

## Chapter Topics

- Iterators
- 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

## List Iterators

In Java a list is an ordered collection of elements. Lists allow duplicate elements and null elements. The `List` interface is a specialization of the `Collection` interface. The `List` interface is implemented by: `AbstractList`, `AbstractSequentialList`, `ArrayList`, `AttributeList`, `CopyOnWriteArrayList`, `LinkedList`, `RoleList`, `RoleUnresolvedList`, `Stack`, `Vector`.

A list iterator allows the programmer to examine the elements of a list collection. See: <http://docs.oracle.com/javase/8/docs/api/java/util/List.html> and see also <http://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html> Note that examining is different from changing. Examining does not require mutation operations while changing does. But why have a special iterator construct?

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

## Classical List Data Structure

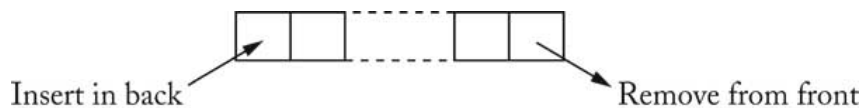
- Traverse links directly

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

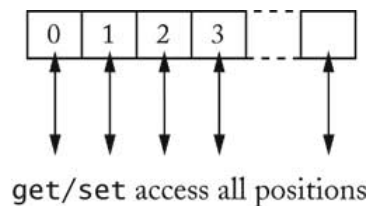
- Exposes implementation. This would allow for unwanted modifications. There is no need for any user of the code to ever modify the code! Once you are confident that an implementation is correct, there is no need for outside parties to have access to the implementation. Note that at some later date the owner of the implementation might replace that implementation with a better one. However, any other party using the original implementation would not notice a difference since the signatures would remain the same. If the implementation was exposed this would not be true.
- Error-prone in use and it allows for unintended modifications.

## High-Level View of Data Structures

- Queue

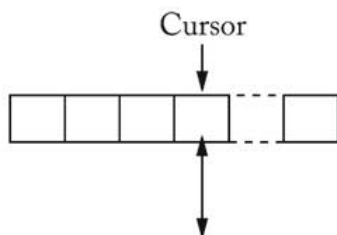


- Array with random access



- List? It does have an order but is not exactly random access.

## List with Cursor



get/set/insert/remove access cursor position

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

- Note that in the code above the constructs relevant to list iteration are used in the defining for loop. Note also there is no hardcoded number of times the loop would execute.

- Disadvantage: Only one cursor per list. Depending on the task at hand might not be a major disadvantage. However, the way to use a list with a cursor might vary across implementations creating a learning curve and increasing the possibility of error. Less learning often produces fewer errors. Or better one might say that the more common the elements and operations, the less specialized learning is required and that produces fewer errors.
- Iterator is a superior concept. It would allow, at least, a common pattern for all lists. This should in turn lower the learning curve and diminish the opportunities for errors.
- This suggests that patterns may be important in the design and implementation of software. So what is a pattern? And now for something totally different. (Actually not relevantly different at all.)

## The Pattern Concept

- History: Architectural Patterns originated by Christopher Alexander. For more on this notion see: [http://en.wikipedia.org/wiki/A\\_Pattern\\_Language](http://en.wikipedia.org/wiki/A_Pattern_Language)
- 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..."

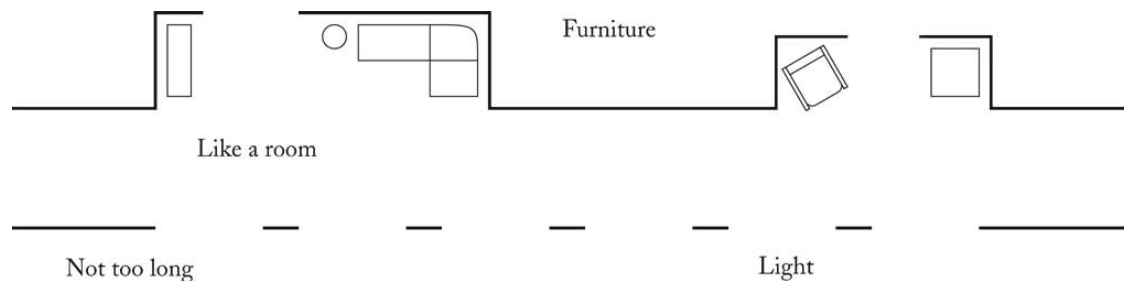
## Problem

A lengthy description of the problem, including

- a depressing picture
- issues of light and furniture
- research about patient anxiety in hospitals
- research that suggests that corridors over 50 ft are considered uncomfortable

## 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.



## Iterator Pattern

Now back to programming and a pattern for an iterator.

