Object Oriented Design & Patterns

Part 1
Design Patterns

• Derived from architectural patterns:
  – rules for design of buildings
  – describe common problems, solutions to those problems

• OO design patterns
  – convenient way of reusing OO code between projects, programmers
  – catalog common interactions between objects that programmers have found useful
Design Patterns

• Each pattern includes:
  – description of problem context
  – prescription for solution

• Idea is to distill a design rule into a simple format
Example: Iterators

• An iterator is a method or set of methods applied to an aggregate data structure (such as a linked list)
• Iterator provides access to structure members without requiring knowledge of internal structure details
Iterator Pattern: Context

- An aggregate object contains element objects
- Clients need access to the element objects
- The aggregate object should not expose its internal structure
- Multiple clients may want independent access
 Iterator Pattern: Solution

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

![Diagram showing the Iterator Pattern]

- **Aggregate**
  - `createIterator()`
- **Client**
  - References **Iterator**
- **Iterator**
  - `next()`, `isDone()`, `currentItem()`
- **Concrete Aggregate**
  - Links to **Aggregate**
- **Concrete Iterator**
  - Linked to **Iterator**
Iterator pattern example: linked list

Design pattern name:  
  Aggregate  
  ConcreteAggregate  
  Iterator  
  ConcreteIterator  
  createIterator()  
  next()  
  isDone()  
  currentItem()  

Actual name:  
  List  
  LinkedList  
  ListIterator  
  anonymous class implementing ListIterator  
  listIterator()  
  next()  
  opposite of hasNext()  
  return value of hasNext()
Model/View/Controller architecture

• Describes interaction of objects in a user interface with multiple editable views

• Example: PowerPoint
  – outline view
  – slide view
  – notes view
  – slide sorter view

• Edit in one view updates another; seems instantaneous
Model/View/Controller architecture

• Model: data structure, no visual representation (example: array)

• Views: visual representations
  – drawn using specific format
  – for example, number table Vs. bar chart

• Controllers: user interaction
Model/View/Controller architecture

• Each aspect has discrete responsibilities:
  – Views/controllers update model
  – Model tells views that data has changed
  – Views redraw themselves
Model/View/Controller architecture
Model/View/Controller architecture

• Minimizes coupling between models, views & controls
  – model has no knowledge of views; just knows to notify them of change
  – views know nothing about controllers
  – easy to add more views to model
  – easy to change controller of a view

• Views are example of Observer pattern
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
  – An object, called the subject, is source of events
  – One or more observer objects want to be notified when such an event occurs.
Observer Pattern

• Solution
  – Define an observer interface type. All concrete observers implement it.
  – The subject maintains a collection of observers.
  – The subject supplies methods for attaching and detaching observers.
  – Whenever an event occurs, the subject notifies all observers.
Observer Pattern

Subject

attach()

«interface» Observer

notify()

Concrete Observer
Observer pattern example: Swing buttons

Pattern name:
- Subject
- Observer
- ConcreteObserver
- attach()
- notify()

Actual name:
- JButton
- ActionListener
- ActionListener implementor
- addActionListener()
- actionPerformed()
Layout Managers

• User interfaces made up of components
• Components placed in containers; need to arrange components
• Swing doesn't use hard-coded pixel coordinates
• Advantages:
  – Can switch "look and feel"
  – Can internationalize strings
• Layout manager controls arrangement
Standard Layout Managers: FlowLayout

Components are laid out left to right; when a row is full, a new row is started
Standard Layout Managers: BoxLayout

Components are laid out left to right or top to bottom

BoxLayout (vertical)

BoxLayout (horizontal)
Standard Layout Managers: BorderLayout

• Default layout manager for JFrames
• Defines 5 areas: Center, North, South, East, West
  – Don’t have to place a component in every area
  – Areas are sized according to need of component
Standard Layout Managers: BorderLayout
Standard Layout Managers: GridLayout

Components laid out in grid, all same size
Standard Layout Managers: GridBagLayout

- Complex set of cells contain components
- Similar to HTML table
Using Layout Managers

• Create container
  Panel p = new Panel();

• Set layout manager
  p.setLayout(new GridLayout(4, 4, 3, 3));

• Add components
  p.add(darken);
Using Layout Managers