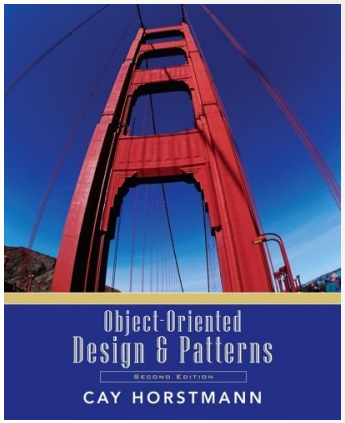
Object-Oriented Design & Patterns

2nd edition

Cay S. Horstmann



Chapter 5: Patterns and GUI Programming

CPSC 2100

Software Design and Development

Chapter Topics

- The Iterators as Patterns.
- The Pattern Concept.
- The OBSERVER Pattern.
- Layout Managers and the STRATEGY Pattern.
- Components, Containers, and the COMPOSITE Pattern.
- Scroll Bars and the DECORATOR Pattern.
- How to Recognize Patterns.
- Putting Patterns to Work.

Chapter Objective

- Introduce the concept of Patterns.
- Explain Patterns with examples from the Swing user interface toolkit, to learn about Patterns and GUI programming at the same time.

List Iterators

```
LinkedList<String> list = . . .;
ListIterator<String> iterator = list.listIterator();
while (iterator.hasNext())
{
      String current = iterator.next();
      . . .
}
```

Why iterators?

Classical List Data Structure

- Traverse links directly

```
Link currentLink = list.head;
while (currentLink != null)
{
    Object current = currentLink.data;
    currentLink = currentLink.next;
}
```

- Problems:
  - Exposes implementation of the links to the users.
  - Error-prone.

High-Level View of Data Structures

- Queue

```
void add(E x)
E peek()
E remove()
int size()
```

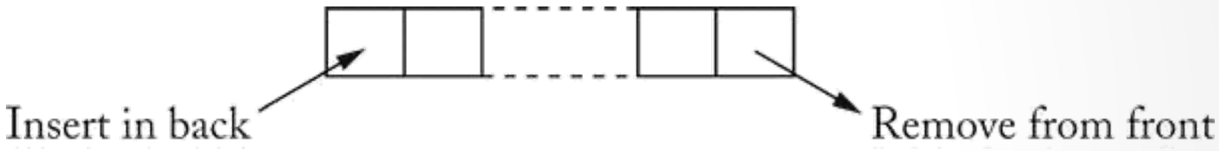


Figure 1: The Queue Interface

- Array with random access

```
E get(int i)
void set(int i, E x)
void add(E x)
int size()
```

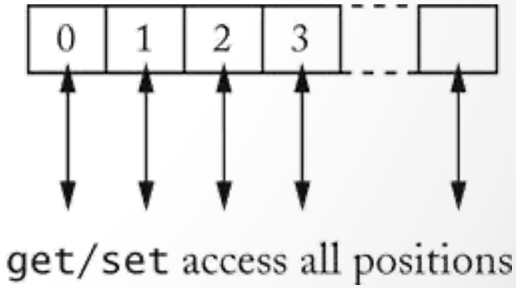


Figure 2: The Array Interface

List with Cursor

- Able to add and remove elements in the middle of the linked list.

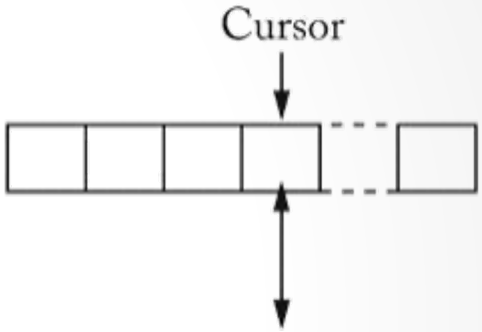


Figure 3: A List with a Cursor

- List with a Curser:

```
E getCurrunt() // Get element at cursor
void set(E x) // Set element at cursor x
E remove() // Remove element at cursor
void insert(E x) // Insert x before cursor
void reset() // Reset cursor to head
void next() // Advance cursor
boolean hasNext() // Check if cursor can be advanced
```

List with Cursor

```
for (list.reset(); list.hasNext(); list.next())
{
    Object x = list.get();
    . . .
}
```

- **Disadvantage:** Only one cursor per list.
  - cannot compare different list elements.
  - cannot print the contents of the list for debugging purposes(side effect of moving the curser to the end).

- **Iterator is superior concept.**
  - A list can have any number of iterators attached to it.

The Pattern Concept

- History: Architectural Patterns
- **Christopher Alexander** (*A pattern language: Towns, Building, Construction*, Oxford University Press, 1977).

- Each pattern has
  - a short *name*.
  - a brief description of the *context*.
  - a lengthy description of the *problem*.
  - a prescription for the solution.

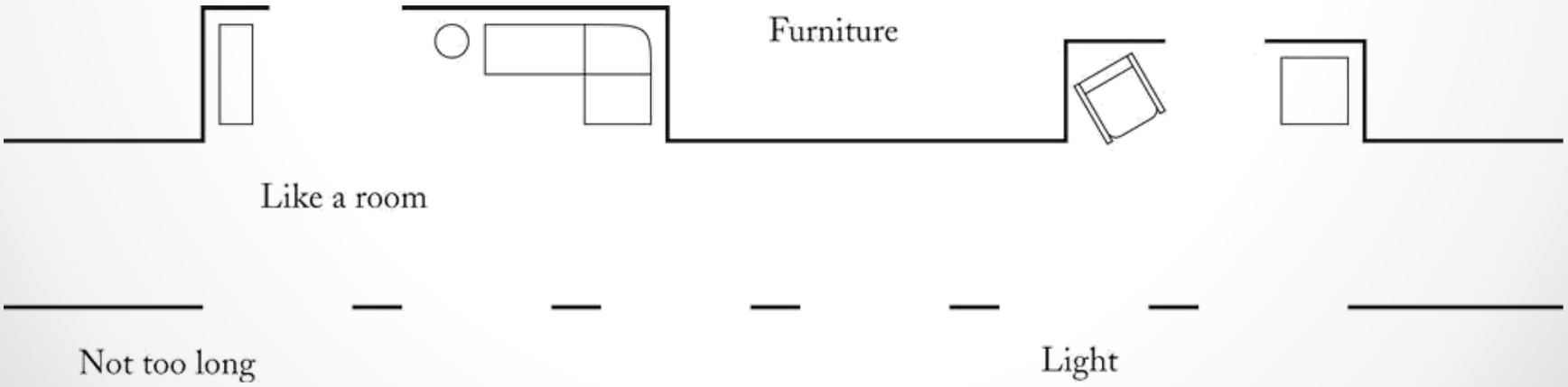
Short Passages Pattern

- **Context**
  - “ . . . long sterile corridors set the scene for everything bad about modern architecture.”

