
    public Maze createMaze () {
        Maze theMaze = makeMaze() ;
        Room r1 = makeRoom(1) ;
        Room r2 = makeRoom(2) ;
        Door theDoor = makeDoor(r1, r2);
        r1.setSide(NORTH, makeWall());
        r1.setSide(EAST,  theDoor);
        r1.setSide(SOUTH, makeWall());
        r1.setSide(WEST,  makeWall());
        r2.setSide(NORTH, makeWall());
        r2.setSide(EAST,  makeWall());
        r2.setSide(SOUTH, makeWall());
        r2.setSide(WEST,  theDoor);
        theMaze.addRoom(r1);
        theMaze.addRoom(r2);
        return theMaze ;
    }
    ...
}
```

Overriding Factory Methods

```
//This class shows how to implement a maze made of different types of rooms.  Note
// in particular that we can call exactly the same (inherited) createMaze() method
// to obtain a new "EnchantedMaze".
public class EnchantedMazeGame extends MazeGame {

    //override MakeRoom to produce "EnchantedRooms"
    @Override
    public Room makeRoom(int id) {

        //create the spell necessary to enter the enchanted room
        Spell spell = makeSpell() ;

        //construct and return an EnchantedRoom requiring a spell to be entered
        return new EnchantedRoom(id, spell);
    }

    //override MakeDoor to produce a door requiring a spell
    @Override
    public Door makeDoor(Room r1, Room r2) {

        //construct and return a Door requiring a spell to be entered
        return new DoorNeedingSpell(r1, r2);
    }

    //new factory method for making spells
    public Spell makeSpell() { return new Spell() ;}
    ...
}
```