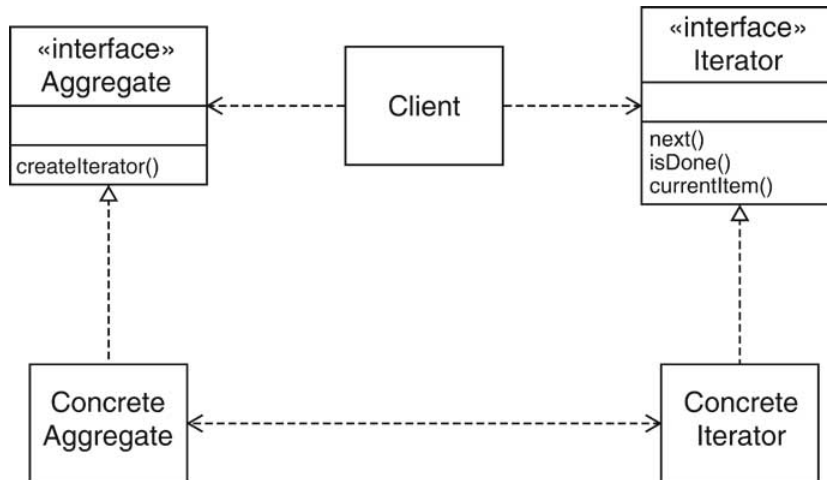
## Context

1. An aggregate object contains element objects
2. Clients need access to the element objects
3. The aggregate object should not expose its internal structure
4. Multiple clients may want independent access

## 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.



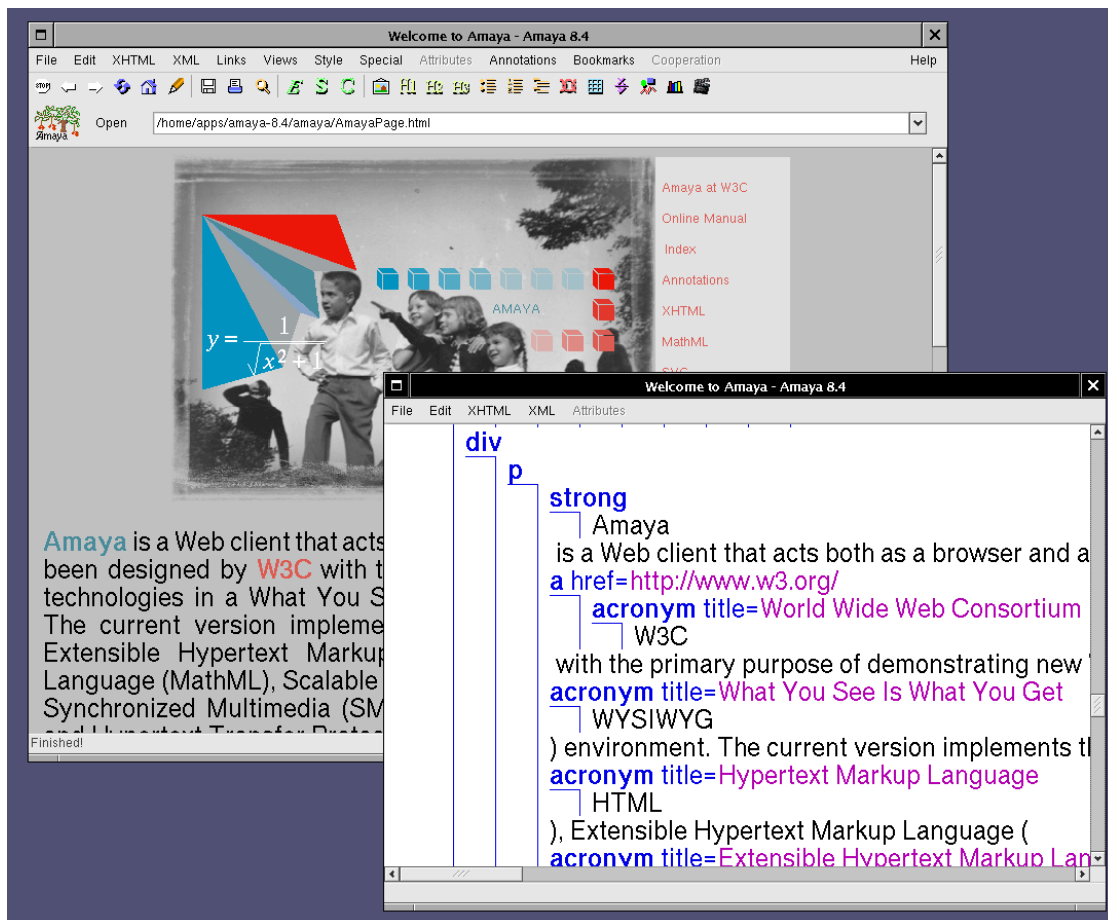
- Names in pattern are *examples*
- Names differ in each occurrence of pattern

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( )

Also see <http://tutorials.jenkov.com/java-generics/implementing-iterable.html> , <https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html> , and the new way with streams <https://www.sitepoint.com/java-8-streams-filter-map-reduce/> , <https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html> , and if you are looking at the 21<sup>st</sup> century stuff <https://www.airpair.com/java/posts/parallel-processing-of-io-based-data-with-java-streams> .