- **Problem**
  - This section contains a lengthy description of the problem of long corridors, with a depressing picture of a long, straight, narrow corridor with closed doors.

Short Passages Pattern

- **Solution**
  - Keep passages short. Make them as much like rooms as possible, with carpets or wood on the floor, furniture, bookshelves, beautiful windows. Make them generous in shape and always give them plenty of light; the best corridors and passages of all are those which have windows along an entire wall.



Patterns Summary

- Pattern distills a design rule into a simple format.
- **Check weather the pattern is useful to you?**
  - If so, follow the recipe for the solution.
    - Solution succeeded in the past.
    - You will benefit from it as well.

Iterator Pattern

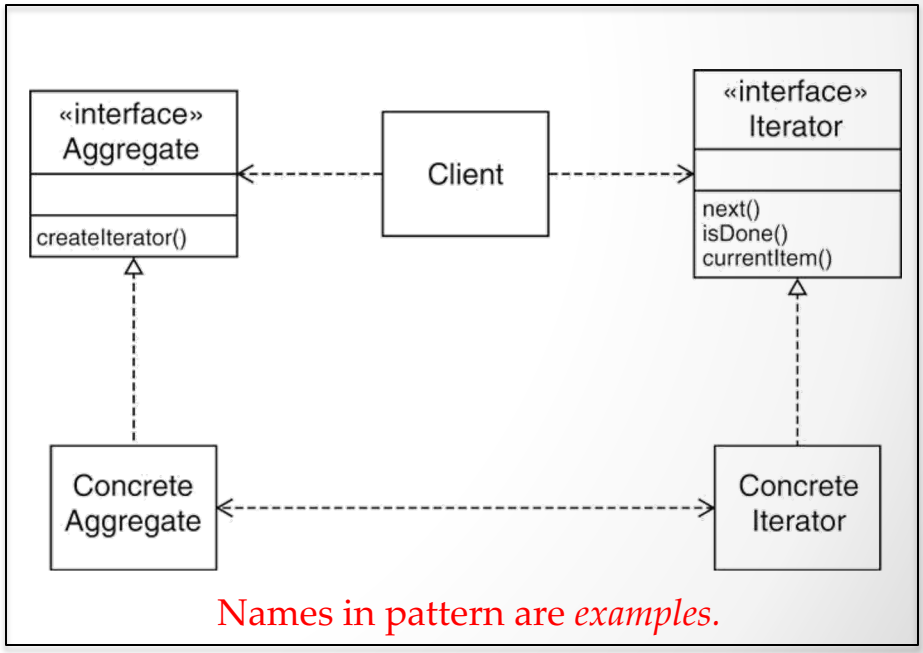
- **Context**
  1. An object (*aggregate*) contains other objects (*elements*).
  2. Clients (*methods*) need access to the element objects.
  3. The aggregate object should not expose its internal structure.
  4. Multiple clients may want independent (simultaneous) access.

Iterator Pattern

- **Solution**
  1. Define an ***iterator*** that fetches one element at a time.
  2. Each ***iterator*** object keeps track of the position of the next element.
  3. If there are several **aggregate/iterator** variations, it is best if the aggregate and iterator classes realize common interface types.

Name in Design Pattern	Actual Name (linked lists)
Aggregate	List
ConcreteAggregate	LinkedList
Iterator	ListIterator
ConcreteIterator	anonymous class implementing ListIterator
createIterator()	listIterator()
next()	next()
isDone()	opposite of hasNext()
currentItem()	return value of hasNext()

Names differ in each occurrence of pattern



Names in pattern are *examples*.

Model/View/Controller

- Some programs have multiple editable views.
- Example: HTML Editor
  - WYSIWYG view
  - structure view
  - source view
- Editing one view updates the other.
- Updates seem instantaneous.

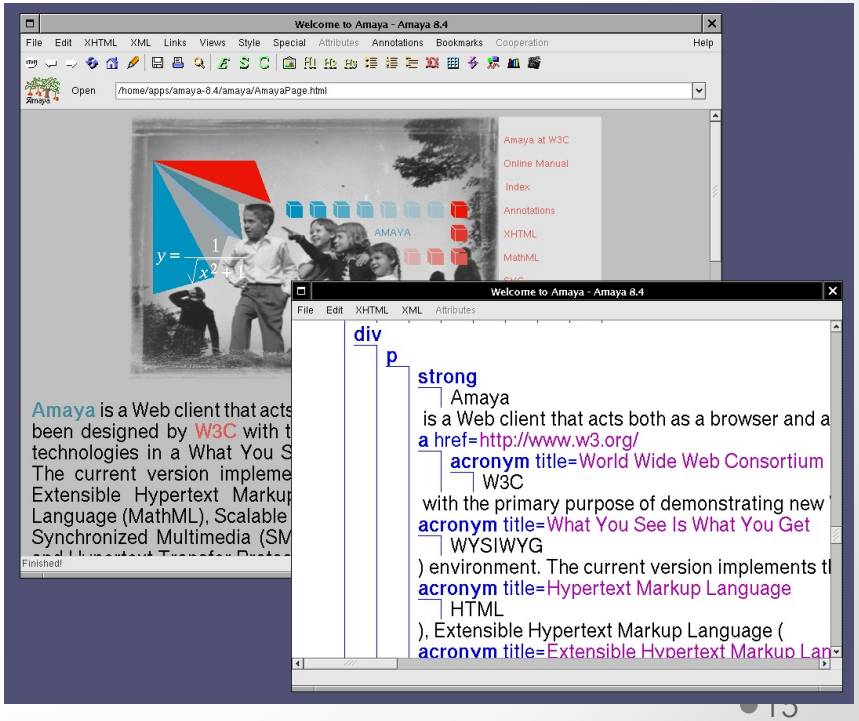


Figure 4:
WYSIWYG
and a Structural View of the
Same HTML Page

Model/View/Controller

- Model: raw data in a data structure, no visual representation.
- Views: visual representations (table, graph, ...).
- Controllers: user interaction.

