

*CSC 335: Object-Oriented
Programming and Design*



Object-Oriented Design Patterns



Outline

- ✦ Overview of Design Patterns

- ✦ Four Design Patterns

- Iterator

- Decorator

- Strategy

- Observer

The Beginning of Patterns

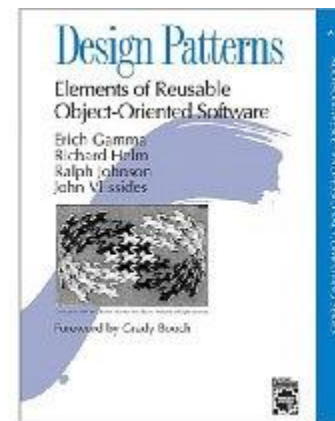
✦ Christopher Alexander, architect

- A Pattern Language--Towns, Buildings, Construction
- Timeless Way of Building (1979)
- “Each pattern describes a *problem* which occurs over and over again in our environment, and then describes the core of the *solution* to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.”

✦ Other patterns: novels (tragic, romantic, crime), movies genres (drama, comedy, documentary)

“Gang of Four” (GoF) Book

- ✦ Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Publishing Company, 1994
- ✦ Written by this "gang of four"
 - Dr. Erich Gamma, then Software Engineer, Taligent, Inc.
 - Dr. Richard Helm, then Senior Technology Consultant, DMR Group
 - Dr. Ralph Johnson, then and now at University of Illinois, Computer Science Department
 - Dr. John Vlissides, then a researcher at IBM
 - Thomas J. Watson Research Center
 - See John's WikiWiki tribute page <http://c2.com/cgi/wiki?JohnVlissides>





Object-Oriented Design Patterns

- ✦ This book defined 23 patterns in three categories
 - *Creational patterns* deal with the process of object creation
 - *Structural patterns*, deal primarily with the static composition and structure of classes and objects
 - *Behavioral patterns*, which deal primarily with dynamic interaction among classes and objects

Documenting Discovered Patterns

- ✦ Many other patterns have been introduced documented
 - For example, the book **Data Access Patterns** by Clifton Nock introduces 4 decoupling patterns, 5 resource patterns, 5 I/O patterns, 7 cache patterns, and 4 concurrency patterns.
 - Other pattern languages include telecommunications patterns, pedagogical patterns, analysis patterns
 - Patterns are mined at places like [Patterns Conferences](#)

ChiliPLoP

Recent patterns books work shopped at ChiliPLoP, Wickenburg and Carefree Arizona

- Patterns of Enterprise Application Architecture Martin Fowler
- Patterns of Fault Tolerant Software, Bob Hamner
- Patterns in XML Fabio Arciniegas
- Patterns of Adopting Agile Development Practices Amr Elssamadisy
- 2010: Patterns of Parallel Programming, Ralph Johnson
 - 16 patterns and one Pattern Language work shopped

GoF Patterns

– *Creational Patterns*

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

– *Structural Patterns*

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

– *Behavioral Patterns*

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

Why Study Patterns?

✦ Reuse tried, proven solutions

- Provides a head start
- Avoids gotchas later (unanticipated things)
- No need to reinvent the wheel

✦ Establish common terminology

- Design patterns provide a common point of reference
- Easier to say, “We could use Strategy here.”

✦ Provide a higher level prospective

- Frees us from dealing with the details too early



Other advantages

- ✦ Most design patterns make software more modifiable, less brittle
 - we are using time tested solutions
- ✦ Using design patterns makes software systems easier to change—more maintainable
- ✦ Helps increase the understanding of basic object-oriented design principles
 - encapsulation, inheritance, interfaces, polymorphism

Style for Describing Patterns

✦ We will use this structure:

- *Pattern name*
- *Recurring problem*: what problem the pattern addresses
- *Solution*: the general approach of the pattern
- *UML for the pattern*
 - *Participants*: a description as a class diagram
- *Use Example(s)*: examples of this pattern, in Java

A few OO Design Patterns

✦ Coming up:

– **Iterator**

- access the elements of an aggregate object sequentially without exposing its underlying representation

– **Strategy**

- A means to define a family of algorithms, encapsulate each one as an object, and make them interchangeable

– **Observer** *a preview*

- One object stores a list of observers that are updated when the state of the object is changed



Iterator

Pattern: *Iterator*

- ✦ Name: Iterator (a.k.a Enumeration)
- ✦ Recurring Problem: How can you loop over all objects in any collection. You don't want to change client code when the collection changes. Want the same methods
- ✦ Solution: 1) Have each class implement an interface, and 2) Have an interface that works with all collections
- ✦ Consequences: Can change collection class details without changing code to traverse the collection

GoF Version of Iterator

page 257

ListIterator
First()
Next()
IsDone()
CurrentItem()

// A C++ Implementation

```
ListIterator<Employee> itr = list.iterator();  
for(itr.First(); !itr.IsDone(); itr.Next()) {  
    cout << itr.CurrentItem().toString();  
}
```



Java version of Iterator

interface Iterator

boolean hasNext()

Returns true if the iteration has more elements.

Object next()

Returns the next element in the iteration and updates the iteration to refer to the next (or have hasNext() return false)

void remove()

Removes the most recently visited element

Java's Iterator interface

```
// The Client code
```

```
List<BankAccount> bank =
```

```
    new ArrayList<BankAccount>();
```

```
bank.add(new BankAccount("One", 0.01) );
```

```
// ...
```

```
bank.add(new BankAccount("Nine thousand", 9000.00));
```

```
String ID = "Two";
```

```
Iterator<BankAccount> itr = bank.iterator();
```