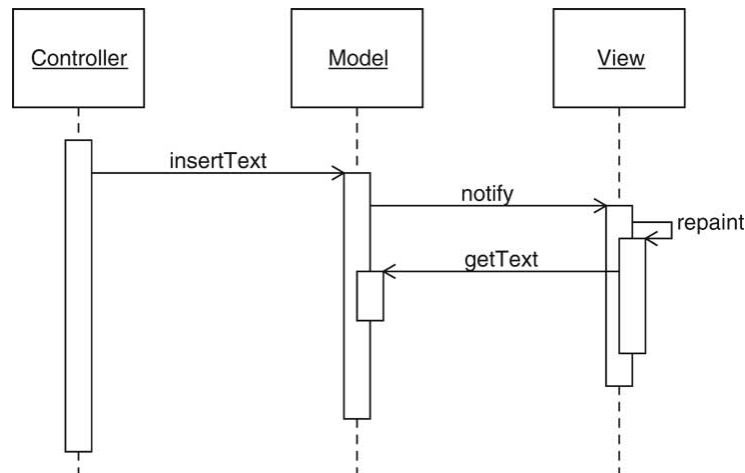
## Model/View/Controller

- This is an essential pattern for building user interfaces. Some programs have multiple editable views. There is a model that can be viewed in different ways and may be altered by different controllers.
- Example: HTML Editor
  - WYSIWYG view
  - structure view
  - source view
- Editing one view updates the other
- Updates seem instantaneous. Note that they seem this way to the human user.



- Model: data structure and logic, no visual representation (The model of the domain.)
- Views: visual representations (These can be various and allow for different visual presentations as well as partial visual presentations)
- Controllers: user interaction (There may be more than one that has the same result.)
- Views/controllers update model
- Model tells views that data has changed (See observer)
- Views redraw themselves

The following item is a bit old but encapsulates the core of MVC in Java  
<http://www.oracle.com/technetwork/articles/javase/index-142890.html> A classic example  
<http://www.cs.utsa.edu/~cs3443/mvc-example.html> And a classic presentation  
<https://ist.berkeley.edu/as-ag/pub/pdf/mvc-seminar.pdf>



## Observer Pattern

The observer pattern is presented rather nicely at [https://sourcemaking.com/design\\_patterns/observer](https://sourcemaking.com/design_patterns/observer) (In fact I would encourage you to use <https://sourcemaking.com/> as a regular reference.) The observer pattern is useful when you want to define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. See: <http://www.journaldev.com/1739/observer-design-pattern-in-java>

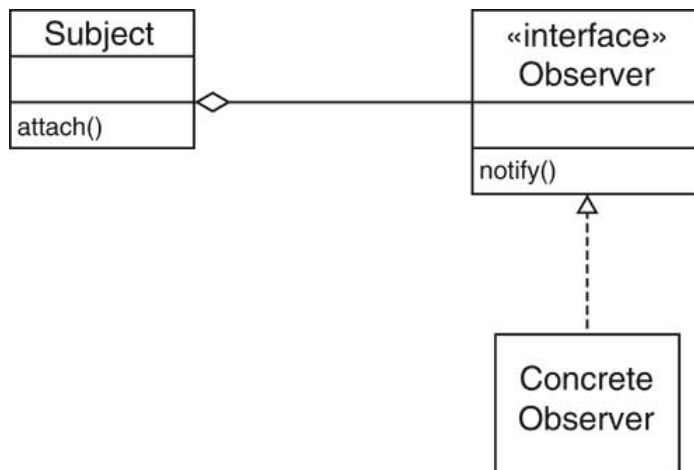
- The observer pattern is an essential pattern for user interfaces. However, that is not the only place to look for this pattern. Look for it wherever in a design some item must be updated if another item changes. A non-gui oriented version can be very useful *in the model*.
- Model notifies views when something interesting happens (The view will need to register its interest.)
- 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*.

## 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.

## Solution

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



## Names in Observer Pattern

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

To get an idea of what layout managers do, see:

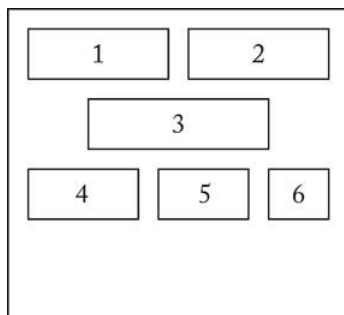
<https://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html>. In brief a layout manager is an object that implements the `LayoutManager` interface

(<https://docs.oracle.com/javase/8/docs/api/java/awt/LayoutManager.html> ) and determines the size and position of the components within a container. Components can provide size and

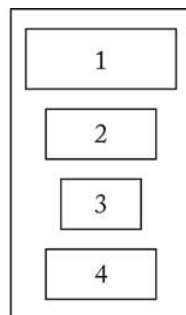


alignment hints, but a container's layout manager ultimately determines the size and position of the components within the container.