Model/View/Controller

- What happened when a user types text into one of the windows:
  1. The controller tells the model to insert the text that the user typed.
  2. The model notifies all views of a change in the model.
  3. All views repaint themselves.
  4. During painting, each view asks the model for the current text.

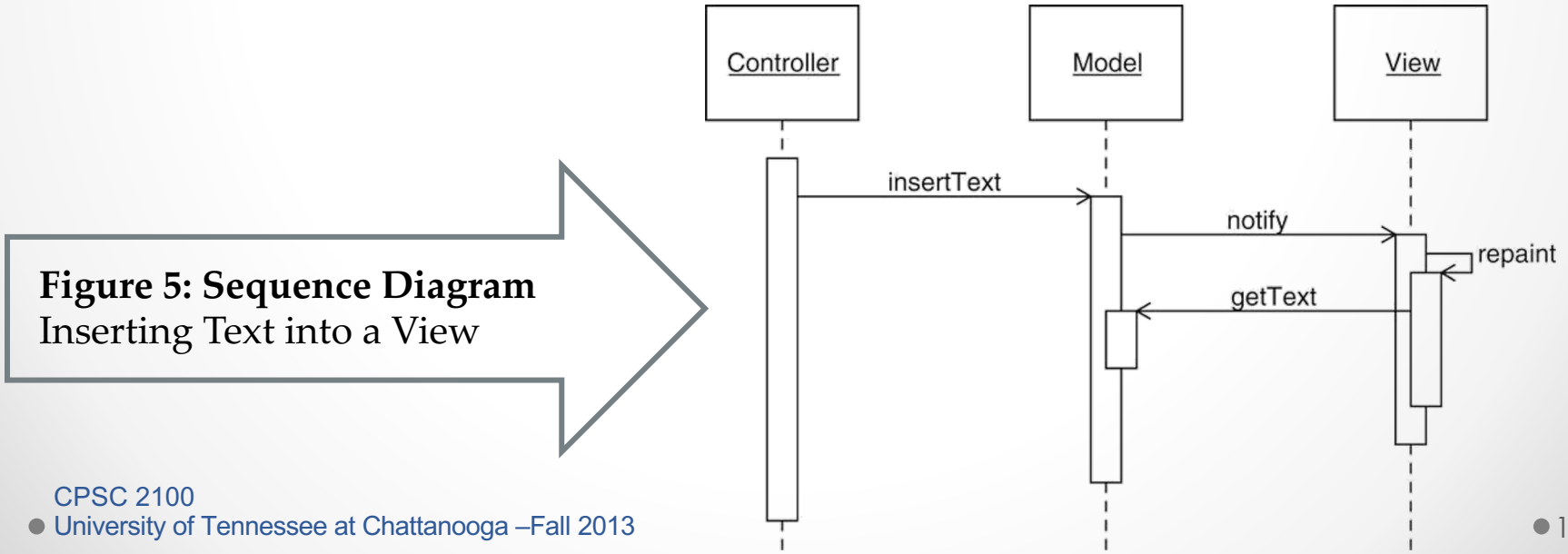


Figure 5: Sequence Diagram
Inserting Text into a View

Observer Pattern

- **Model** notifies views when something interesting happens.
  - Button notifies action listeners when something interesting happens.
- **Views** attach themselves to model in order to be notified.
  - Action listeners attach themselves to button in order to be notified.
- **Generalize:** *Observers* attach themselves to *subject*.

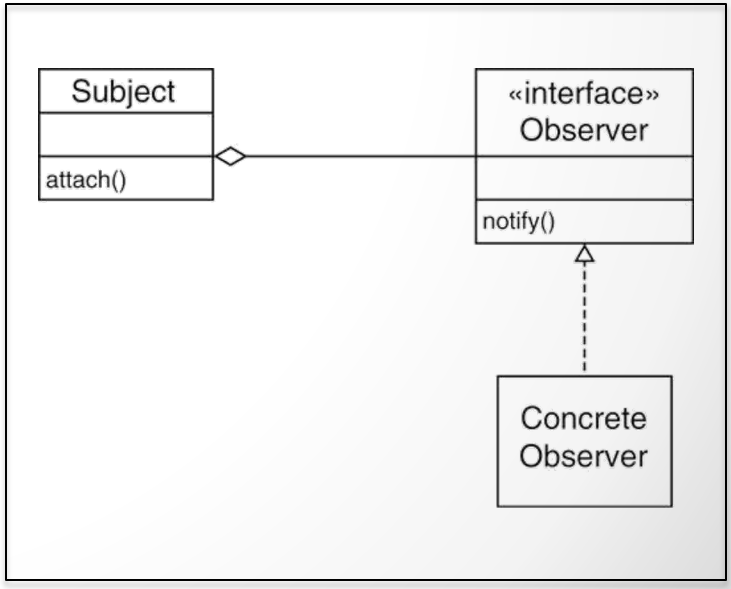
Observer Pattern

- **Context**
  1. An object, called the subject, is source of events.
  2. One or more observer objects want to be notified when such an event occurs.

Observer Pattern

- **Solution**
  1. Define an ***observer*** interface type. All concrete observers implement it.
  2. The ***subject*** maintains a collection of observers.
  3. The ***subject*** supplies methods for attaching and detaching observers.
  4. Whenever an event occurs, the subject notifies all observers.

Name in Design Pattern	Actual Name (Swing buttons)
Subject	JButton
Observer	ActionListener
ConcreteObserver	the class that implements the ActionListener interface type
attach()	addActionListener()
notify()	actionPerformed()



Layout Managers

- User interfaces made up of *components* (buttons, text fields, sliders, ...)
- Components placed in *containers* (frames)
- Container needs to arrange components.

- Swing doesn't use hard-coded pixel coordinates.
  - Advantages:
    - Can switch "look and feel"
    - Can internationalize strings.

- Layout manager controls arrangement.

Layout Managers
Predefined in Java

- **FlowLayout**
  - left to right, start new row when full.
- **BoxLayout**
  - left to right or top to bottom.
- **BorderLayout**
  - 5 areas, Center, North, South, East, West.
- **GridLayout**
  - grid, all components have same size.
- **GridBagLayout**
  - complex, like HTML table, grid but rows and columns can have different sizes and components can span multiple rows and columns.

