CS151 Midterm #1

Three Stages of Software Development

1) Analysis
2) Design
3) Implementation/Testing

Common Development Processes

1) **Waterfall** - All analysis, then all design then all implementation.
   - Three separate, non repeated stages.
2) **Iterative** - Analysis, design, and implementation repeated in a loop.
3) **Agile** - All stages are extremely short and a working version of the program always exists.

Requirements Analysis

Requirements Analysis - A requirements spec should have the following properties

1) Completely defines all tasks to be solved
2) Free from internal contradictions.
3) Readable by laypeople and developers
4) Testable against reality

Design Stage:

Goal - Produce descriptions and diagrams that show classes and their relationships.

1) Identify classes
2) Identify class responsibilities
3) Identify relationship between classes

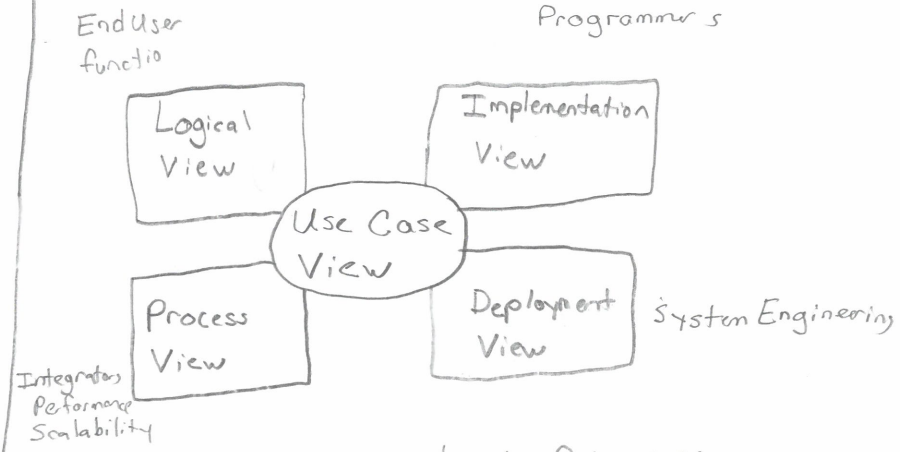
Result

1) Textual Description of classes and responsibilities
2) Diagrams of class relationships.
3) Diagrams of usage situations
4) State diagrams for objects with rich state

- UML Class Diagrams
- CRC cards

Kruchten's 4+1

A model used to describe a software system from the perspective of different stake holders.



Logical View - Classes and parts of the program

Process View - Concurrency and synchronization aspects of the design

Implementation View - Static organization of the software in its development environment.

Deployment View - Mapping of the software onto the hardware and the program's distributed aspects.

Implementation and Testing

Proceeds in Individual Units

1) Unit coding
2) Unit testing
3) Integration Testing

Goal of this stage: A usable software package.

Identifying Classes

Classes correspond to things/nouns.

Five Class Categories

1) **Events and Transactions** – Used to retain information about what has happened or what will happen.
2) **User and role** – Stand in for actual users of a program. Example "Administrator" or "Reviewer" class.
3) **System Class** – Model of a system or subsystem. Roles include initialization, shutdown, start of flow, etc.
4) **System interface class** – Used to model interaction with the operating system, a database, etc.
5) **Foundation Class** – Encapsulates basic data types (Date, String, Rectangle)

Relationships between classes

Three most common relationships between classes.

1) **Dependency** ("uses")
2) **Aggregation** ("has")
3) **Inheritance** ("is")

Dependency – A class depends on another class if it manipulates objects of the other class.

- **A check for dependency** – Can one class operate without knowing the other class exists?
- Increased dependency, increased coupling.

Aggregation – One class contains objects of another class.

- Keyword for aggregation relationship: "**has a**"
- Can be determined by the instance fields of the class.
- For simple classes (number, date, string), aggregation is not used to describe the relationship.

Inheritance – One class inherits from another if all objects of its class are special cases of the objects of other classes.

- **Superclass** – More generalized class
- **Subclass** – more specialized class
- Relationship keywords: "**is a**"

Software Metrics

Coupling – Dependency Relationship. One piece of code relies on another piece of code.
- High coupling means if one piece of code changes, another will also need to change.

Cohesion – Similar to encapsulation.

