CSC 133

Object-Oriented Computer Graphics Programming

Design Patterns I

Dr. Kin Chung Kwan

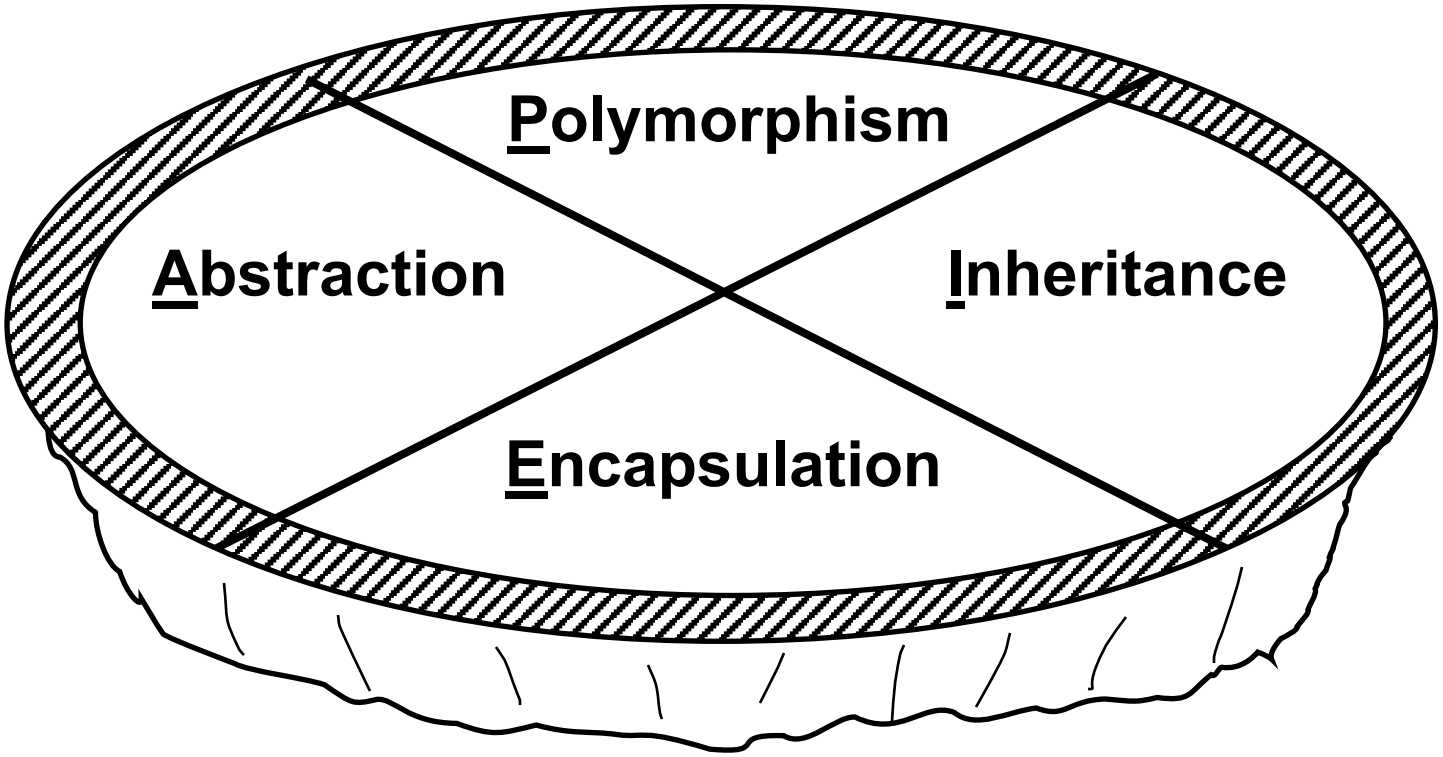
Spring 2023

Computer Science Department
California State University, Sacramento

SACRAMENTO STATE

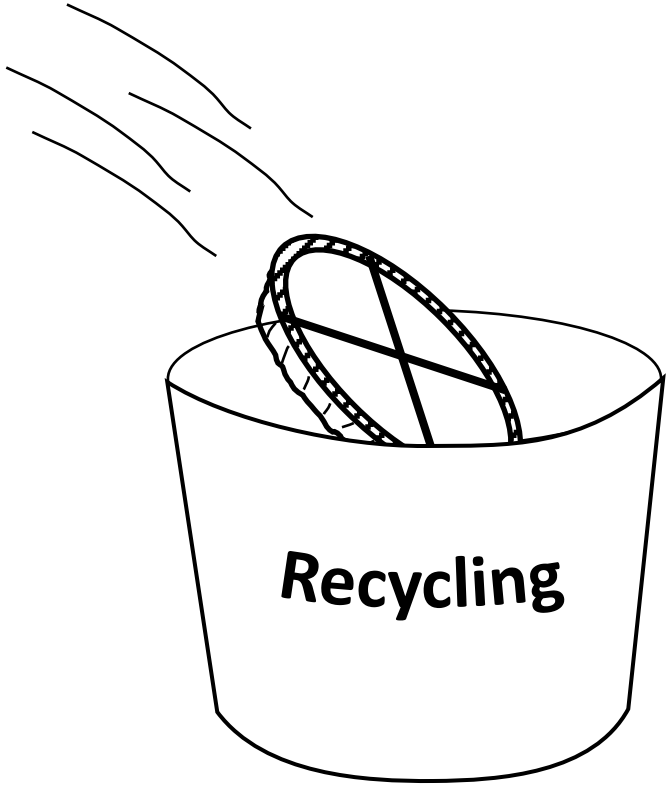
“A Pie”

Four distinct OOP Concepts (or Pillars)



Don’t Worry

- Not this time



Design Patterns

Design Patterns

Definition:

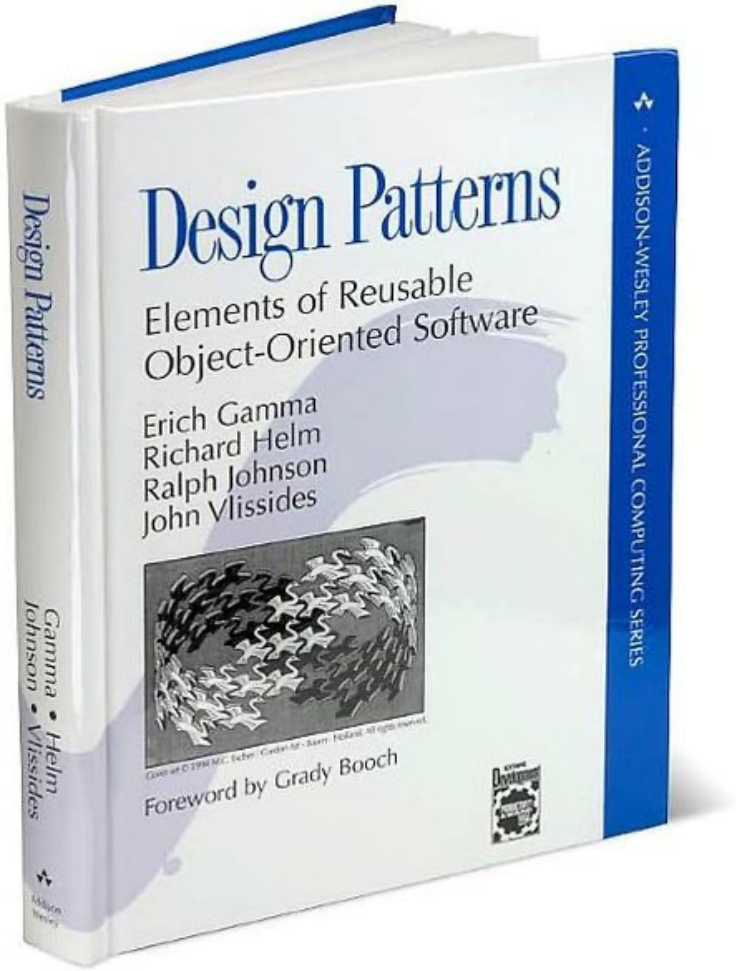
- A generic solution to common problems.

Popularized by

"Design Patterns: Elements of Reusable Object-Oriented Software"

by Gamma et al.