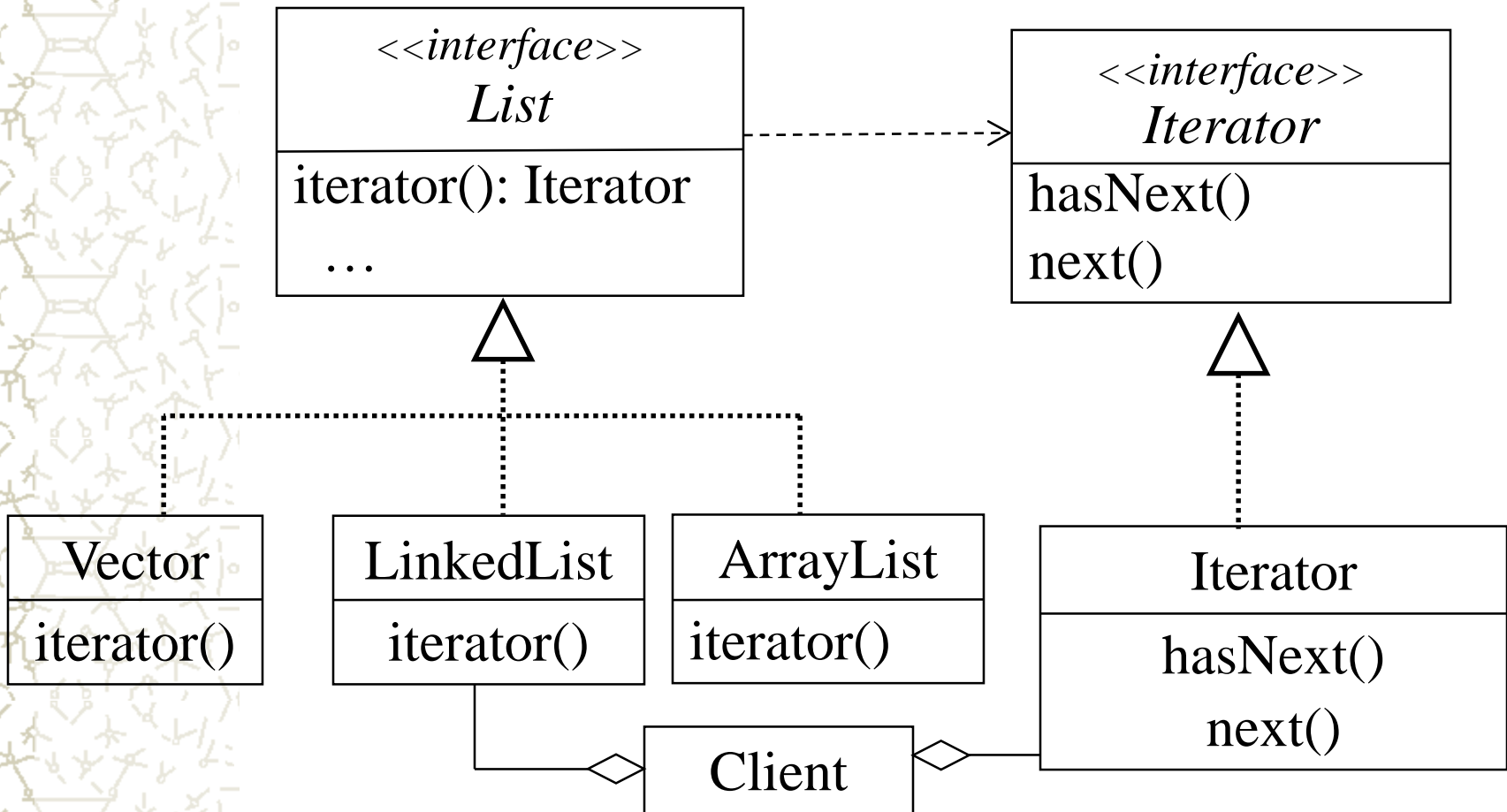
```
while(itr.hasNext()) {
```

```
    if(itr.next().getID().equals(searchAcct.getID()))
```

```
        System.out.println("Found " + ref.getID());
```

```
}
```

UML Diagram of Java's Iterator with a few Collections





Decorator Design Pattern

Rick Mercer
*CSC 335: Object-Oriented
Programming and Design*

The Decorator Pattern from GoF

🔦 Intent

- Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub classing to extend flexibility

🔦 Also Known As Wrapper

🔦 Motivation

- Want to add properties to an existing object.

🔦 2 Examples

- Add borders or scrollbars to a GUI component
- Add stream functionality such as reading a line of input or compressing a file before sending it over the wire

Applicability

Use Decorator

- To add responsibilities to individual objects dynamically without affecting other objects
- When extending classes is impractical
 - Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination (this inheritance approach is on the next few slides)

An Application

- ✚ Suppose there is a TextView GUI component and you want to add different kinds of borders and/or scrollbars to it
- ✚ You can add 3 types of borders
 - Plain, 3D, Fancy
- ✚ and 1 or 2 two scrollbars
 - Horizontal and Vertical
- ✚ An inheritance solution requires 15 classes for one view