- High cohesion means that all code relating to a specific thing or task is located in one place.

CS151 - Modeling Software

Three Design Representation Schemes
1) Use Cases
2) Design
3) UML

Why not simply write text? Text can be verbose and ambiguous. A description leads to questions which lead to more text and in turn more questions.

Why not use code? Code is intimidating and meaningless to managers, customers, and executives who may need to examine the design.

UML - Unified Modeling Language
- Used to make software more analyzable and understandable by lay people and programmers.

Types of UML Diagrams
1) Use Case
2) Activity
3) Class
4) Object
5) Sequence
6) Package
7) StateMachine

Use Cases
- Used as an **analysis** technique.
- A use case specifies user input and program output.
  - Output must be of some value to the user.
- Allow domain-specific knowledge to flow from designers to developers.
- Software QA can confirm or deny whether the software meets the requirements of the use cases.
- Use cases are very easy to create and intuitive for everyone to use.

CRC Cards - Class, Responsibilities and Collaborators Cards

Class Name	
Responsibilities	Collaborators

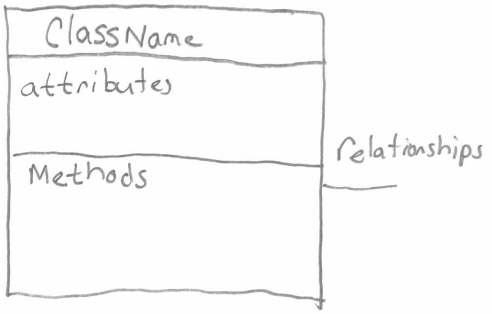
Typically written on 3x5 index cards.

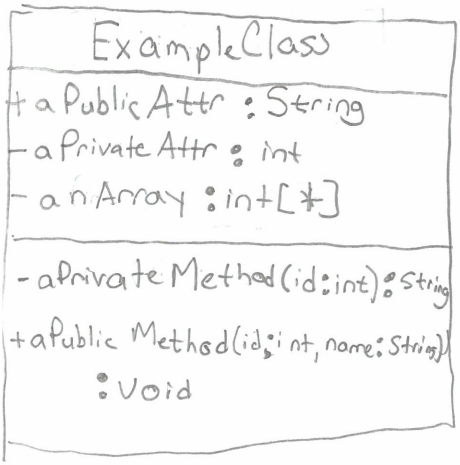
CRC Card Process - Usually Done Iteratively
1) Identify classes
2) Identify responsibilities
3) Define collaborators
4) Move collaborators close together
5) Walk through use cases.

CRC Card Tips
1) Responsibilities are not methods.
2) Usually 1-3 responsibilities per class
3) Collaborators are unordered

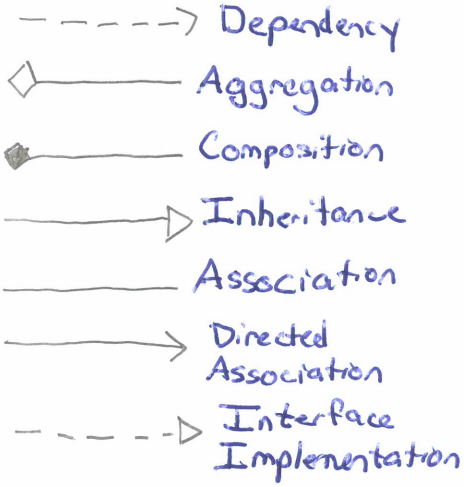
UML Class Diagram

- Lays out parts of a class and its relationship to other classes





Class Relationships



Dependency relationship – One class uses another at some point either as a parameter variable or as a local variable.