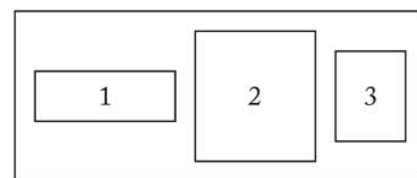
- User interfaces made up of *components* (See: <http://docs.oracle.com/javase/8/docs/api/java/awt/Component.html> )
- Components placed in *containers* (Begin with <http://docs.oracle.com/javase/tutorial/uiswing/components/toplevel.html> )
- Container needs to arrange components
- Swing doesn't require the use of hard-coded pixel coordinates
- Advantages:
  - Can switch "look and feel" (See: <http://docs.oracle.com/javase/tutorial/uiswing/lookandfeel/plaf.html> and <http://www.javootoo.com/> )
  - Can internationalize strings
- Layout manager controls arrangement
- FlowLayout: left to right, start new row when full
- BorderLayout: 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



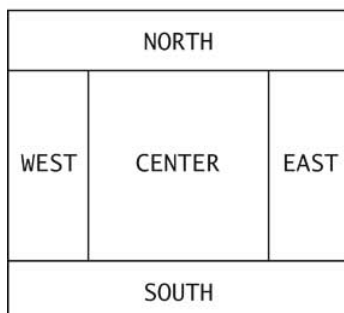
FlowLayout



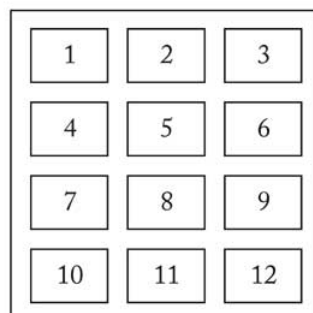
BoxLayout (vertical)



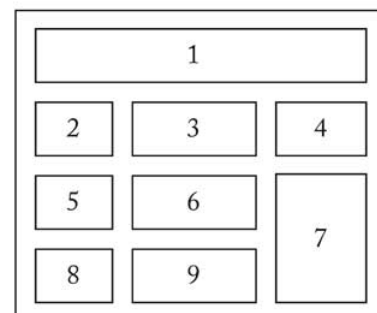
BoxLayout (horizontal)



BorderLayout



GridLayout



GridBagLayout

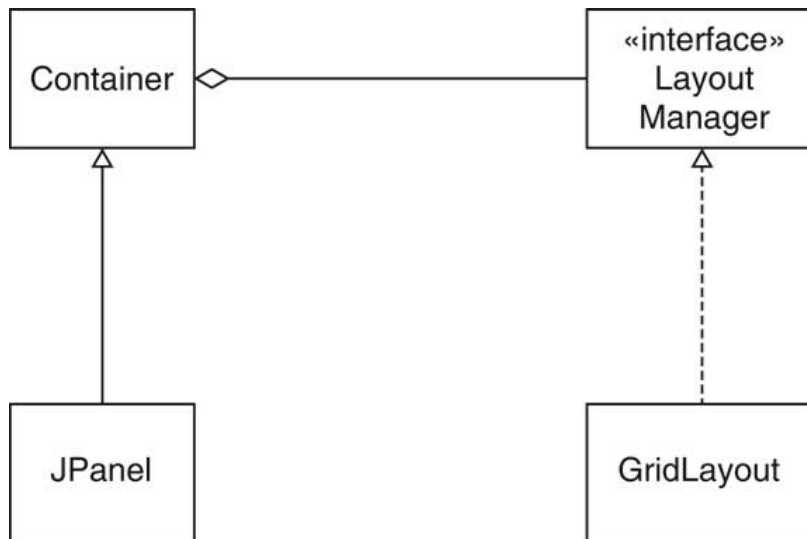
- The layout managers can be used in the NetBeans IDE (by default it is freeform <https://netbeans.org/kb/docs/java/quickstart-gui.html> ; you might also check

<https://www.youtube.com/channel/UCLuXaf-zBsw8Lzz6nqr66Jw> ) or can be set programmatically.

- Set layout manager

```
JPanel keyPanel = new JPanel();
keyPanel.setLayout(new GridLayout(4, 3));
```
- Add components

```
for (int i = 0; i < 12; i++)
    keyPanel.add(button[i]);
```



### Voice Mail System GUI

- Same backend as text-based system
- Only Telephone class changes
- Buttons for keypad
- Text areas for microphone, speaker



Speaker:		
You have reached mailbox 12. Please leave a message now.		
1	2	3
4	5	6
7	8	9
*	0	#
Microphone:		
Hello Fifi! This is Aramis. Are we still on for lunch today? Please call me back. Thanks!		
Send speech		Hangup

- Arrange keys in panel with GridLayout:

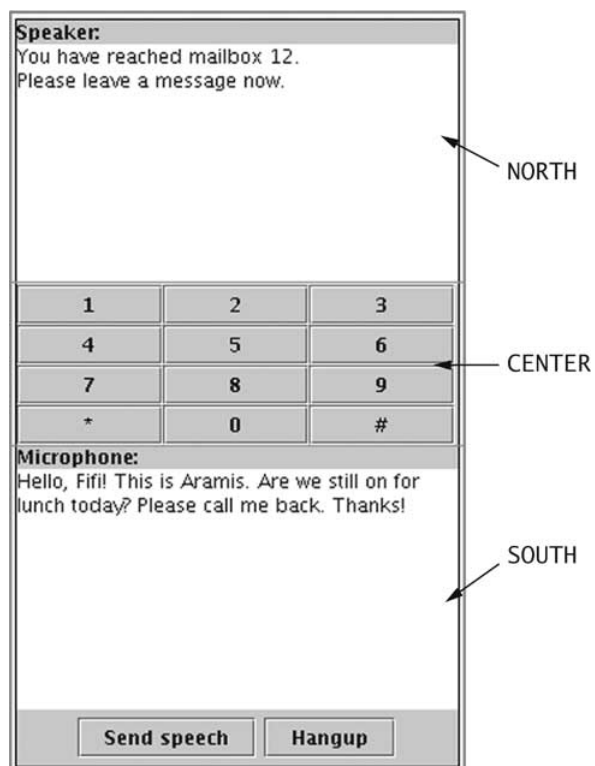
```
JPanel keyPanel = new JPanel();
keyPanel.setLayout(new GridLayout(4, 3));
for (int i = 0; i < 12; i++)
{
    JButton keyButton = new JButton(...);
    keyPanel.add(keyButton);
    keyButton.addActionListener(...);
}
```

- Panel with BorderLayout for speaker

```
JPanel speakerPanel = new JPanel();
speakerPanel.setLayout(new BorderLayout());
speakerPanel.add(new JLabel("Speaker:"), BorderLayout.NORTH);
speakerField = new JTextArea(10, 25);
speakerPanel.add(speakerField, BorderLayout.CENTER);
```

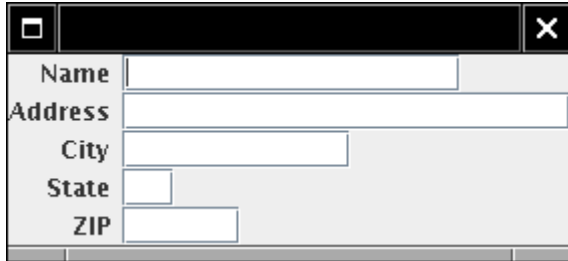


- Add speaker, keypads, and microphone panel to frame
- Frame already has BorderLayout



## Custom Layout Manager

- Make your own *Form* layout
- Odd-numbered components right aligned
- Even-numbered components left aligned
- Implement `LayoutManager` interface type