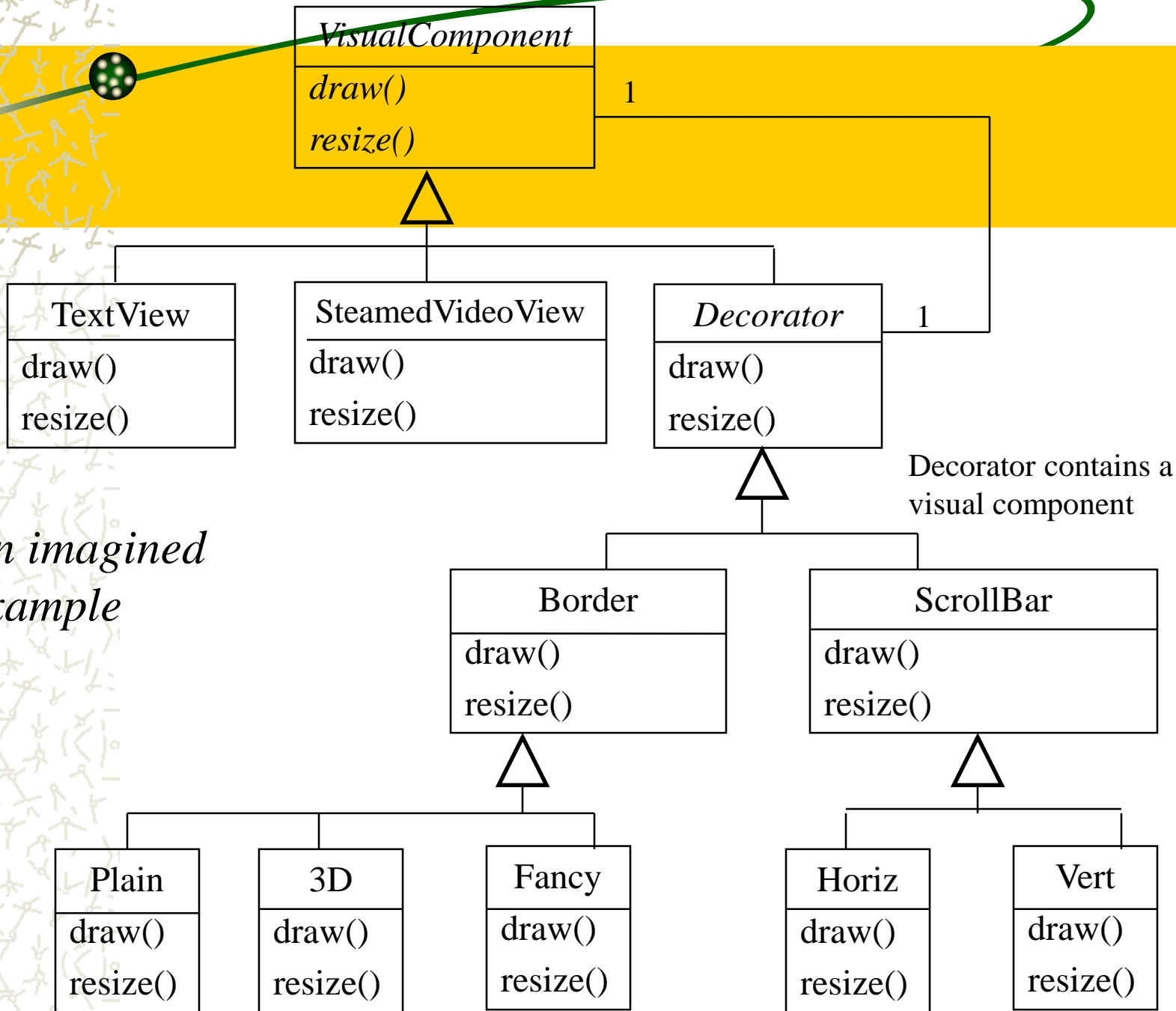


That's a lot of classes!

- 1.TextView_Plain
- 2.TextView_Fancy
- 3.TextView_3D
- 4.TextView_Horizontal
- 5.TextView_Vertical
- 6.TextView_Horizontal_Vertical
- 7.TextView_Plain_Horizontal
- 8.TextView_Plain_Vertical
- 9.TextView_Plain_Horizontal_Vertical
- 10.TextView_3D_Horizontal
- 11.TextView_3D_Vertical
- 12.TextView_3D_Horizontal_Vertical
- 13.TextView_Fancy_Horizontal
- 14.TextView_Fancy_Vertical
- 15.TextView_Fancy_Horizontal_Vertical