Layout Managers

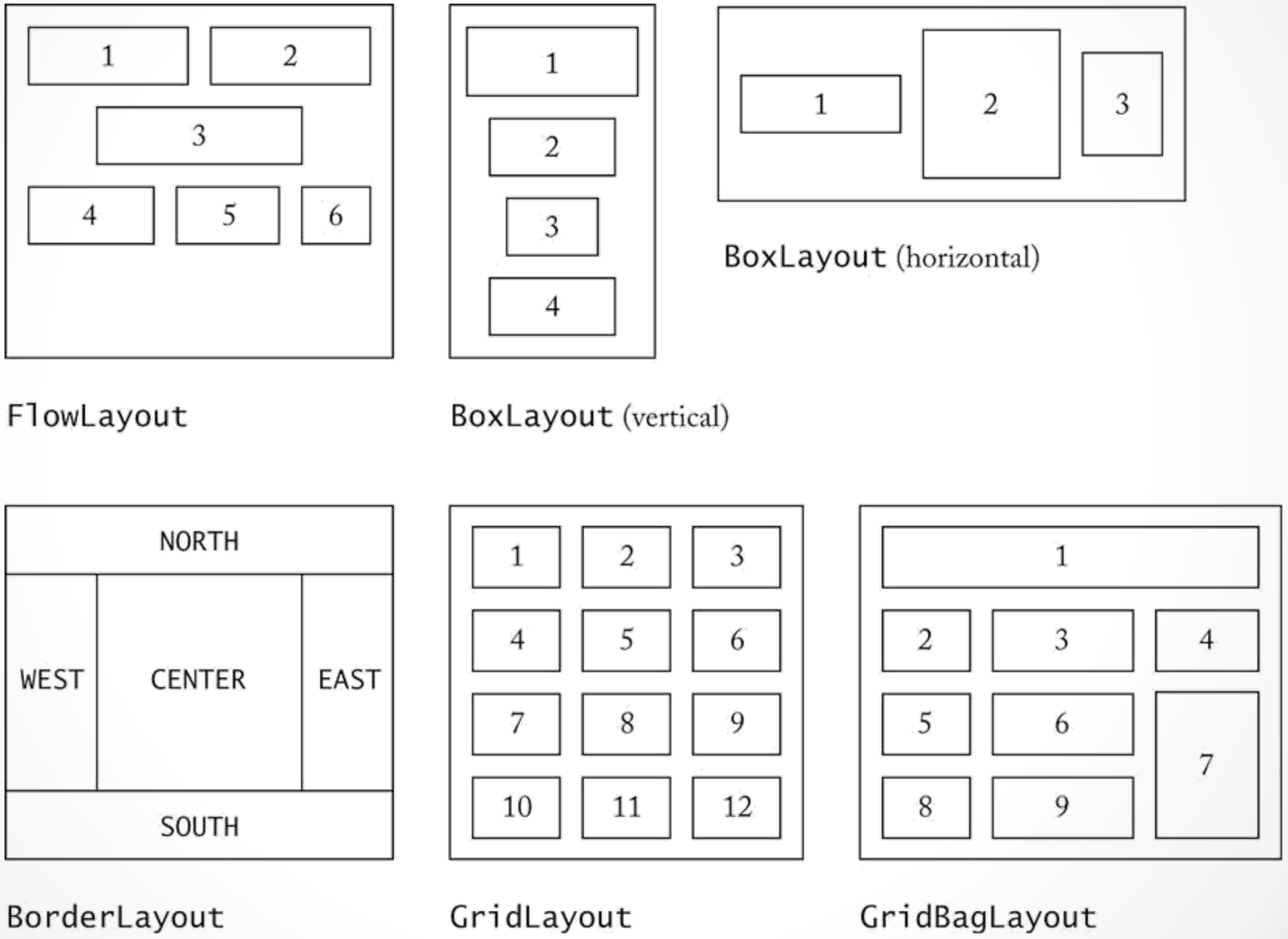


Figure 6: Layout Managers

Layout Managers

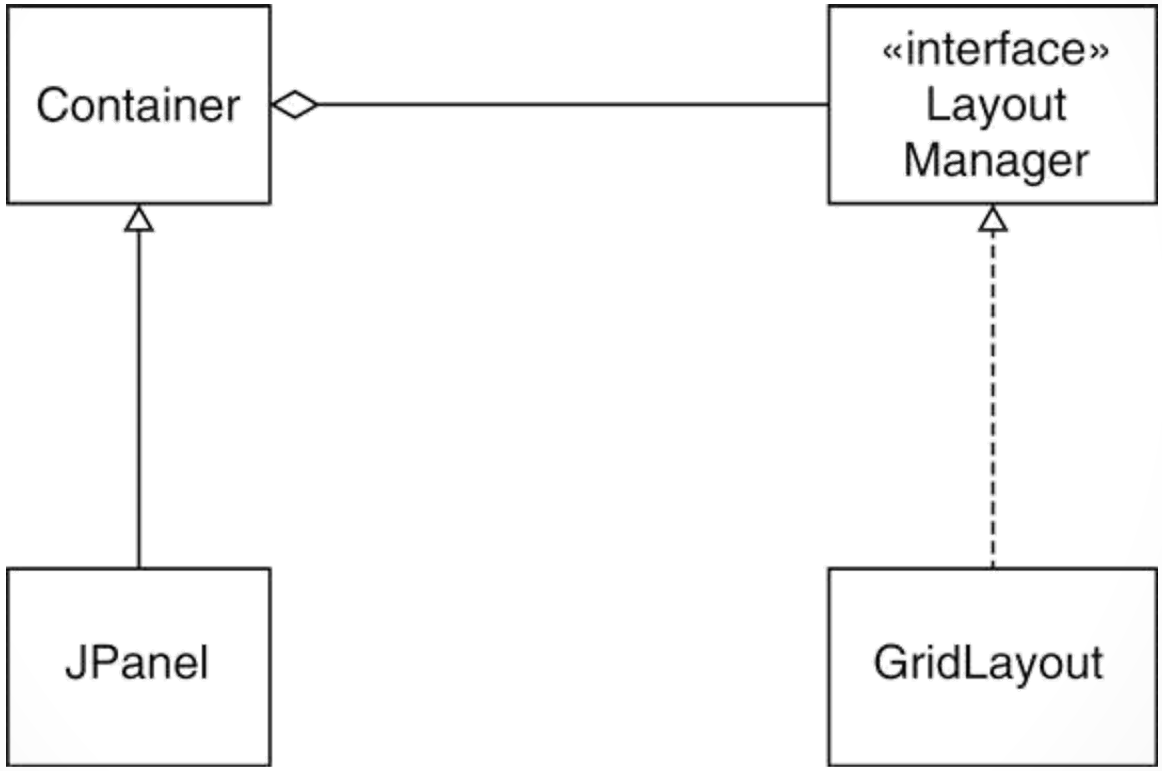


Figure 7: Layout Management Classes

Custom Layout Manager

- Form layout.
- Odd-numbered components right aligned.
- Even-numbered components left aligned.
- Implement LayoutManager interface type.