## 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);
}
```

## Form Layout

- See text for sample code for the form layout manager.
- Note: Can use `GridBagLayout` to achieve the same effect

## Strategy Pattern

The strategy pattern allows for a class behavior or its algorithm to be changed at run time. In Strategy pattern, we create objects which represent various strategies and a context object whose behavior varies as per its strategy object. The strategy object changes the executing algorithm of the context object. See: <http://www.journaldev.com/1754/strategy-design-pattern-in-java-example-tutorial> and [https://sourcemaking.com/design\\_patterns/strategy](https://sourcemaking.com/design_patterns/strategy) .

- Pluggable strategy for layout management but not restricted to this. It is a general and versatile pattern
- Layout manager object responsible for executing concrete strategy
- Generalizes to Strategy Design Pattern
- Other manifestation: Comparators

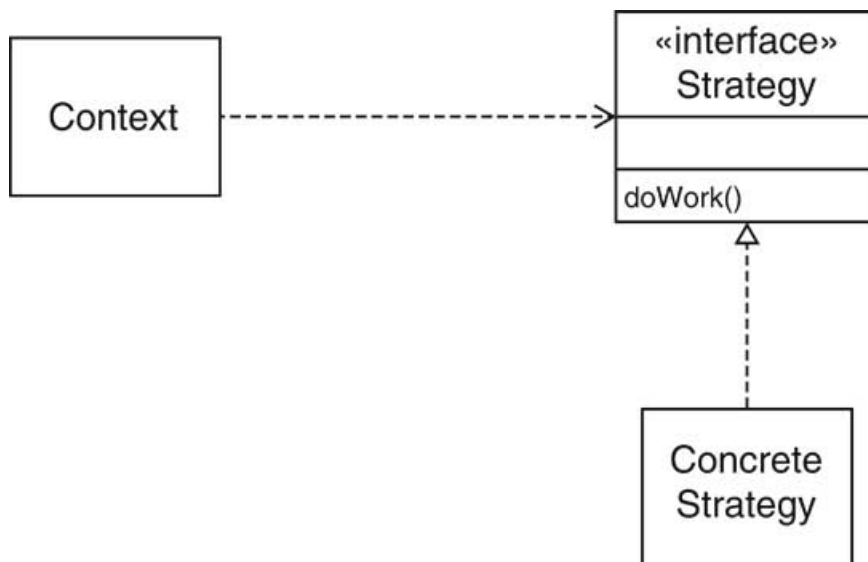
```
Comparator<Country> comp = new CountryComparatorByName();
Collections.sort(countries, comp);
```

## Context

1. A class can benefit from different variants for an algorithm
2. Clients sometimes want to replace standard algorithms with custom versions

## 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



## Strategy Pattern: Layout Management

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

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

## Containers and Components

There is really no way to have a good understanding of this topic without looking at the variety of the items. The best way to do that is to go through the tutorial <https://docs.oracle.com/javase/tutorial/uiswing/components/index.html>. If you really like the visual approach try <http://web.mit.edu/6.005/www/sp14/psets/ps4/java-6-tutorial/components.html>

- Containers collect GUI components
- A container can be added to another container
- Container should *be* a component
- Composite design pattern
- Composite method typically invoke component methods
- E.g. `Container.getPreferredSize` invokes `getPreferredSize` of components

## Composite Pattern

The composite pattern is used where you need to treat a group of objects in a similar way as a single object. Composite pattern composes objects in term of a tree structure to represent a part - whole hierarchy. See [https://sourcemaking.com/design\\_patterns/composite](https://sourcemaking.com/design_patterns/composite)

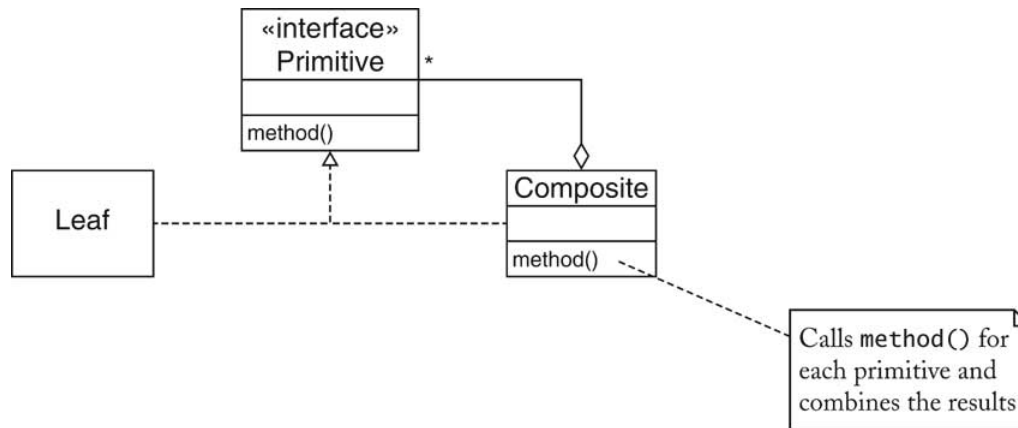
### Context

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

### 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)

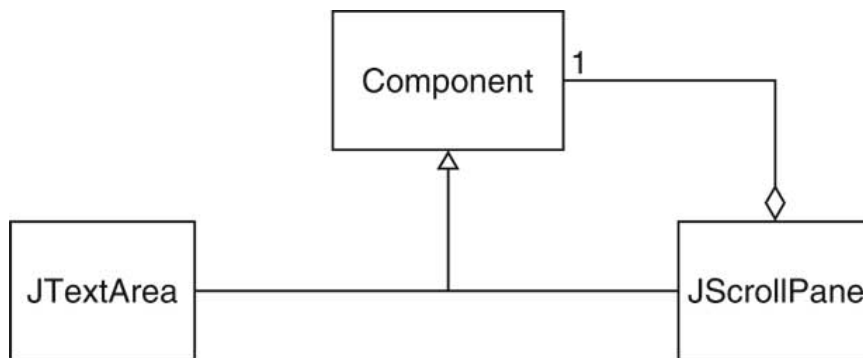
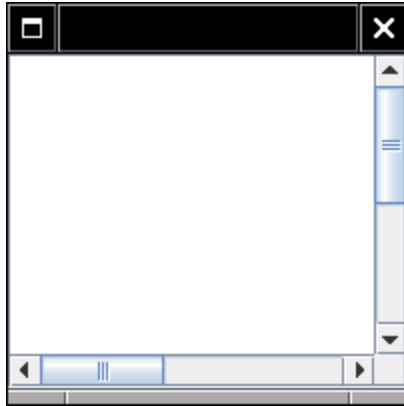
## Scroll Bars

- Scroll bars can be attached to components
- Approach #1: Component class can turn on scroll bars
- Approach #2: Scroll bars can surround component