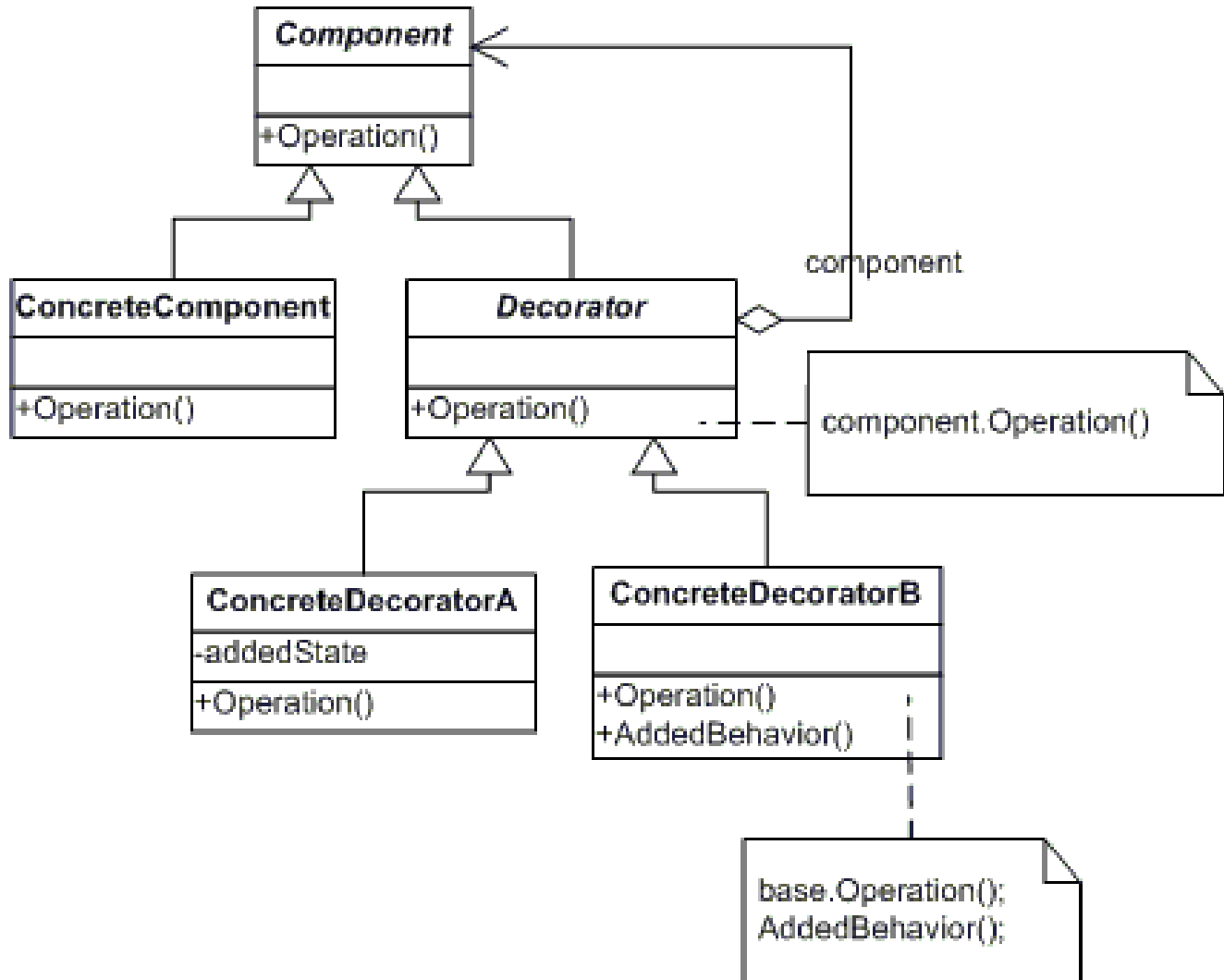
Disadvantages

- ✚ Inheritance solution has an explosion of classes
- ✚ If another view were added such as StreamedVideoView, double the number of Borders/Scrollbar classes
- ✚ Solution to this explosion of classes?
 - Use the Decorator Pattern instead



An imagined example

Decorator's General Form



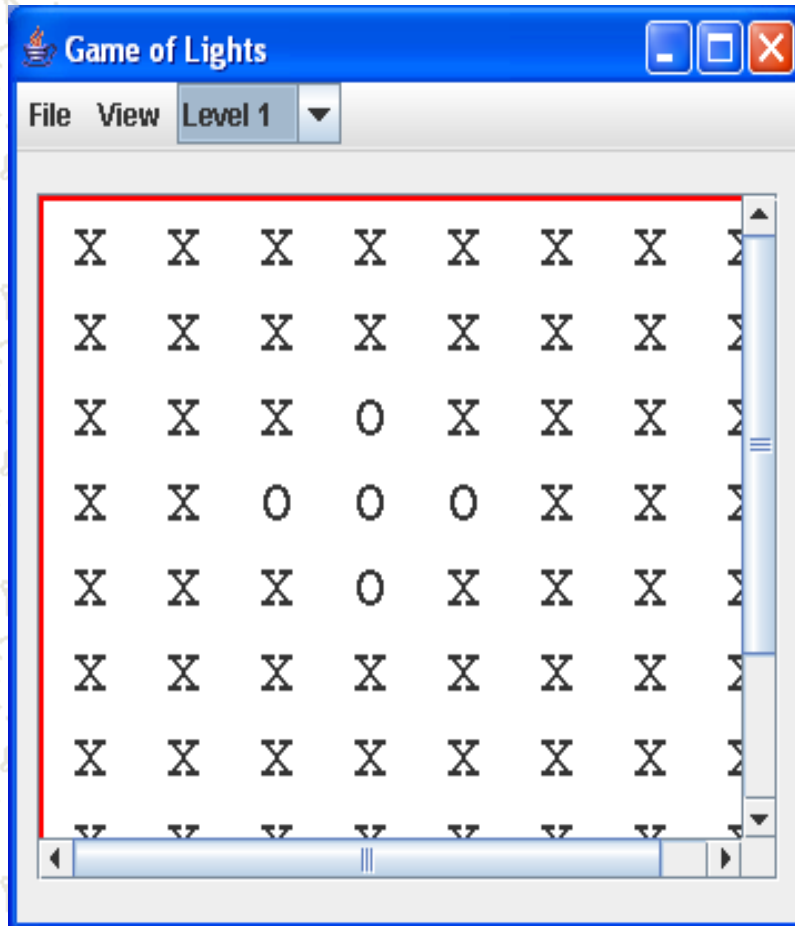


JScrollPane

- ✚ Any Component such as Container, JList, Panel can be decorated with a JScrollPane
- ✚ The next slide shows how to decorate a JPanel with a JScrollPane

Decorate a JPanel

```
JScrollPane scrollPane = new JScrollPane(toStringView);  
add(scrollPane); // Add to a JFrame or another panel
```



Motivation Continued

- ✦ The more flexible containment approach encloses the component in another object that adds the border
- ✦ The enclosing object is called the **decorator**
- ✦ The decorator conforms to the interface of the component so its presence is transparent to clients
- ✦ The decorator forwards requests to the component and may perform additional actions before or after any forwarding

Decorator Design: Java Streams

- ✚ `InputStreamReader(InputStream in)` *System.in is an InputStream object*
 - ... bridge from byte streams to character streams: It reads bytes and translates them into characters using the specified character encoding. Java™API
- ✚ `BufferedReader`
 - Read text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines. Java™API
- ✚ What we had to do for console input before Java 1.5's Scanner

```
BufferedReader keyboard =
    new BufferedReader(new
        InputStreamReader(System.in));
```

Decorator pattern in the real world

BufferedReader *decorates* InputStreamReader

BufferedReader

```
readLine() // add a useful method
```

InputStreamReader

```
read() // 1 byte at a time
```

```
close()
```

Still needed to parse integers, doubles, or words

Java streams

- With > 60 streams in Java, you can create a wide variety of input and output streams
 - this provides flexibility *good*
 - it also adds complexity
 - Flexibility made possible with inheritance and classes that accept classes that extend the parameter type

Another Decorator Example

- ✦ We decorated a `FileInputStream` with an `ObjectInputStream` to read objects that implement `Serializable`
 - and we used `FileOutputStream` with `ObjectOutputStream`
 - then we were able to use nice methods like these two read and write large complex objects on the file system:

```
outFile.writeObject(list);
```

```
// and later on ...
```

```
list = (ArrayList<String>)inFile.readObject();
```

Another Decorator Example

- ✦ Read a plain text file and compress it using the GZIP format `ZIP.java`
- ✦ Read a compress file in the GZIP format and write it to a plain text file `UNGZIP.java`
- ✦ Sample text [iliad10.txt](#) *from Project Gutenberg*