In 1995



Gang of Four

The authors of this book

- Erich Gamma
- Richard Helm
- Ralph Johnson
- John Vlissides

Ralph, Erich, Richard and John

23 Patterns

Identified the original set of 23 patterns

- Common practice
- Code frequently needs to do things that have been done before

Categorize them into three types

- Creational
- Structural
- Behavioral

Types of Design Patterns

- **Creational**
  - Deal with process of object creation
- **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
- **Behavioral**
  - Deal with interaction between objects
  - Encapsulate processes performed by objects

Common Design Patterns

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

Common Design Patterns

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

Iteration

- Repeating the process multiple times

- Loop in Java
  - For loop

```
for (int i = 0; I < 10; i++) {...}
```

  - While loop

```
while ( condition ) {...}
```

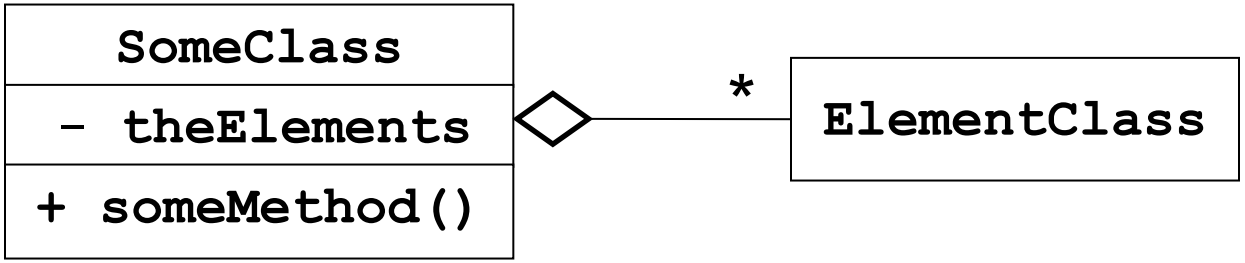
  - Do-while loop

```
do { ... } while (condition);
```

The Iterator Pattern

Motivation:

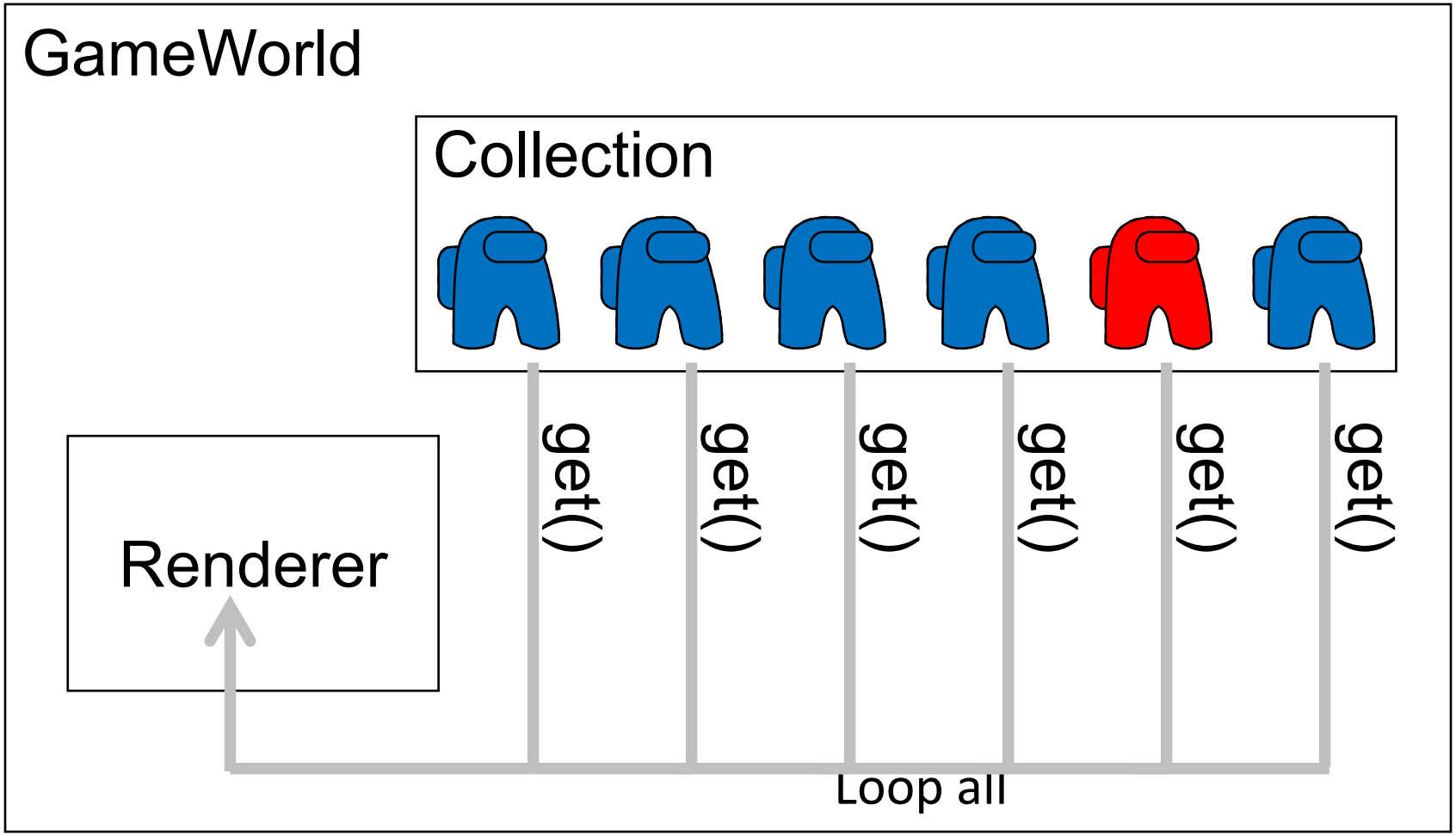
- An “aggregate” object contains “elements”
- “Clients” need to access these elements
- Aggregate shouldn’t expose internal structure



Case Example

- A Game has a lot of game characters
- Want to display the characters
- Without knowing the how character stored in a GameWorld

Solution


GameWorld
Collection
Renderer
get()
get()
get()
get()
get()
get()
Loop all

Collection Class

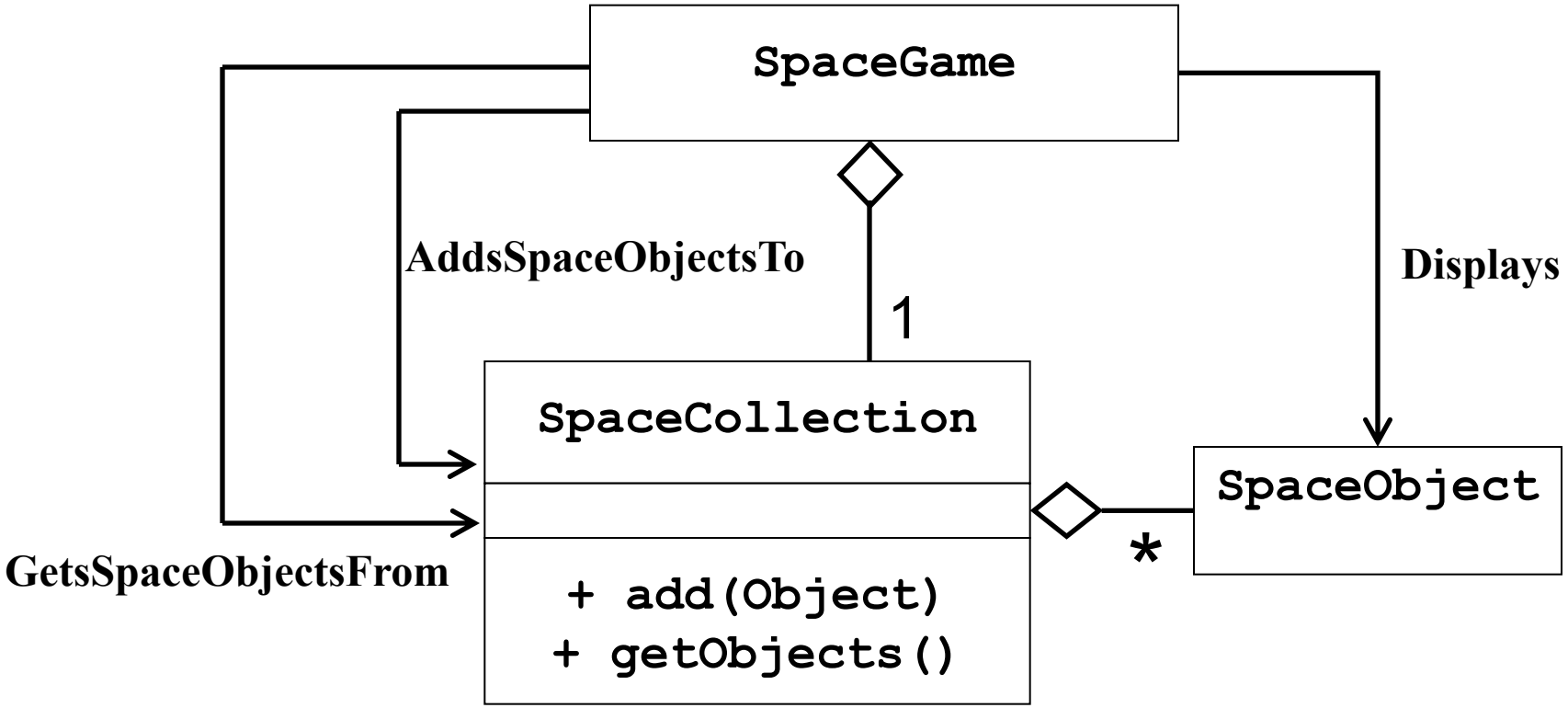
- A class that can store most elements with
  - Multiple variables
  - Array/Vector/Table/List/etc