```
JScrollPane pane = new JScrollPane(component);
```

- Swing uses approach #2
- JScrollPane is again a component





## Decorator Pattern

Decorator pattern allows a user to add new functionality to an existing object without altering its structure. This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact. See:

[https://sourcemaking.com/design\\_patterns/decorator](https://sourcemaking.com/design_patterns/decorator)

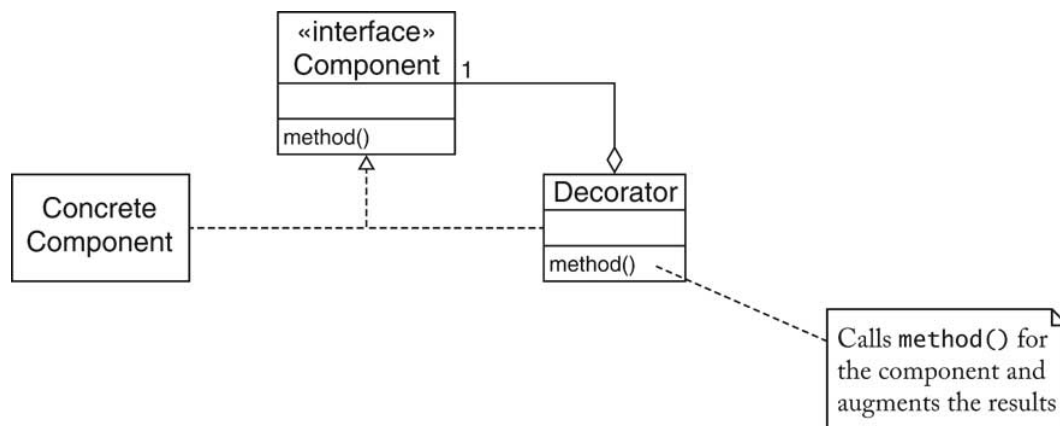
## Context

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

## 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

- 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.



### Decorator Pattern: Scroll Bars

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

### File Streams and related items

```

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

- `BufferedReader` takes a `Reader` and adds buffering
- 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	<code>Reader</code>
ConcreteComponent	<code>InputStreamReader</code>
Decorator	<code>BufferedReader</code>
method( )	<code>read</code>

## How to Recognize Patterns

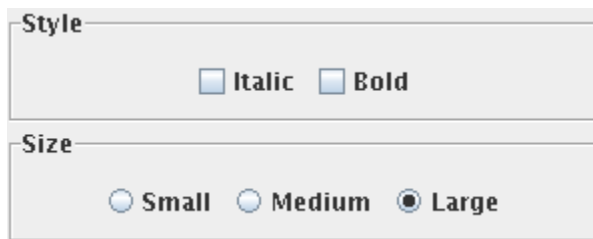
- Look at the *intent* of the pattern
- E.g. COMPOSITE has different intent than DECORATOR
- Remember common uses (e.g. STRATEGY for layout managers)
- Not everything that is strategic is an example of STRATEGY pattern
- Use context and solution as “litmus test”

## Litmus Test

- Can add border to Swing component

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

- Undeniably decorative
- Is it an example of DECORATOR?



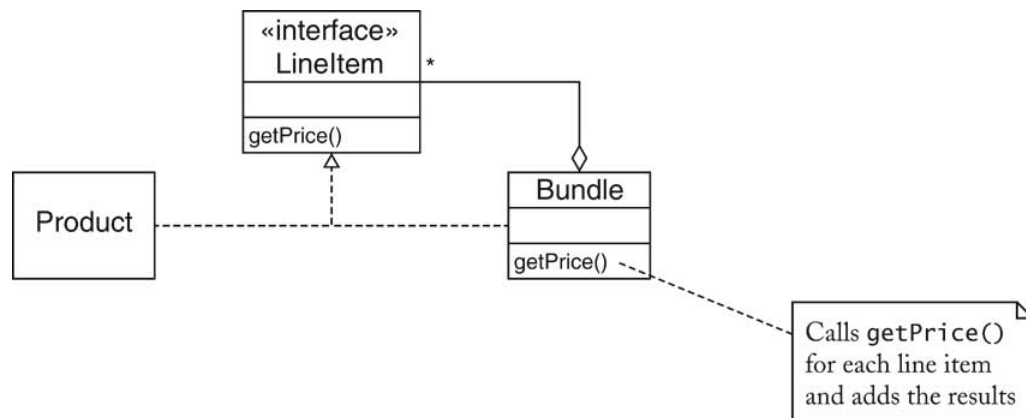
1. Component objects can be decorated (visually or behaviorally enhanced)  
**PASS**
2. The decorated object can be used in the same way as the undecorated object  
**PASS**
3. The component class does not want to take on the responsibility of the decoration  
**FAIL--the component class has `setBorder`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

## Putting Patterns to Work

- Invoice contains *line items*
- Line item has description, price
- Interface type `LineItem`: see sample code in text
- `Product` is a concrete class that implements this interface: see sample code in text