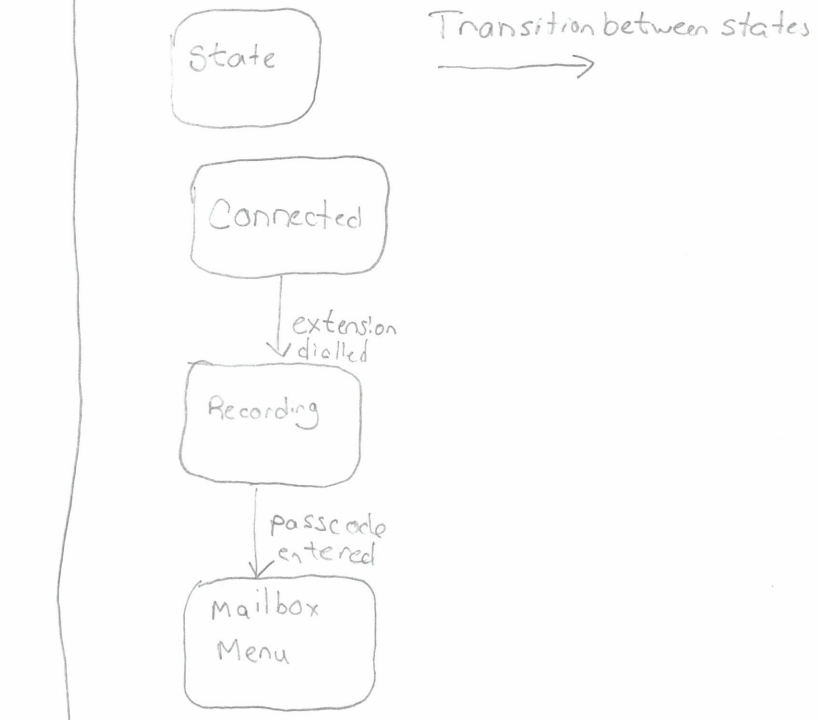
Association Relationship – A relationship between two objects where each has its own lifecycle and there is no owner.

Aggregation – A relationship between two objects where each has its own lifetime but there is ownership.

Composition – Specialized Form of aggregation. A child object does not have a lifecycle without a parent object.

UML State Diagrams

- Depicts the state of an object and the transition between states.



UML Sequence Diagrams

- Show the dynamics of a particular scenario. They show the interaction between objects (not classes).

Writing Object Names

1) objectName : ClassName
2) objectName
3) :ClassName

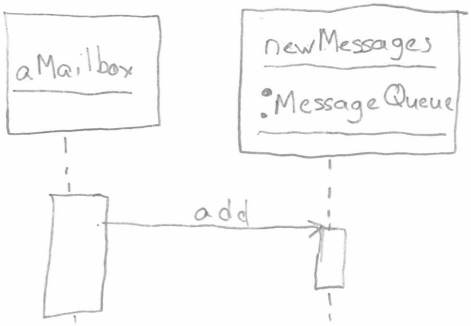
Names must be underlined

Lifeline – Dashed vertical line emanating from an object.

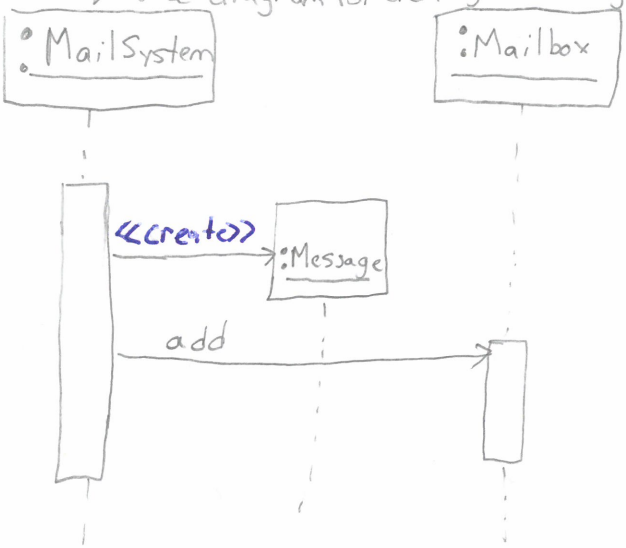
Activation Bar – Rectangles along the lifeline that show when an object is executing a method. An activation bar ends when a method returns.

<<Create>> notation to indicate the creation of an object.

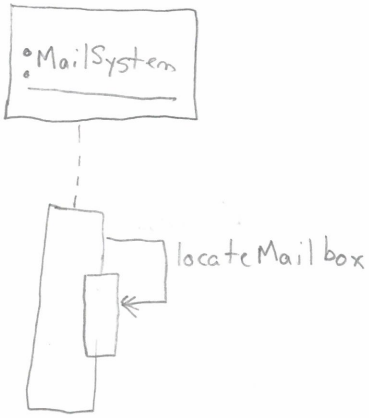
Arrows – Accompanied by text and the text represents a methodName.