- Provide a get function to get the elements

```
collection.getElement();
```

- Usually with polymorphism
  - E.g., `GameObject` to store character/enemy/items

Collection Classes



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();
     }
}
```

SpaceGame

```
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));
       }
  }
}
```

SpaceCollection with Vector

```
/**  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 with Hashtable

```
/**  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 ;
    }
}
```

Implementation

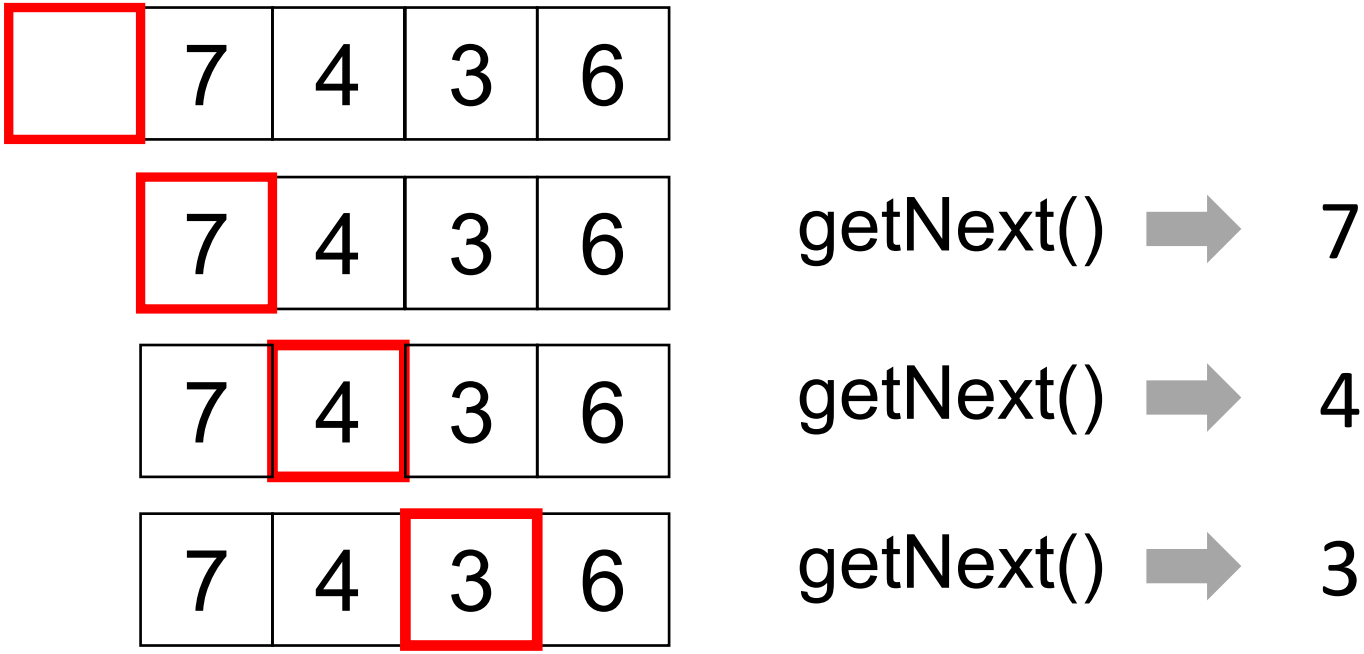
- Use different storage type to achieve different goals.
  - Sorted
  - Short access time
  - Random access
  - Memory consumption
  - Etc.

- No matter which
  - Able to return the element one by one

Iterator

One common approach is to use “iterator.”

- A class that can `getNext();`
- Move one step and then return



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 ( ) ;
     }
     ...continue...
```

SpaceCollection With Iterator

```
  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
```

Inner Class

```
public class SpaceCollection {
  ...
  private class SpaceVectorIterator {
    ...
  }
}
```

A class inside another class

- Private class only

UML Notation

For an inner class relationship

Using An Iterator

```
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 ) ;
          }
     }
}
```

CN1's Iterator Interface

`boolean hasNext()`

Returns true if the collection has more elements.

`Object next()`

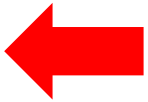
Returns the next element in the collection.

`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.

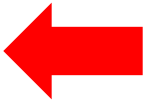
boolean **isEmpty**() : Returns true if this collection contains no elements.

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

- CN1 Collection interface is a subinterface of CN1 Iterable interface.
- Implementing this interface allows an object to be the target of the "***foreach***" statement…

```
interface Iterable {
   public Iterator iterator();
}
```

Foreach loop

```
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());
      }
   }
}
```

Iterators In C++

- C++ Standard Template Library (STL) provides container classes (e.g., vector, map, list, …)
- All containers provide iterators over their contents, using functions returning pointers:

```
using namespace std;
vector<SpaceObject> myObjs;      //create a container
  //... code here to add SpaceObjects to the container (vector)
vector<SpaceObject>::iterator itr;      //declare an iterator
for (itr = myObjs.begin(); itr != myObjs.end(); itr++) {
   cout << *itr ;   //output next obj pointed to by itr
}
```

The Composite

The Composite Pattern

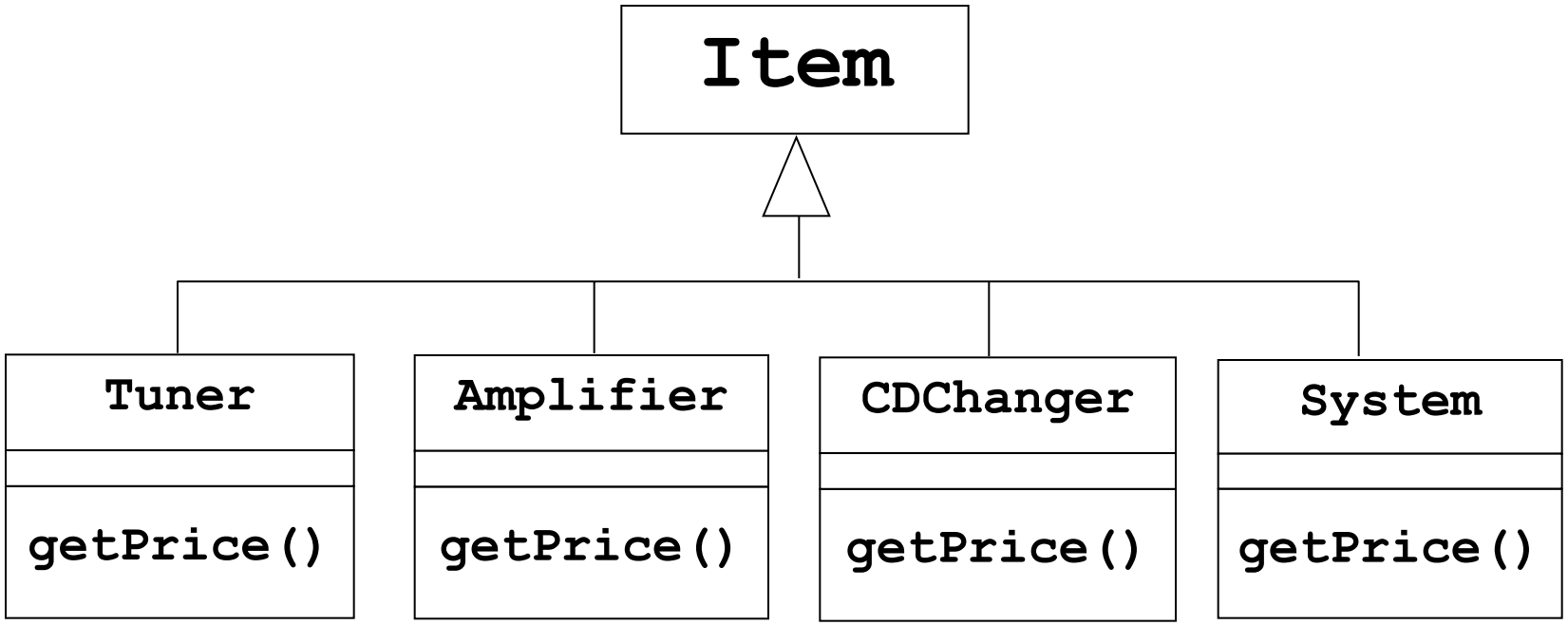
- **Motivation:**
  - Objects organized in a hierarchical manner
  - Some objects are *groups* of the other objects
  - Individuals and groups need to be treated uniformly

Problems Case

- A store sells stereo component items:
  - Tuners, Amplifiers, CDChangers, etc.
  - Each item has a getPrice() method

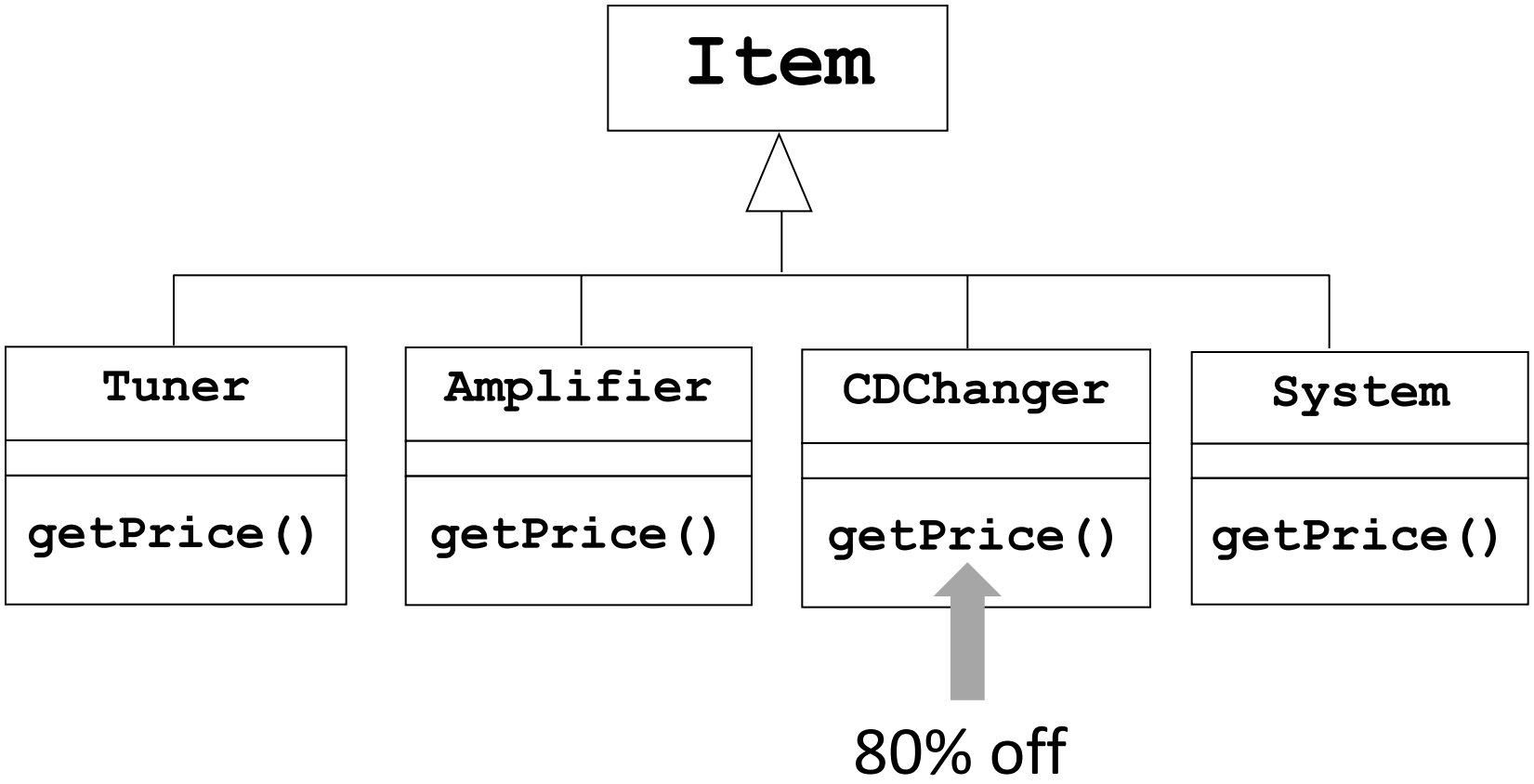
- The store also sells complete stereo systems