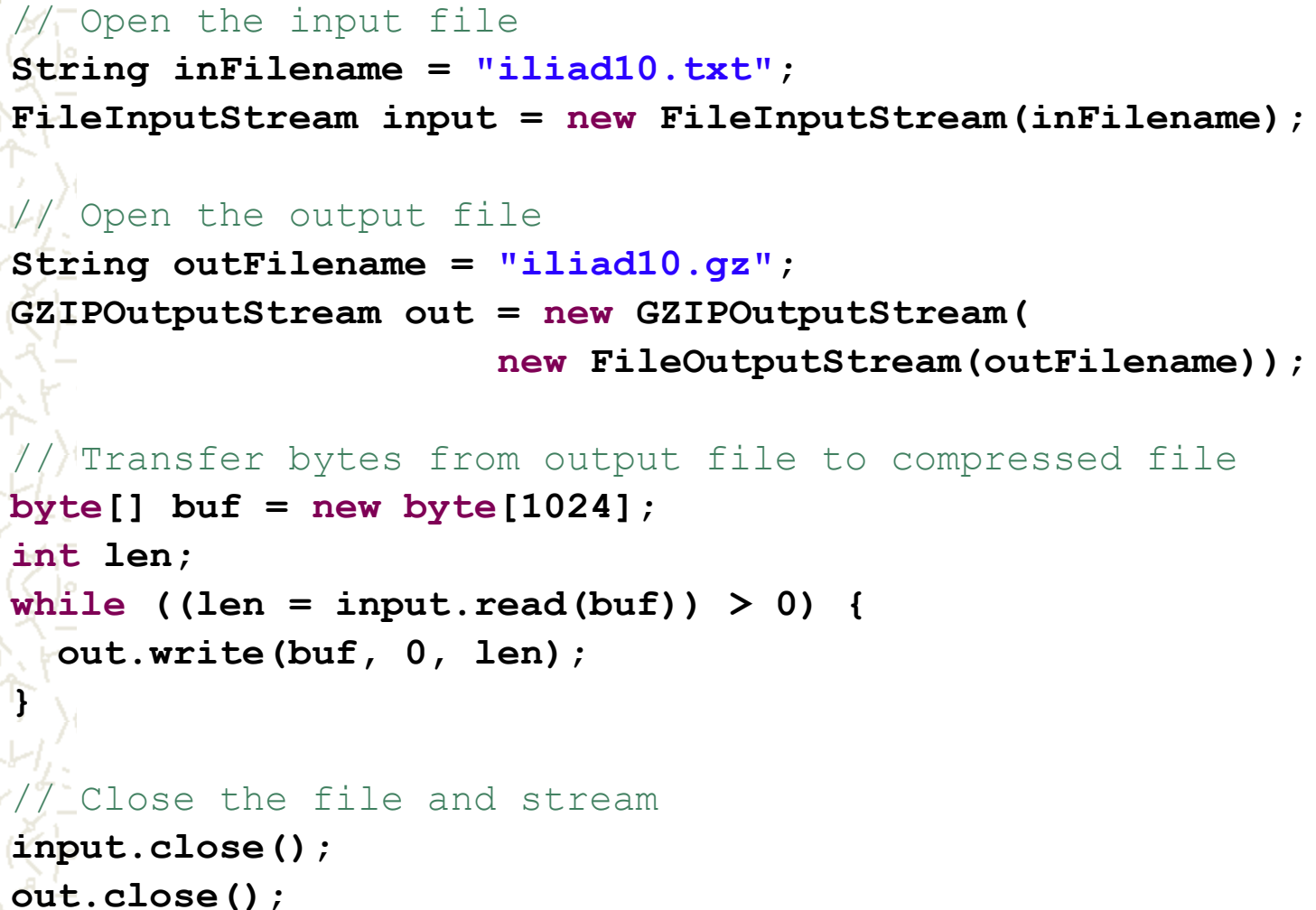
bytes

875,736 `iliad10.txt` bytes

305,152 `iliad10.gz`

875,736 `TheIliadByHomer`

(after code on next slide)