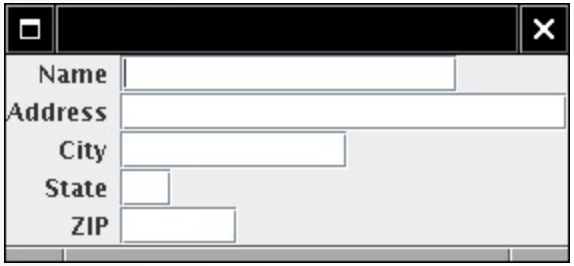


Figure 12: The FormLayout Custom Layout Manager

Note: Can use GridBagLayout to achieve the same effect.

The LayoutManager Interface Type

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

Strategy Pattern

- Standard layout managers can be used to provide a custom layout manager:
  1. Make an object of the layout manager class and give it to the container.
  2. When the container needs to execute the layout algorithm, it calls the appropriate methods of the layout manager object.

- Strategy Design Pattern.

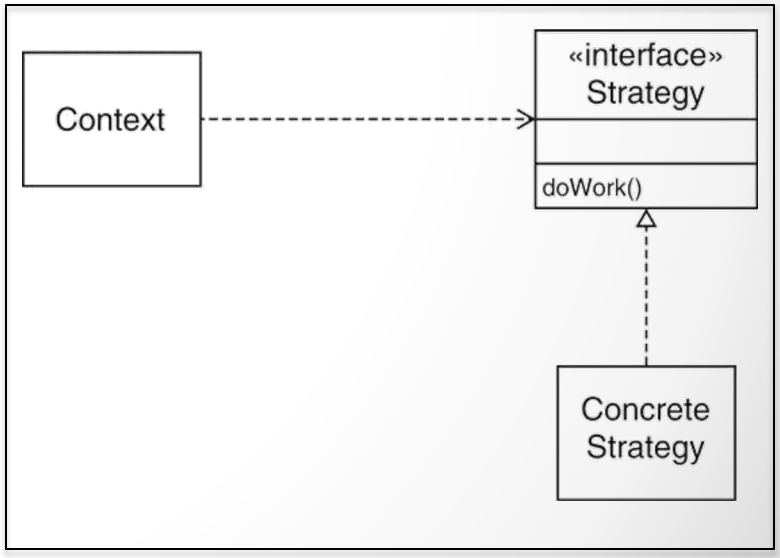
Strategy Pattern

- **Context**
  1. A class (*context class*) can benefit from different variants for an algorithm.
  2. Clients sometimes want to replace standard algorithms with custom versions.

Strategy Pattern

- **Solution**
  1. Define an interface type that is an abstraction for the algorithm.
  2. Actual strategy classes realize this interface type.
  3. Clients can supply strategy objects.
  4. Whenever the algorithm needs to be executed, the context class calls the appropriate methods of the strategy object.

Name in Design Pattern	Actual Name (layout management)
Context	Container
Strategy	LayoutManager
ConcreteStrategy	a layout manager such as BorderLayout
doWork()	a method such as layoutContainer



Strategy Pattern
Sorting

- Other manifestation: Comparators

```
Comparator<Country> comp = new CountryComparatorByName();
Collections.sort(countries, comp);
```

- The **comparator** object encapsulates the **comparison** algorithm.
- By varying the comparator, you can sort by different criteria.

Name in Design Pattern	Actual Name (sorting)
Context	Collections
Strategy	Comparator
ConcreteStrategy	a class that implements Comparator
doWork()	compare

Composite Pattern

- **Telephone example:**
  - Group components into a panel in order to achieve a satisfactory layout.

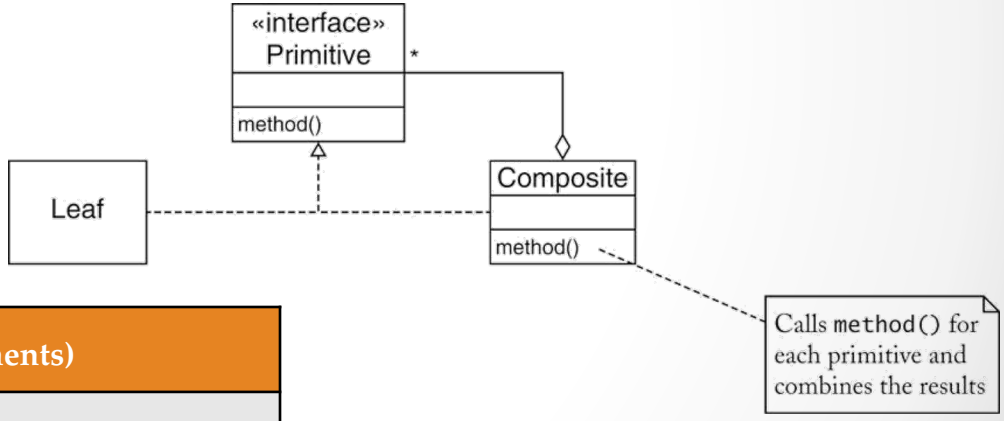
- **Technical issue:**
  - User interface components are contained in a containers.
  - JPanel can contain other components ⇨ JPanel is a container.
  - Add the panel to the frame ⇨ JPanel is a component.

  - **Can a container itself be a component?**

- **Context**
  1. Primitive objects can be combined to composite objects.
  2. Clients treat a composite object as a primitive object.

Composite Pattern

- **Solution**
  1. Define an interface type that is an abstraction for the primitive objects.
  2. Composite object collects primitive objects.
  3. Composite and primitive classes implement same interface type.
  4. When implementing a method from the interface type, the composite class applies the method to its primitive objects and combines the results.