  - Systems also have a getPrice() method
  - with a discounted sum of the prices.

Possible Class Organization



Problem ?

Imagine that



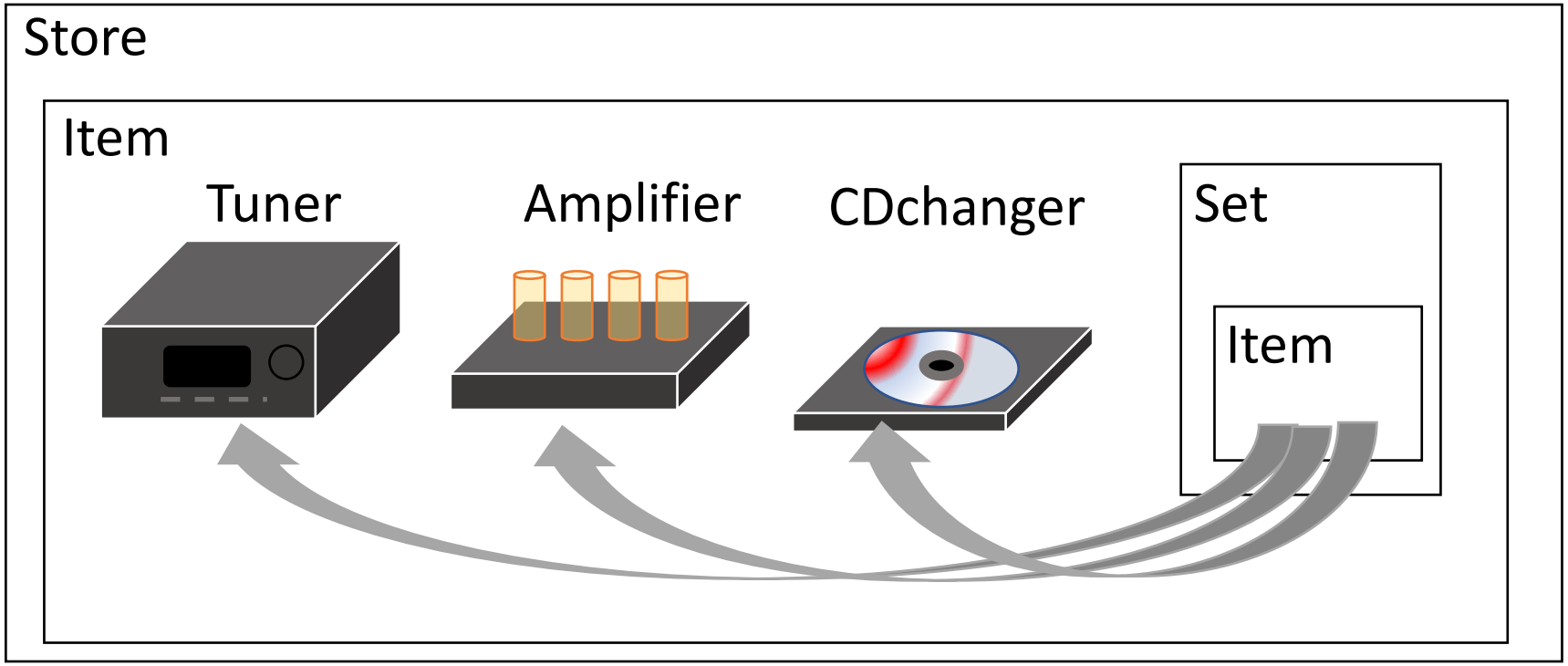
Problem

- If you change the price of other items
  - The combined one does not change.

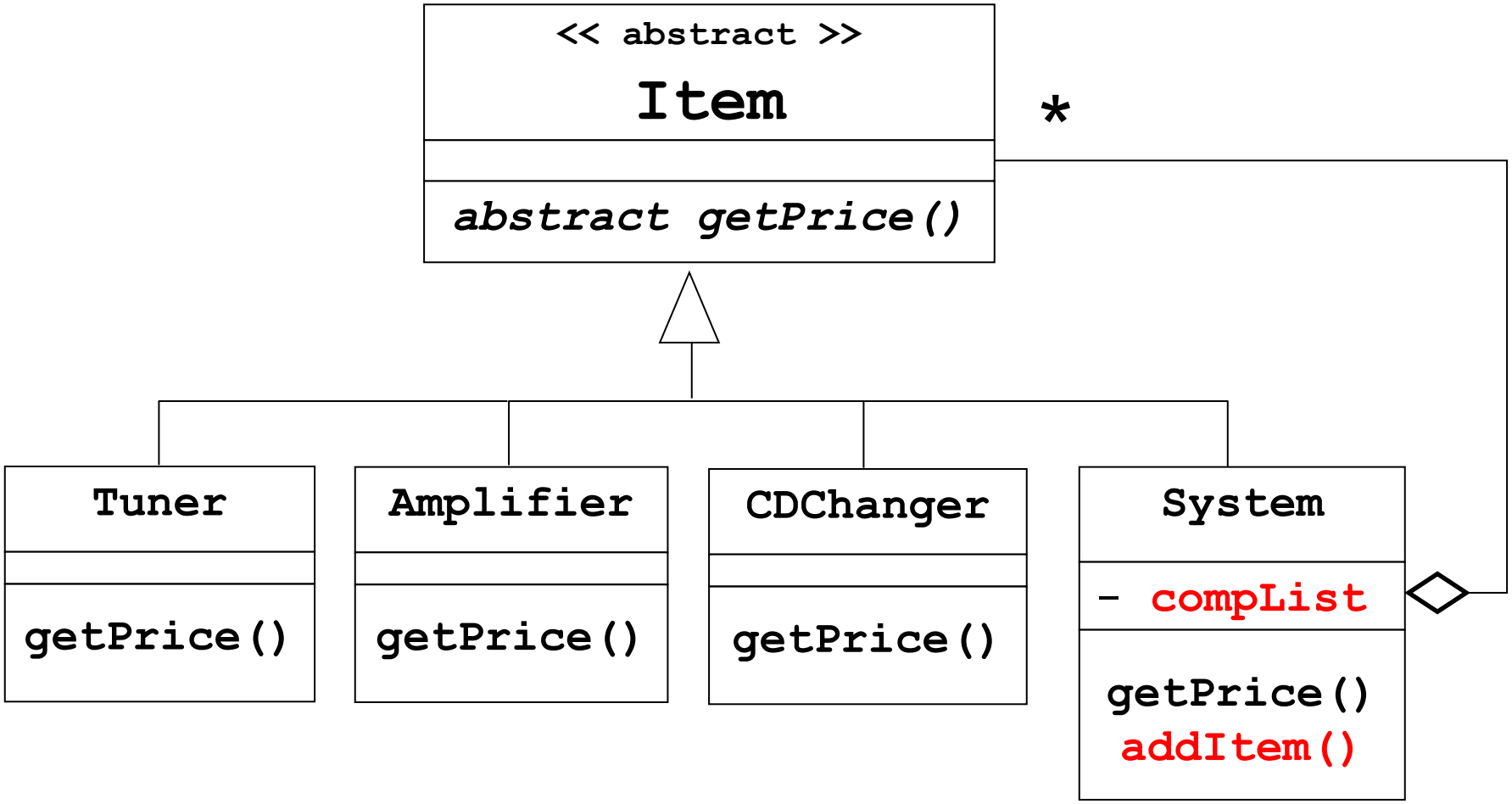
- We want to calculate the combined one
  - Automatically

Solution

- An item can contain other items

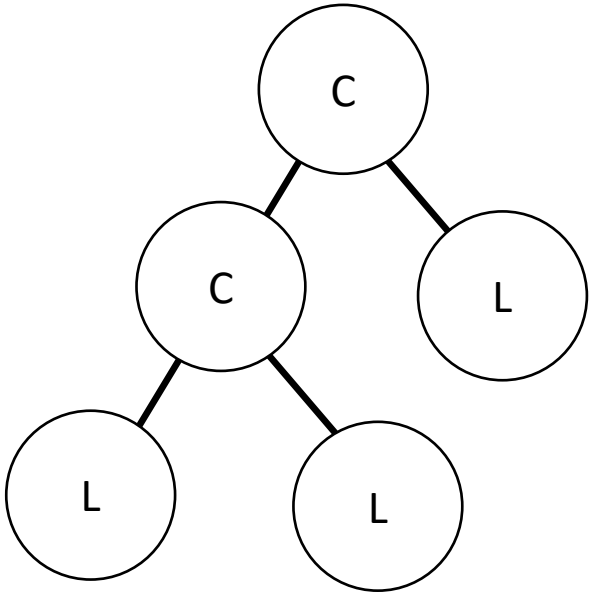


UML

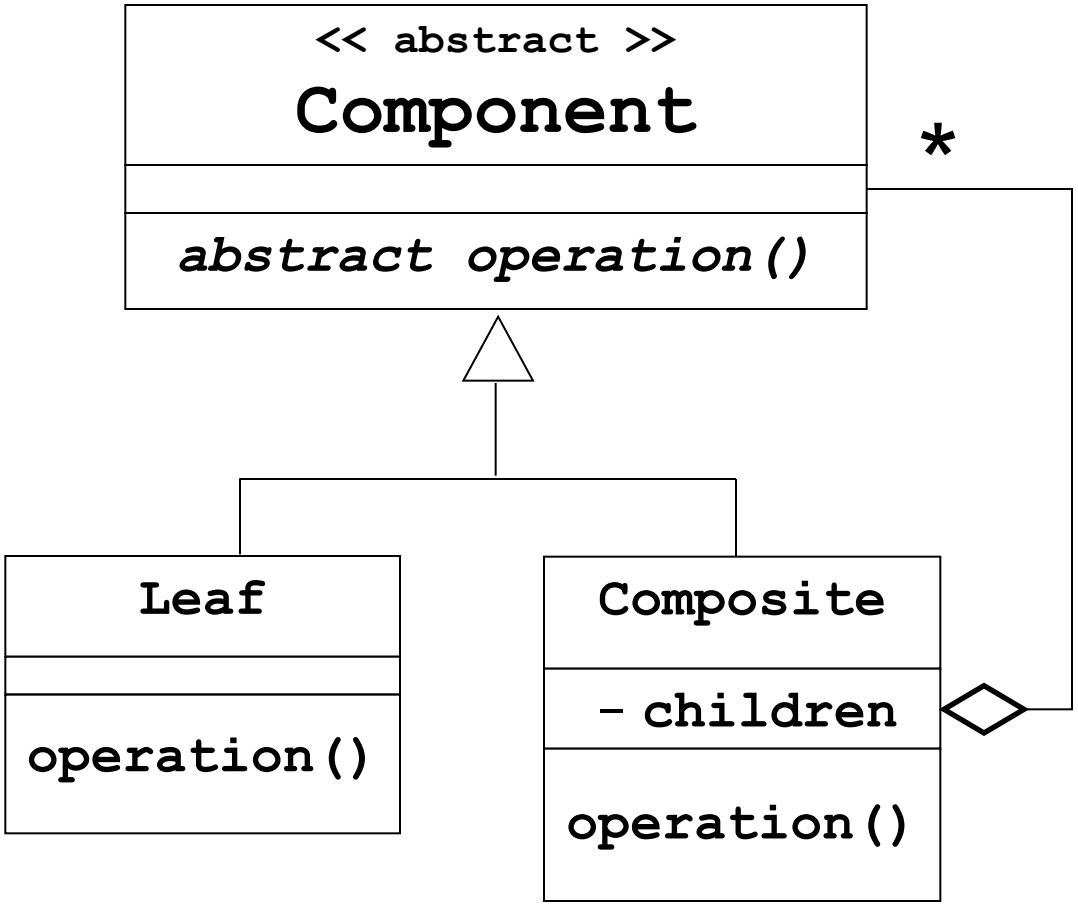

<< abstract >>
Item
abstract getPrice()
*
Tuner
getPrice()
Amplifier
getPrice()
CDChanger
getPrice()
System
- compList
getPrice()
addItem()

Composite

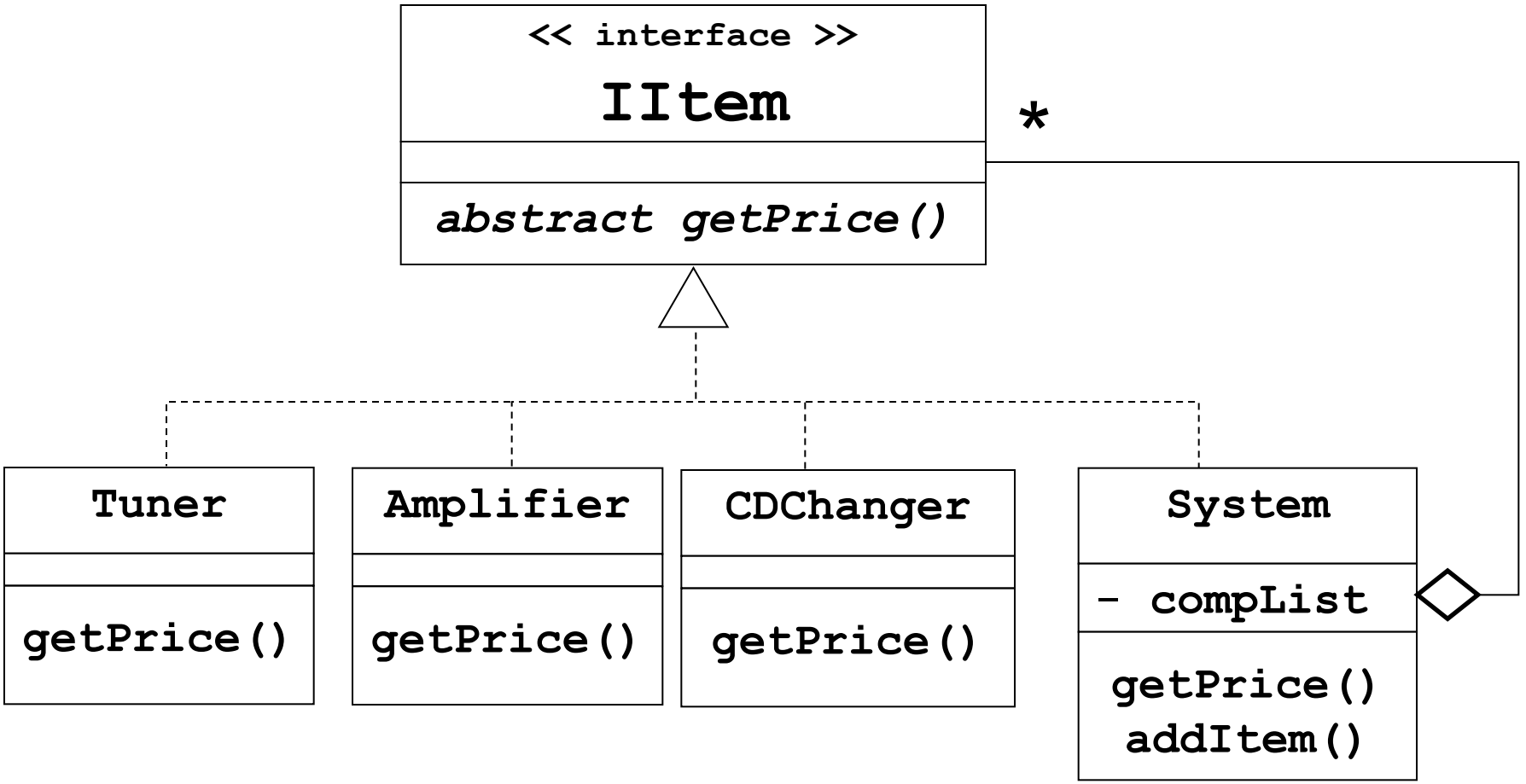
- **Component** :
  - Classes to maintain/access the child groups
- **Leaf**