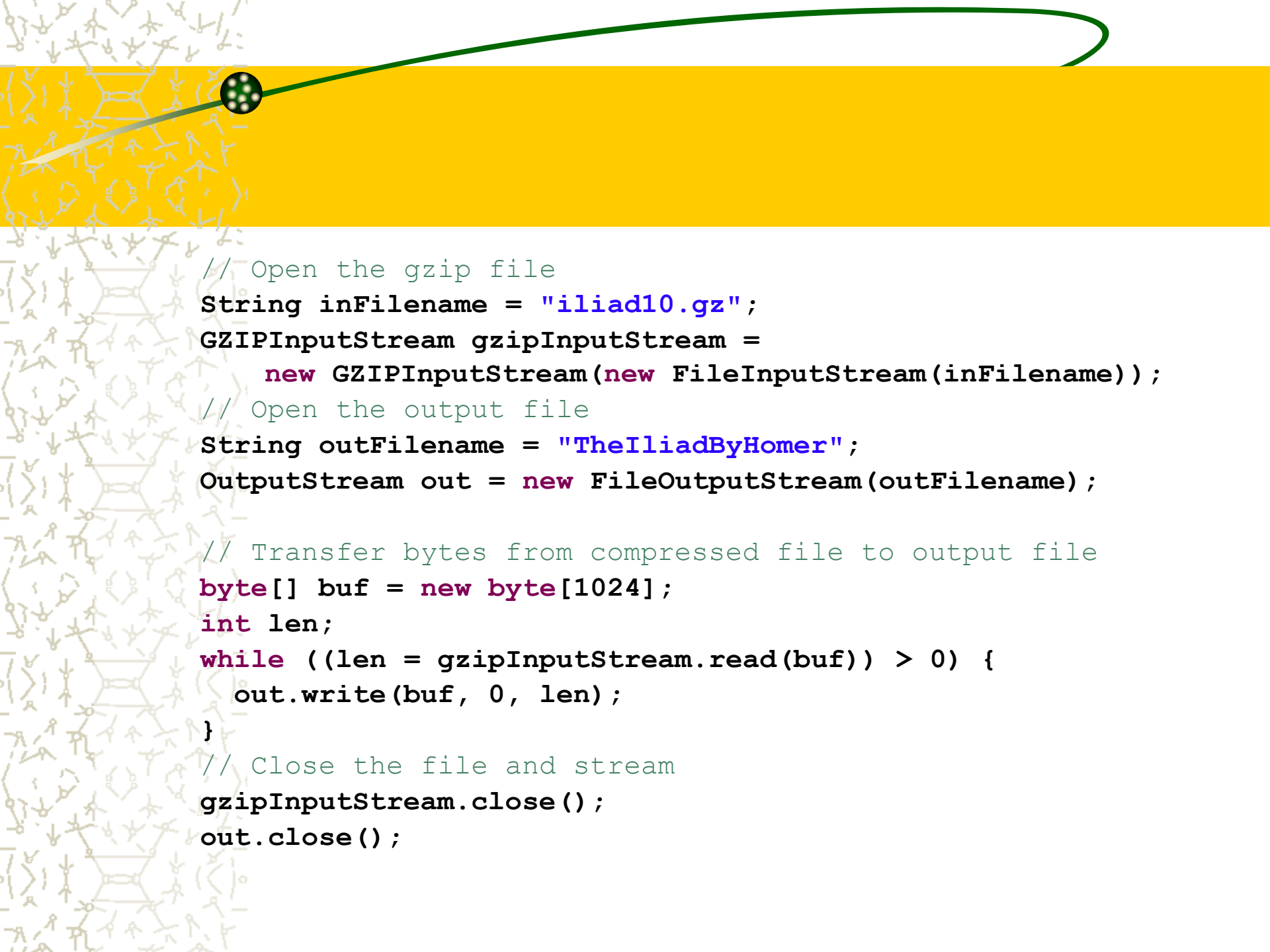


```
// Open the input file
String inFilename = "iliad10.txt";
FileInputStream input = new FileInputStream(inFilename);

// Open the output file
String outFilename = "iliad10.gz";
GZIPOutputStream out = new GZIPOutputStream(
    new FileOutputStream(outFilename));

// Transfer bytes from output file to compressed file
byte[] buf = new byte[1024];
int len;
while ((len = input.read(buf)) > 0) {
    out.write(buf, 0, len);
}

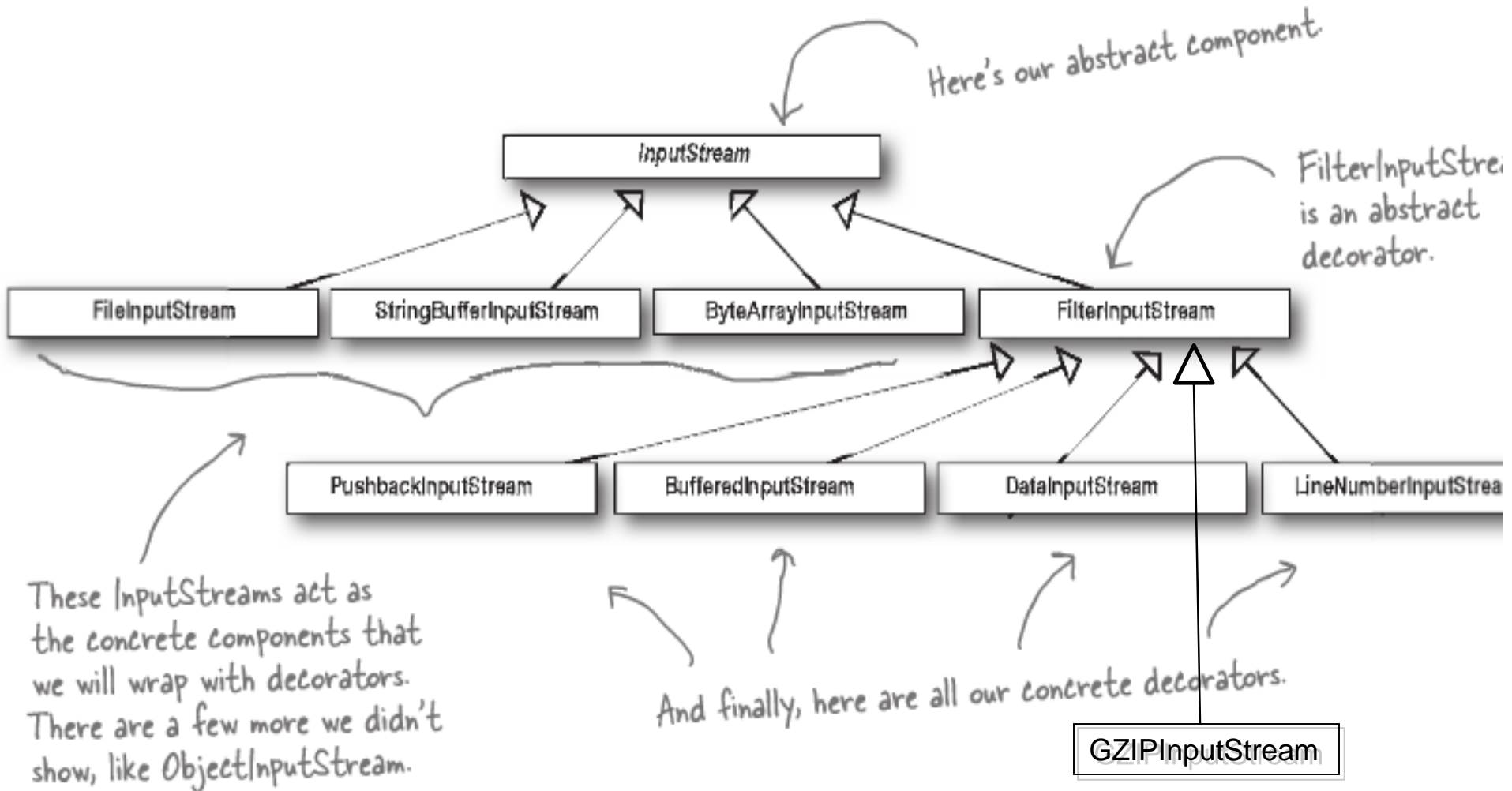
// Close the file and stream
input.close();
out.close();
```



```
// Open the gzip file
String inFilename = "iliad10.gz";
GZIPInputStream gzipInputStream =
    new GZIPInputStream(new FileInputStream(inFilename));
// Open the output file
String outFilename = "TheIliadByHomer";
OutputStream out = new FileOutputStream(outFilename);

// Transfer bytes from compressed file to output file
byte[] buf = new byte[1024];
int len;
while ((len = gzipInputStream.read(buf)) > 0) {
    out.write(buf, 0, len);
}
// Close the file and stream
gzipInputStream.close();
out.close();
```