Name in Design Pattern	Actual Name (AWT components)
Primitive	Component
Composite	Container
Leaf	a component without children (e.g. Jbutton)
method()	A method of Component(e.g. getPreferredSize)

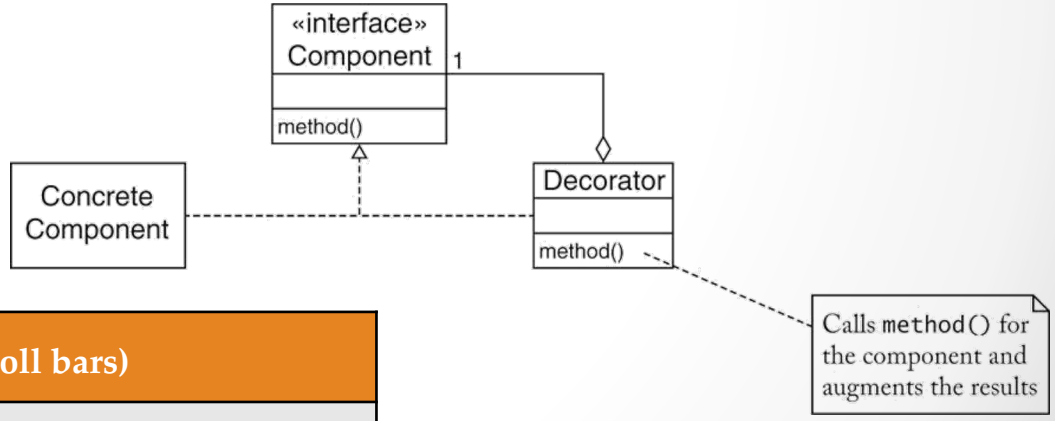
Decorator Pattern

- **Decorator Pattern** applies whenever a class enhances the functionality of another class while preserving its interface.

- **Context**
  1. Component objects can be decorated (visually or behaviorally enhanced)
  2. The decorated object can be used in the same way as the undecorated object.
  3. The component class does not want to take on the responsibility of the decoration.
  4. There may be an open-ended set of possible decorations.

Decorator Pattern

- **Solution**
  1. Define an interface type that is an abstraction for the component.
  2. Concrete component classes realize this interface type.
  3. Decorator classes also realize this interface type.
  4. A decorator object manages the component object that it decorates.
  5. When implementing a method from the component interface type, the decorator class applies the method to the decorated component and combines the result with the effect of the decoration.



Name in Design Pattern	Actual Name (scroll bars)
Component	Component
ConcreteComponent	JTextArea
Decorator	JScrollPane
method()	A method of Component(e.g. paint)

Scroll Bars

- Scroll bars can be attached to (*decorate*) components.

```
JTextArea area = new JTextArea(10, 25);
JScrollPane scroller = new JScrollPane(area);
```

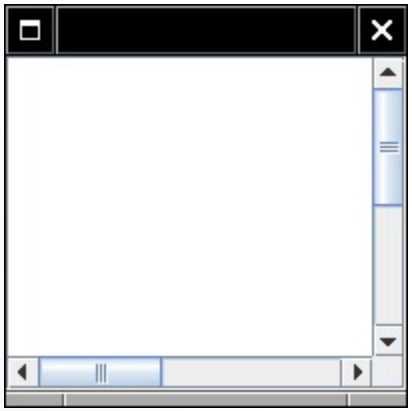


Figure 13: Scroll Bars

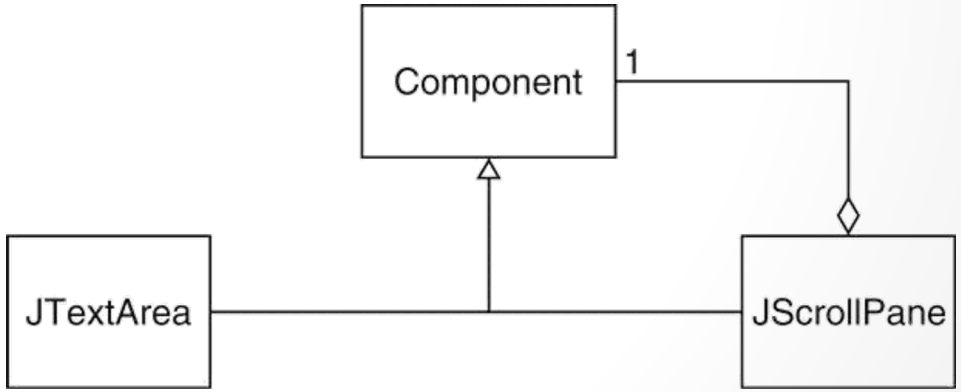


Figure 14: Adding a Scroll Bar to a Text Area

Streams

- The **Reader** class supports basic input operations: reading a single character or an array of characters.
- **FileReader** subclass implements these methods, reading characters from a file.
  - No method for reading a line of input.
- **BufferedReader** takes a Reader and adds buffering.

```
InputStreamReader reader = new InputStreamReader(System.in);
BufferedReader console = new BufferedReader(reader);
```

- Result is another Reader: Decorator pattern.
- Many other decorators in stream library, e.g. PrintWriter.

Decorator Pattern: Input Streams

Name in Design Pattern	Actual Name (input streams)
Component	Reader
ConcreteComponent	InputStreamReader
Decorator	BufferedReader
method()	read

- The **BufferedReader** class is a decorator. It takes an arbitrary reader and yields a reader with additional capabilities.

How to Recognize Patterns

1. Look at the *intent* of the pattern.
   - E.g. COMPOSITE has different intent than DECORATOR.

2. Remember common uses (e.g. STRATEGY for layout managers)
   - Not everything that is strategic is an example of STRATEGY pattern.

3. Use context and solution as "litmus test"

Litmus Test

- Can add border to Swing component

```
Border b = new EtchedBorder()
component.setBorder(b);
```

- Undeniably decorative.
- **Is it an example of DECORATOR?**

Figure 15: Borders Around Panels

Litmus Test

1. Component objects can be decorated (visually or behaviorally enhanced)
   PASS

2. The decorated object can be used in the same way as the undecorated object.
   PASS

3. The component class does not want to take on the responsibility of the decoration.
   FAIL--the component class has setBorder method

4. There may be an open-ended set of possible decorations

Putting Patterns to Work

- invoice.jar
- Invoice contains *line items*.
- Line item has a description and a price.
- Interface type LineItem:

  LineItem.java

- Product is a concrete class that implements this interface:

  Product.java

Bundles

- Bundle = set of related items with description and price.
  - E.g. stereo system with tuner, amplifier, CD player and speakers.
- A bundle has line items.
- A bundle is a line item.