  - The primitive objects.



Composite Pattern Organization

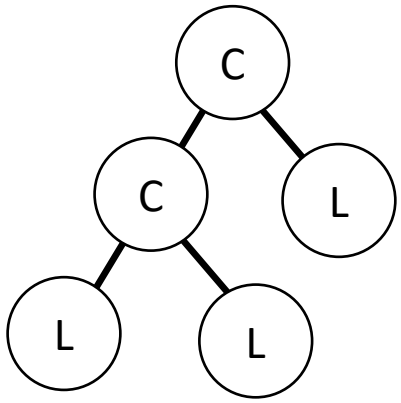


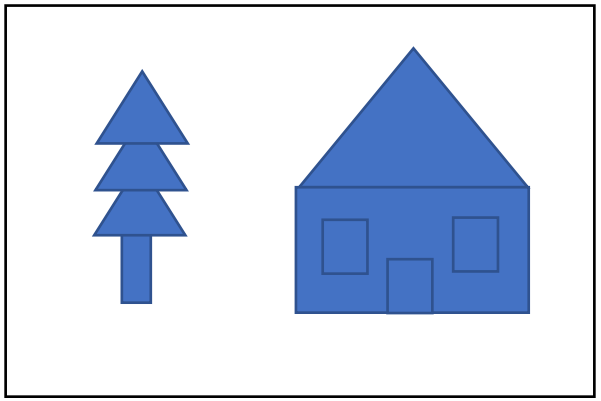
Composite Specified With Interfaces



Examples Of Composites

- Trees
  - Internal nodes (groups) and leaves
- Arithmetic expressions
  - `<exp> ::= <term> | <term> “+” <exp>`
- Graphical Objects
  - Shape is formed by shapes





The Singleton

The Singleton Pattern

- **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**
  - The player character in a single game

Idea of Singleton

- A global class to store one objects

- Provide one static `get()` method
  - To get the object

- If the object is not found
  - Create the objects

Singleton Implementation

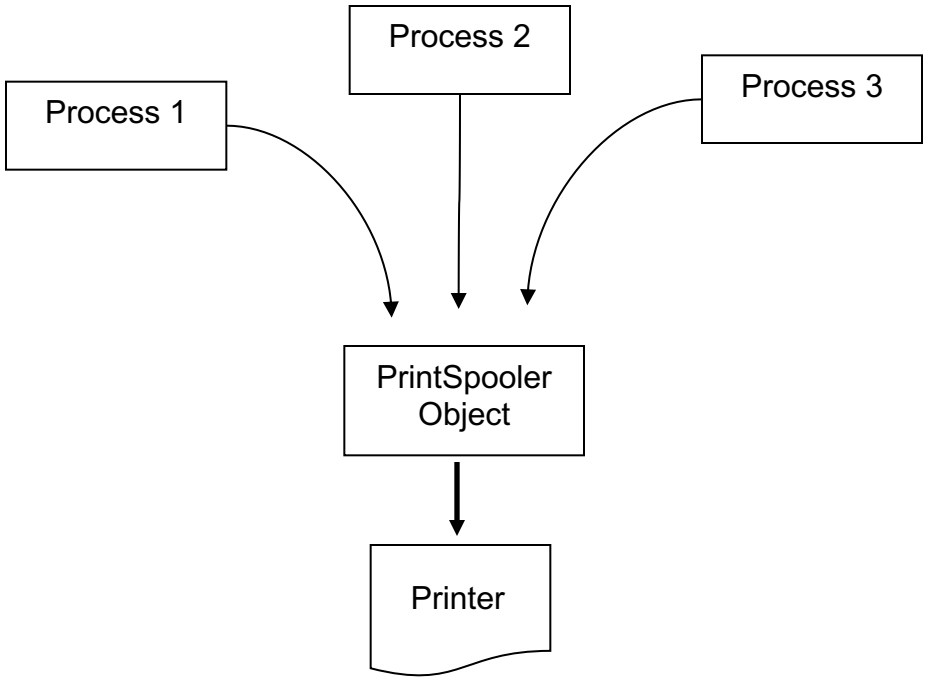
```
public class PrintSpooler {