GZIPInputStream is a Decorator





Summary

- ✦ Decorators are very flexible alternative of inheritance
- ✦ Decorators enhance (or in some cases restrict) the functionality of decorated objects
- ✦ They work dynamically to extend class responsibilities, even inheritance does some but in a static fashion at compile time



Strategy Design Pattern

Strategy

Pattern: *Strategy*

- ✚ **Name:** Strategy (a.k.a Policy)
- ✚ **Problem:** You want to encapsulate a family of algorithms and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it (GoF)
- ✚ **Solution:** Create an abstract strategy class (or interface) and extend (or implement) it in numerous ways. Each subclass defines the same method names in different ways

Design Pattern: *Strategy*

Consequences:

- Allows families of algorithms

Known uses:

- Critters seen in section for Rick's 127B / 227
- Layout managers in Java
- Different Poker Strategies in a 335 Project
- Different PacMan chase strategies in a 335 Project
- Different Jukebox policies that can be

Java Example of Strategy

```
this.setLayout(new FlowLayout());  
this.setLayout(new GridLayout());
```

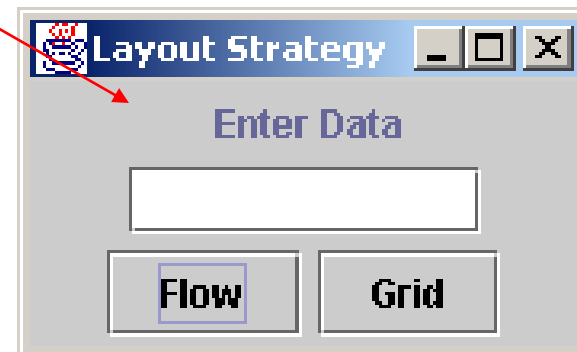
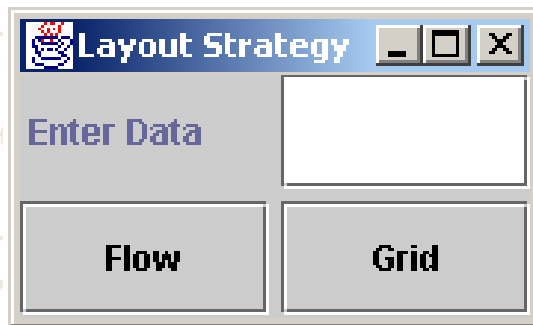
✿ In Java, a container HAS-A layout manager

- There is a default
- You can change a container's layout manager with a **setLayout** message

Change the strategy at runtime

◆ Demonstrate LayoutControllerFrame.java

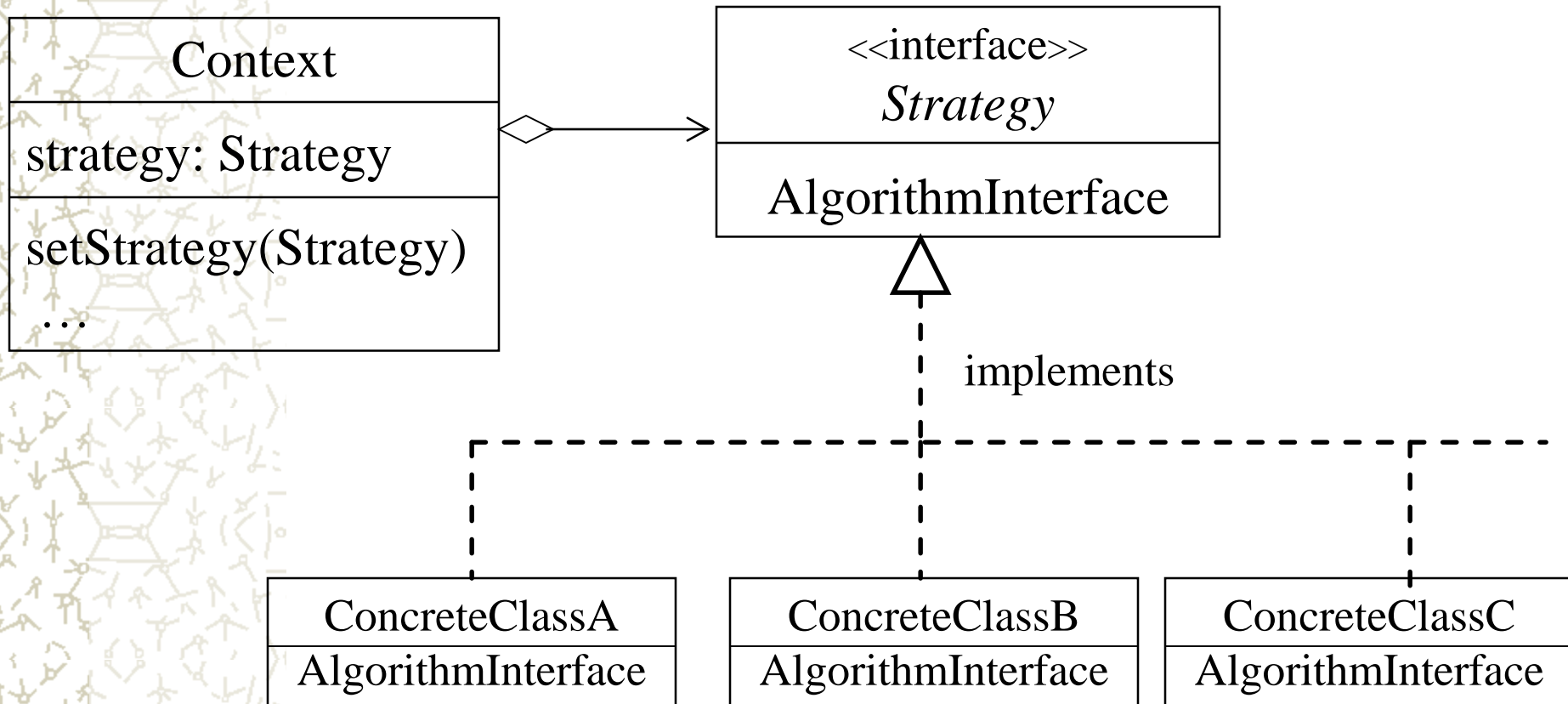
```
private class FlowListener
    implements ActionListener {
    // There is another ActionListener for GridLayout
    public void actionPerformed(ActionEvent evt) {
        // Change the layout strategy of the JPanel
        // and tell it to lay itself out
        centerPanel.setLayout(new FlowLayout());
        centerPanel.validate();
    }
}
```