Bundle.java (look at getPrice)

COMPOSITE pattern

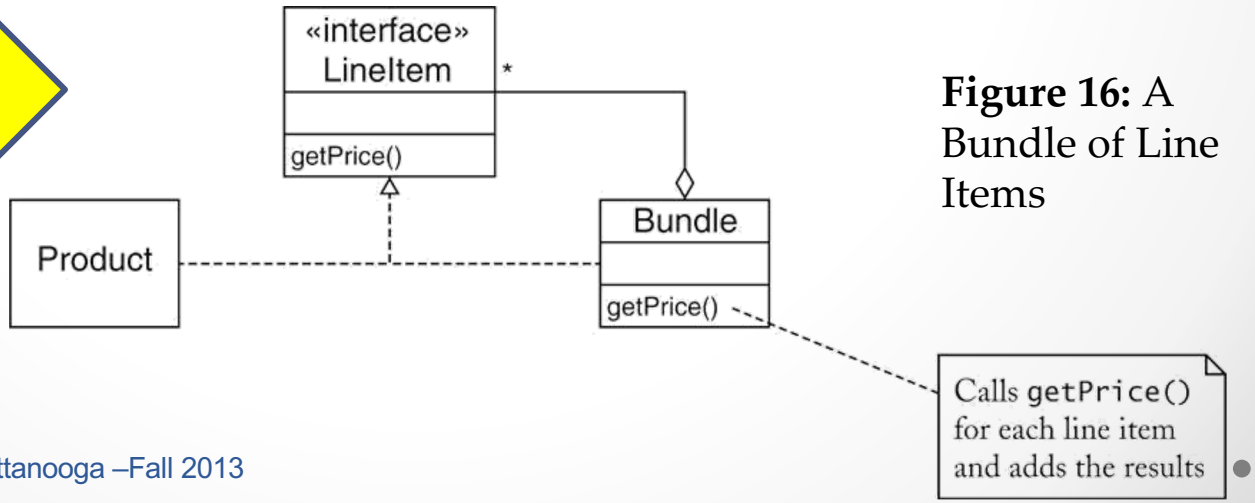


Figure 16: A Bundle of Line Items

Discounted Items

- Store may give discount for an item.
- Discounted item is again an item.

  DiscountedItem.java (look at getPrice)
- Alternative design: add discount to LineItem.

DECORATOR pattern

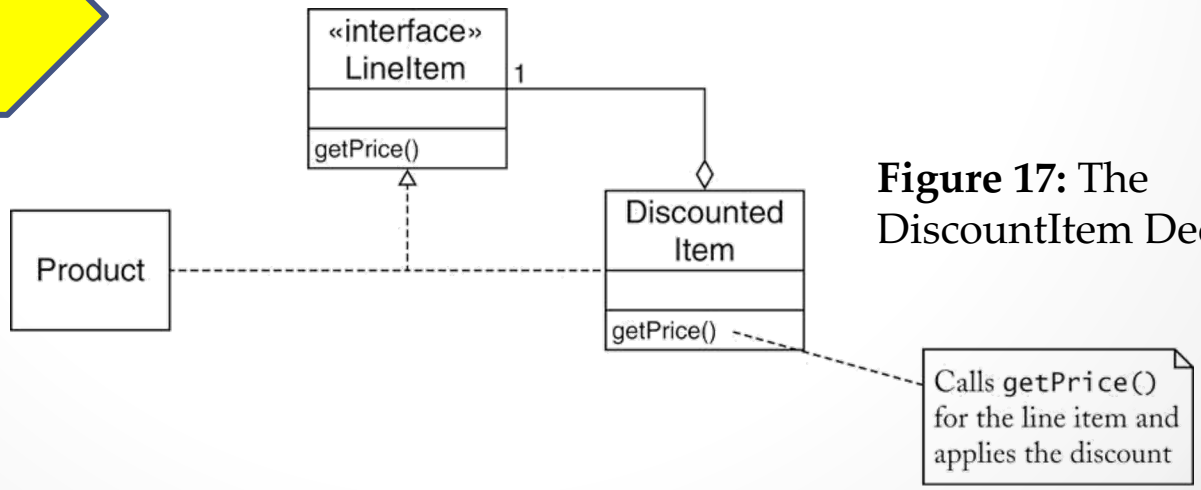


Figure 17: The DiscountItem Decorator

Model/View Separation

- GUI has commands to add items to invoice.
- GUI displays invoice.
- Decouple input from display.
- Display wants to know *when* invoice is modified.
- Display doesn't care which command modified invoice.
- **OBSERVER pattern.**

Model/View Separation

- **OBSERVER pattern:**
  1. Define an observer interface type. Observer classes must implement this interface type.
  2. The subject maintains a collection of observer objects.
  3. The subject class supplies methods for attaching observers.
  4. Whenever an event occurs, the subject notifies all the observers.

Change Listeners

- Use standard ChangeListener interface type

```
public interface ChangeListener
{
    void stateChanged(ChangeEvent event);
}
```

- Invoice collects ArrayList of change listeners.
- When the invoice changes, it notifies all listeners:

```
ChangeEvent event = new ChangeEvent(this);
for (ChangeListener listener : listeners)
    listener.stateChanged(event);
```

OBSERVER pattern

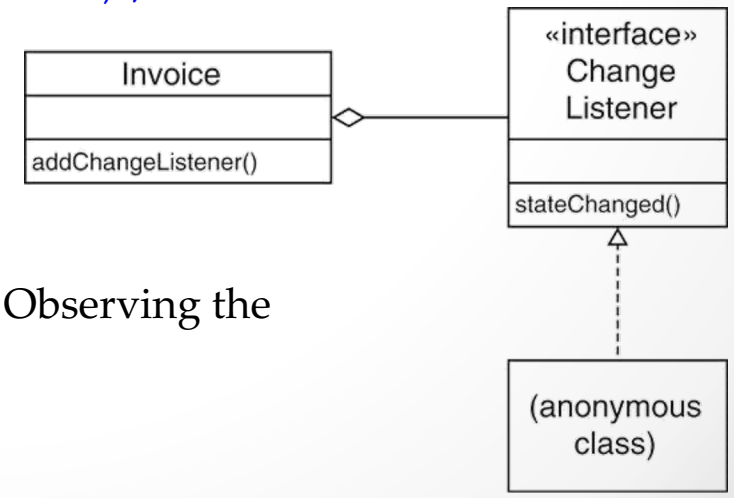


Figure 18: Observing the Invoice

Change Listeners

- Display adds itself as a change listener to the invoice.
- Display updates itself when invoice object changes state.

```
final Invoice invoice = new Invoice();
final JTextArea textArea = new JTextArea(20, 40);
ChangeListener listener = new
ChangeListener()
{
    public void stateChanged(ChangeEvent event)
    {
          textArea.setText(...);
    }
};
invoice.addChangeListener(listener);
```