     // 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;
     }

}
```

To get the reference:

- **`PrintSpooler.getSpooler()`**

Usage Example

Multiple processes should not access a single printer simultaneously

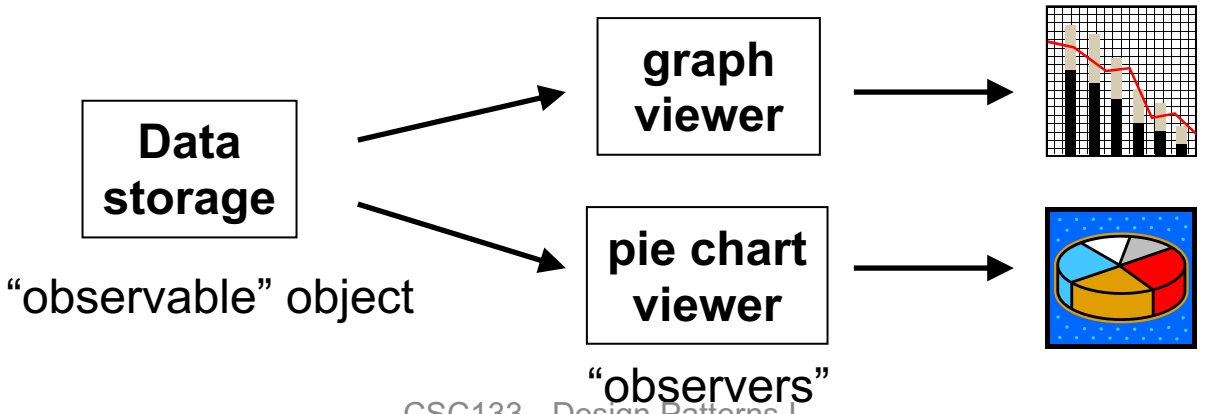


The Observer

The Observer Pattern

Motivation

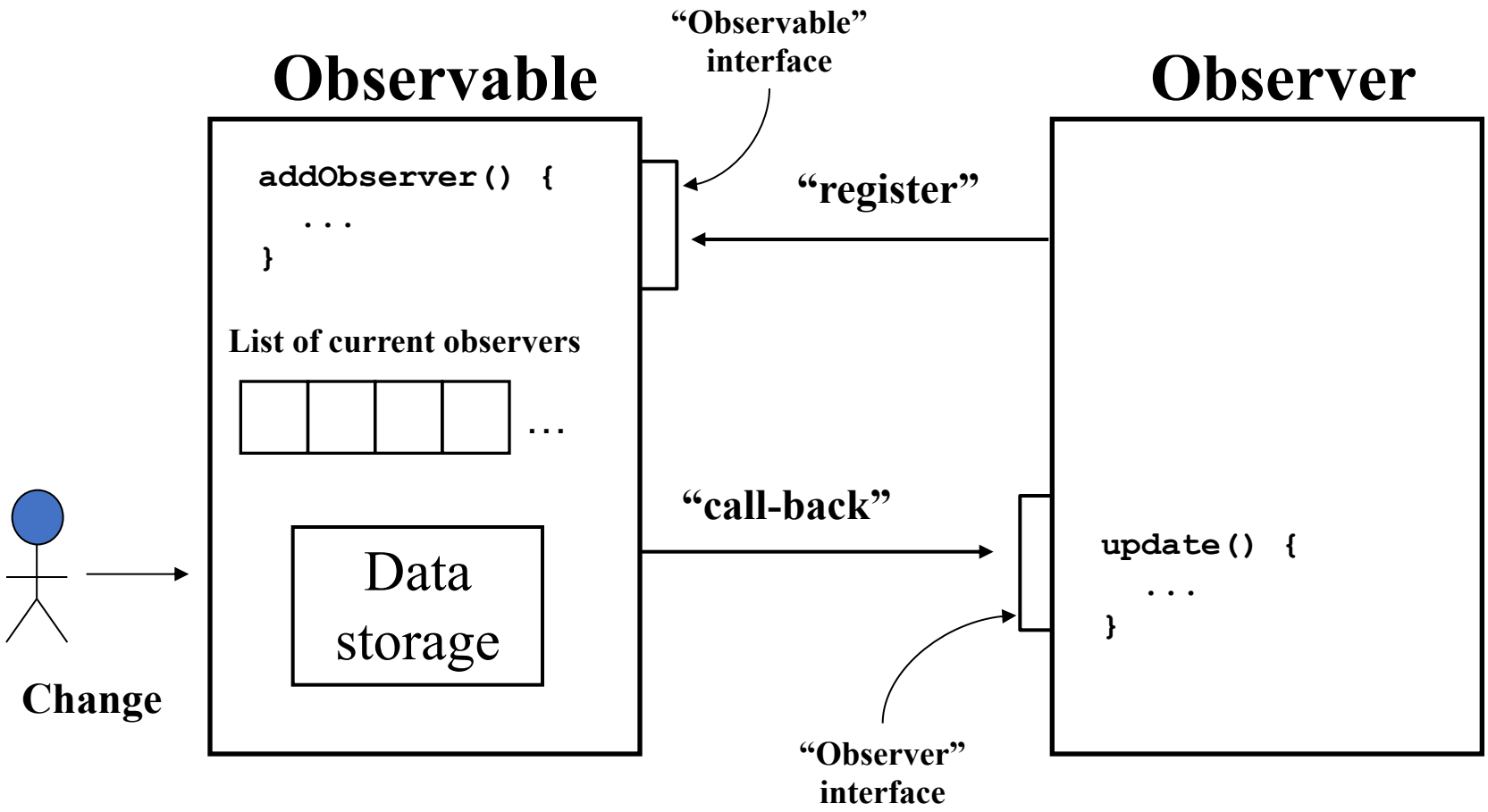
- An object stores data that changes regularly
- Various clients use the data in different ways
- Clients need to know when the data changes
- No need to change the data storage object when new clients are added



Idea of `Observer`

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

Idea of Observer



Observer/Observable