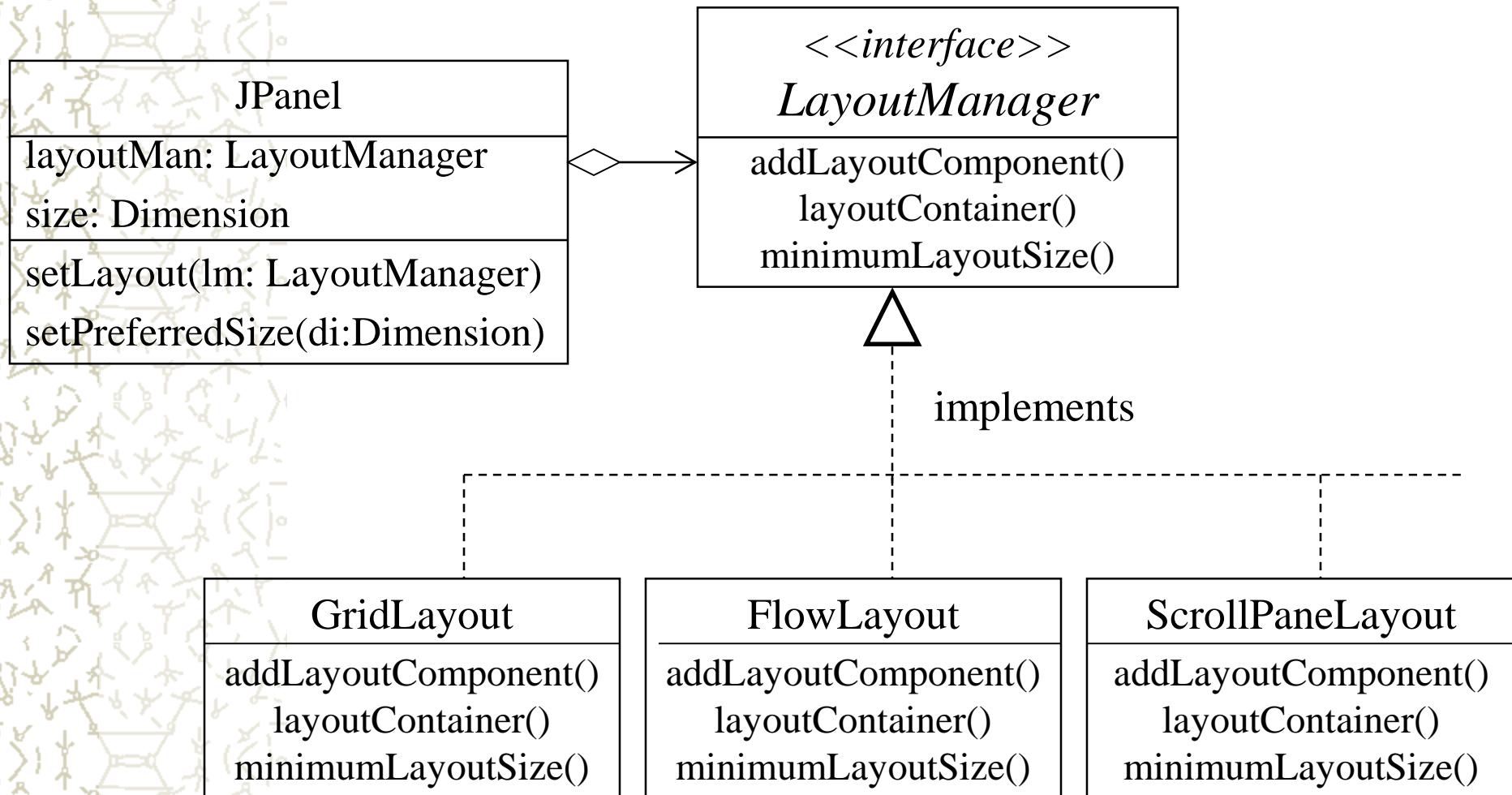
interface `LayoutManager`

- Java has interface [`java.awt.LayoutManager`](#)
- Known Implementing Classes
 - `GridLayout`, `FlowLayout`, `ScrollPaneLayout`
- Each class implements the following methods
 - `addLayoutComponent(String name, Component comp)`
 - `layoutContainer(Container parent)`
 - `minimumLayoutSize(Container parent)`
 - `preferredLayoutSize(Container parent)`
 - `removeLayoutComponent(Component comp)`

UML Diagram of Strategy General Form



Specific UML Diagram of LayoutManager in Java



Another Example

- Pac Man GhostChasesPacMan strategies in 2001
 - Level 1: random
 - Level 2: a bit smarter
 - Level 3: use a shortest path algorithm

<http://www.martystepp.com/applets/pacman/>

- Could be interface ChaseStrategy is in the Ghost class

```
interface ChaseStrategy {  
    public Point nextPointToMoveTo();  
}
```

- and Ghost has setChaseStrategy(new ShortestPath())



The Observer Design Pattern

- ✦ Name: Observer
- ✦ Problem: Need to notify a changing number of objects that something has changed
- ✦ Solution: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically

Examples

- ✦ From Heads-First: Send a newspaper to all who subscribe

- People add and drop subscriptions, when a new version comes out, it goes to all currently described

- ✦ Spreadsheet

- Demo: Draw two charts—two views--with some changing numbers--the model

Examples

- ✦ File Explorer (or Finders) are registered observers (the view) of the file system (the model).
- ✦ Demo: Open several finders to view file system and delete a file
- ✦ Later in Java: We'll have two views of the same model that get an update message whenever the state of the model has changed

Observer Example