Iterating Through Invoice Items

- Invoice collect line items.
- Clients need to know the line items inside an invoice.
- Don't want to expose ArrayList.
- May change (e.g. if storing invoices in database).
- **ITERATOR pattern.**

Iterators

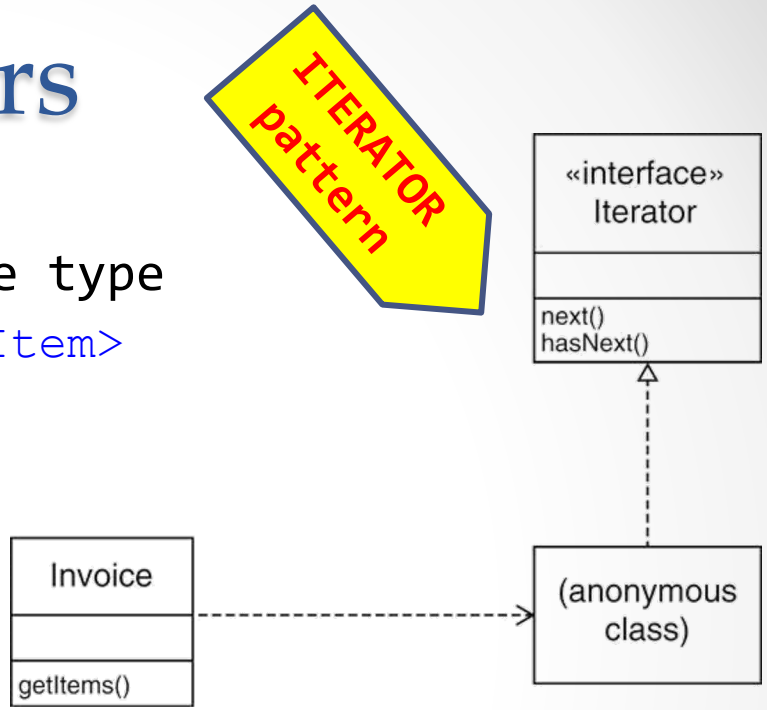


Figure 19: Iterating Through the Items in an Invoice

- Use standard Iterator interface type

```
public interface Iterator<LineItem>
{
    boolean hasNext();
    LineItem next();
    void remove();
}
```

- remove is "optional operation".
- implement to throw **UnsupportedException**

Invoice.java

Formatting Invoices

- Simple format: dump into text area.
  - May not be good enough, HTML tags for display in browser.
- Want to allow for multiple formatting algorithms.
- **STRATEGY pattern**

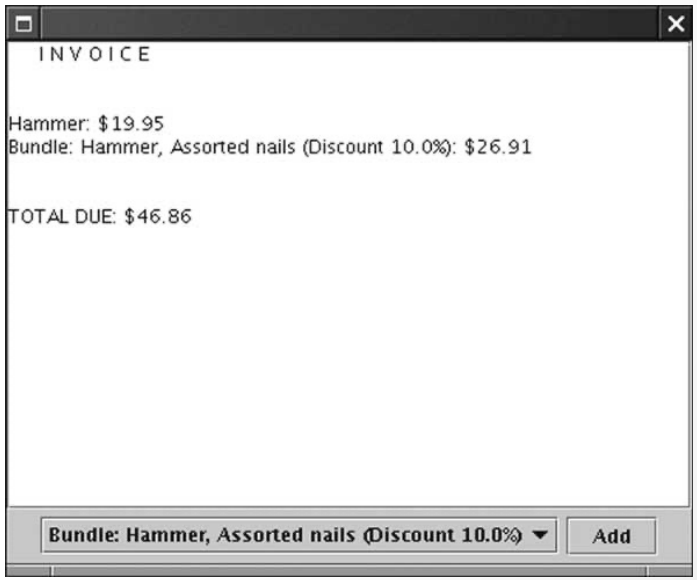


Figure 20: The InvoiceTester Program

Formatting Invoices

STRATEGY pattern

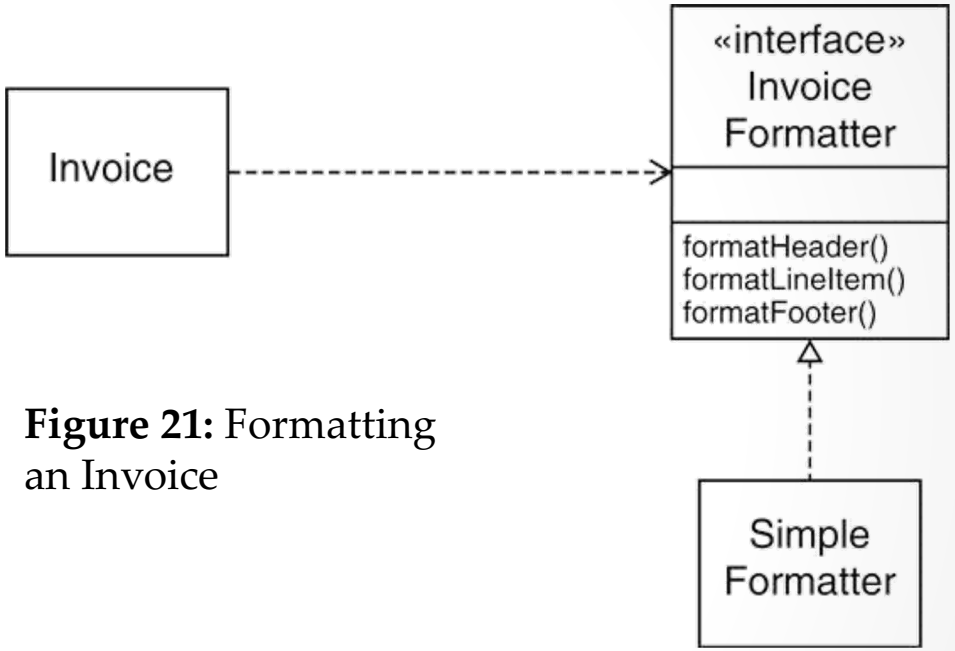


Figure 21: Formatting an Invoice

InvoiceFormatter.java

SimpleFormatter.java

InvoiceTester.java

End of Chapter 5