```
public interface Observer { //built-in 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 { //built-in class
  public void addObserver (Observer obs) {...}
  public void notifyObservers() {...}
  protected void setChanged() {...}
  ...
}
```

Observable Class

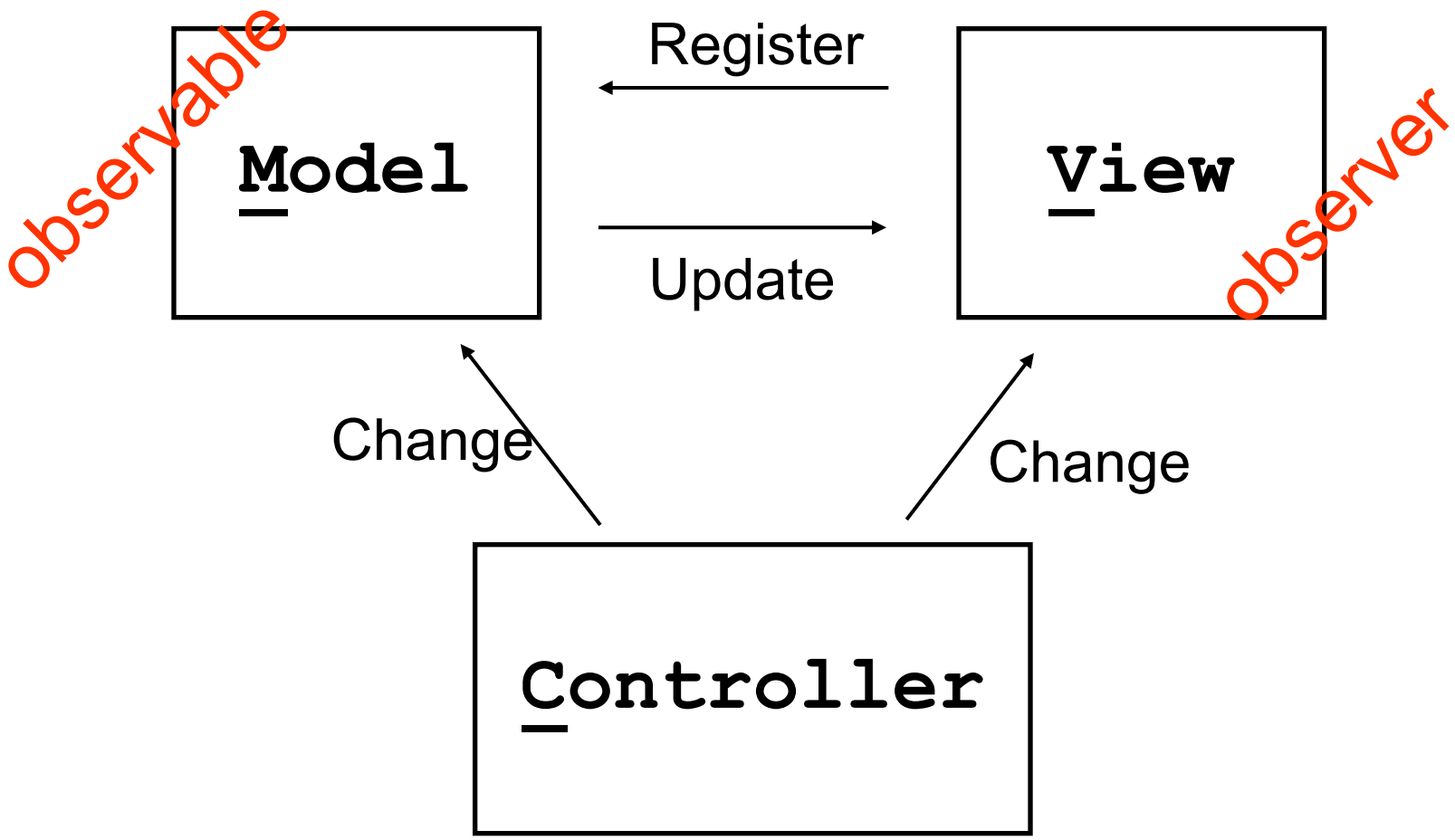
Extending from a built-in `Observable` class:

- 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()`

MVC Architecture

- Common architecture in different application
  - Can have different structure
- **Controller**
  - Input
- **Model**
  - Data storage
- **View**
  - Output

MVC Architecture Idea



```
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
  }
}

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
  }
}
```

The Command

The Command Pattern

Motivation

- Avoid duplicate code that performs the same operation invoked from different sources
- Separate a command from the object
- Maintaining state information about the command

Problems

- Copy & Paste in notepad
  - Ctrl+C & Ctrl+V
  - Edit > Copy & Edit > Paste
  - Select by mouse > Right click > Copy > …
- Implement it three times?
  - We can just send a command copy/paste to the system
    - `notepad.copy();`
    - Encapsulation

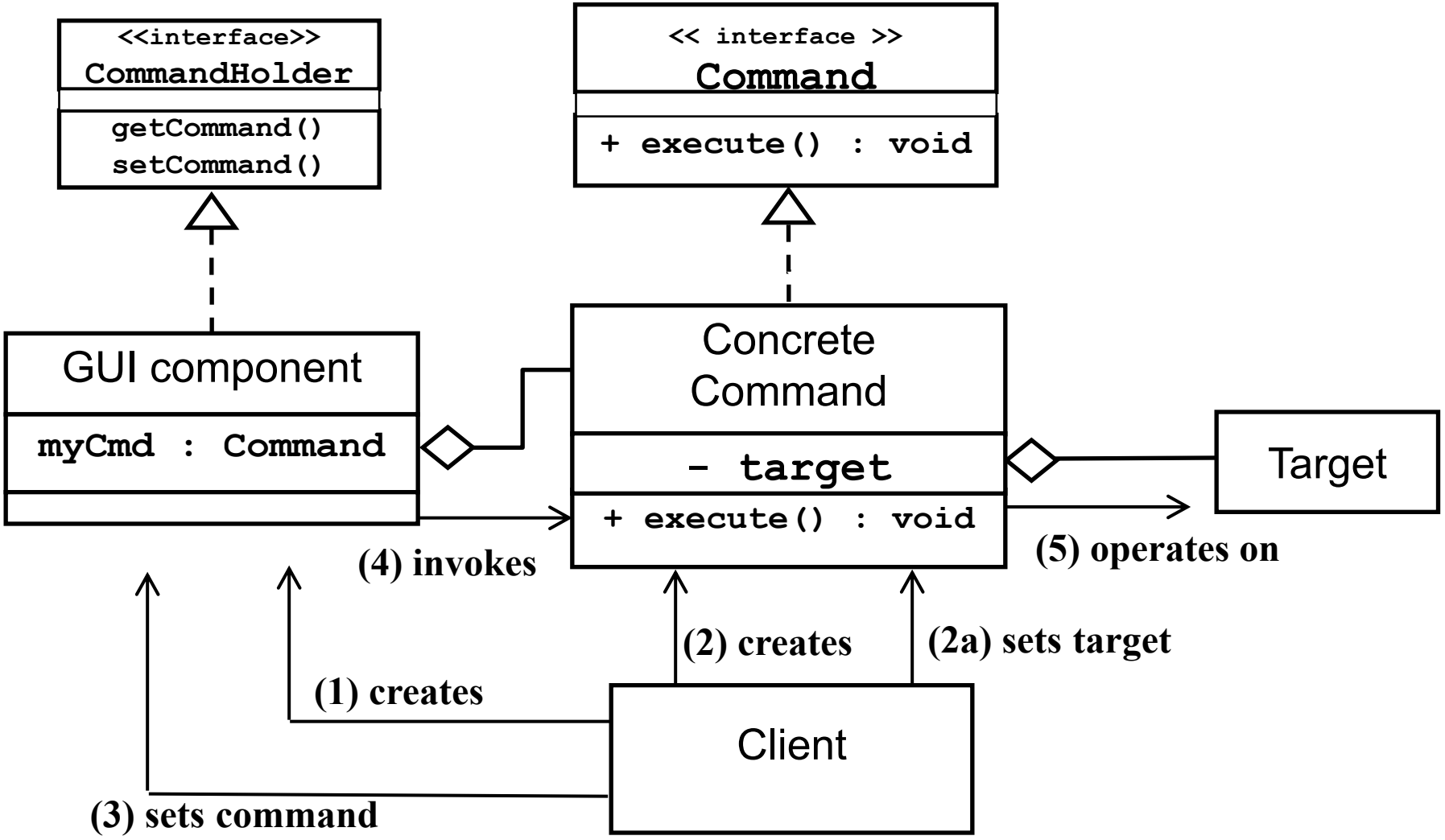
Idea of Command

- Similar to order food in restaurants

Invoker	**Command**	**Receiver**
- You - Send a message to a servers	- Servers - Place a paper order	- Cook - Read the paper and cook

You don’t need multiple cook

Command Pattern Organization



Next

Common Design Patterns

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

Common Design Patterns

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

Any Questions?