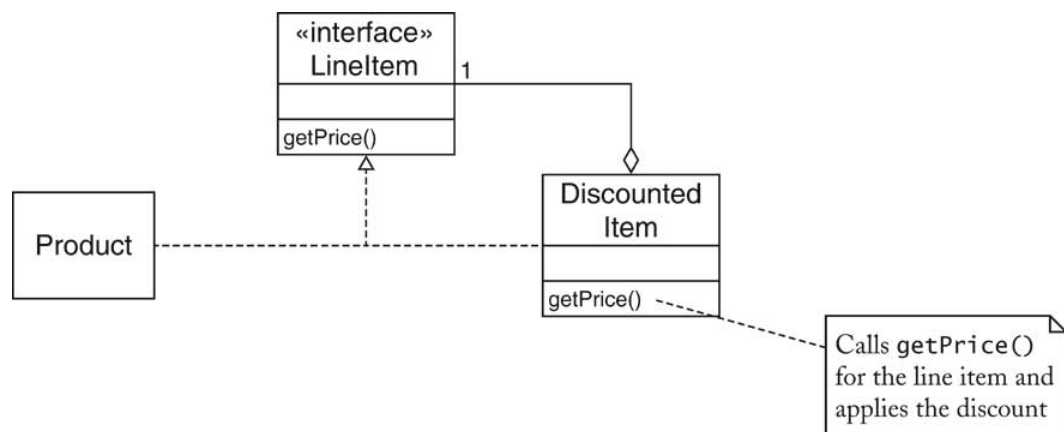
## Bundles

- Bundle is a set of related items with description+price
- E.g. stereo system with tuner, amplifier, CD player + speakers
- A bundle has line items
- A bundle is a line item
- COMPOSITE pattern
- Look at `getPrice` in the sample code in the text



## Discounted Items

- Store may give discount for an item
- Discounted item is again an item
- DECORATOR pattern
- Look at `getPrice` in the sample code in the text
- Alternative design: add discount to `LineItem`



## 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

## Change Listeners

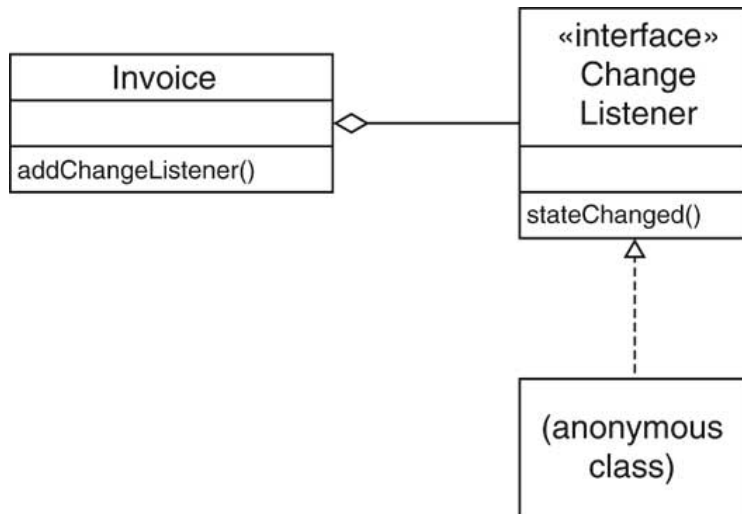
- Use standard `ChangeListener` interface type

```
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);
```
- 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(...);
        }
    };
};
```

## Observing the Invoice



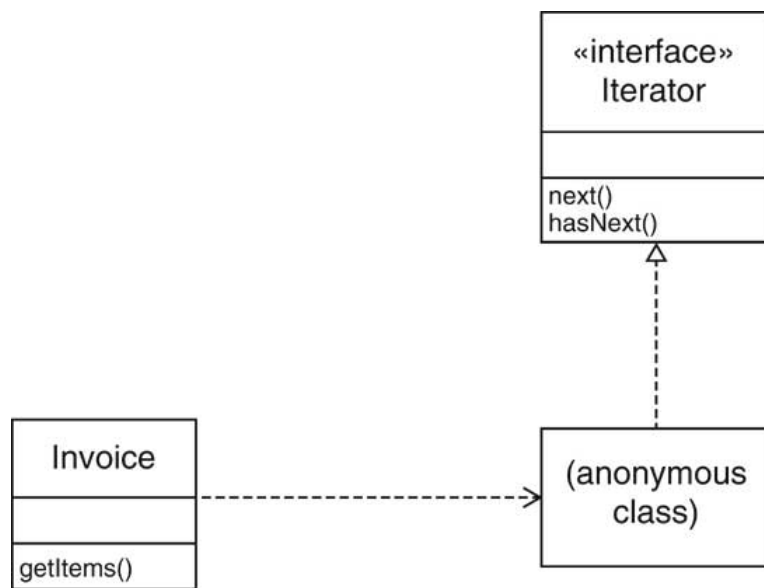
## Iterating Through Invoice Items

- Invoice collect line items
- Clients need to iterate over line items
- Don't want to expose `ArrayList`
- May change (e.g. if storing invoices in database)
- ITERATOR pattern

## Iterators

- Use standard `Iterator` interface type

```
public interface Iterator<LineItem>
{
    boolean hasNext();
    LineItem next();
    void remove();
}
```
- `remove` is “optional operation” (see ch. 8)
- implement to throw `UnsupportedException` (is part of contract)
- implement `hasNext`/`next` manually to show inner workings
- See implementing code in text



## Formatting Invoices

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