Sequence diagram for creating then adding a Message



Selfcall



Designing Classes - Encapsulation

Encapsulation – How data and methods are segregated into classes. A method for restricting access to some of an object's components.

Three primary ways to write a class:

1) Naive (i.e. from scratch) approach
2) Inheritance
3) Interface implementation.

Accessor Method – Reads and returns a class' instance fields.

Mutator Method – Changes the state of an object.

Immutable Class – Any class with no mutator methods. These classes make it safe to pass references around.

There should not be accessors and mutators for all of a class' attributes.

Side Effect – Any data modification that is observable when a method is called.

Example: Changing the state of an object

Example: Printing to system.out.

Law of Demeter – A method should only use:

1) Instance fields of its class
2) Function parameters
3) Objects it creates new.

Protecting an Object's State

- An accessor should never return a reference to a mutable field. Otherwise, a user may invalidate the state of the object.

Exception: Safe to return immutable or primitive (e.g. int, double) fields.

Solution: Clone (`.clone()`) the object before returning it. In Java if the attribute is a reference, it performs a **shallow copy** and only copies the reference address. You should perform a **deep copy** and duplicate all data.

Final Instance Fields: For fields that will not change, you can mark them final. This is a way to ensure immutable objects are not mutated.

Keep Accessors and Mutators Separate: Accessors should never mutate an object. Mutators should return void.

Side Effects and Accessors: Accessors should not have side effects.

Side Effects and Mutators: All mutators have side effects.

Private Methods – Helper methods should always be private.

- Helper methods may change if implementation changes.
- Helper methods may require special protocols or calling orders the programmer may not know.
- Clutters the interface

General Rule for Methods and Attributes

- Once public, always public.

Decoupling with an Interface

- By linking two objects through an interface, the implementation of one object can change without affecting the second object.

Decoupling through Inheritance

- Define an abstract class and update a set of select methods based on a specific need.

Date class – Time in milliseconds since 1/1/1970 at midnight GMT.

Calendar class Depends on the Date class and is seperate from a **Gregorian Calendar** class since not everyone follows the Gregorian Calendar.

Software Quality Metrics – Class Interface Measures

Cohesion – A class operations must fit together to support a single coherent purpose.

- Aggregation can be a danger to cohesion.
- Operations that do not fit into a class' framework should be assigned to another class.

Convenience – The interface should make common tasks easy.

- One method call should encapsulate a single operation.
- A good interface makes all operations possible and common operations easy.

Consistency – The operations in a class should be consistent with each other with respect to names, parameters, and return values.

- **Example:** If method names are camelCase with no underscore, then all method names should be that way.

Completeness – A class should support all operations that are part of the abstraction that class represents.

- Somewhat open to interpretation and is subjective.

Clarity – Interface should be clear to programmers without generating confusion even with minimal documentation.

- Classes, methods, and parameters should be well named and their purpose is clear.

Class invariant – Logical condition for a class that holds for all objects of a class except when it is being mutated.

- Invariants are true after an object has been created even if it is uninitialized.

- **Interface Invariants** – condition involving only the public interface of a class.

Implementation Invariants – Details of a particular implementation to ensure correctness.

Interface Invariant must use class interface methods in definition.

Example: `1<=getMonth() && 12>=getMonth()`.

Programming By Contract

- **Precondition** (javadoc `@precondition`) – A condition that must be fulfilled before a method is called.
- **Postcondition** (javadoc `@postcondition`) – A condition the service provider guarantees upon completion.

Assertion – When a function requires a precondition Call an assert at the beginning of the function to ensure it is fulfilled.

Java Assert Syntax:

```
assert condition:"Message";
```

By default assertions are disabled in Java. They must be enabled at runtime:

```
java -enablesessions ClassName.
```

Exceptions (javadoc `@throws`) – A way to deal with problematic code.

Note: assertions can be turned off while exceptions cannot.

Interface Types and Polymorphism

Inteface – Set of operations a class can perform.

Interface Type –
- No implementation information.
- All methods **must** be public.
- Multiple classes can implement the same interface.

Java implementation of an Interface

```
public interface Animal {
    public void eat();
    public void travel();
}
```

Polymorphism – The ability of a program to call different methods based on an object's actual type.
- If a class implements an interface type, it can be assigned to variables of that interface type.
- Promotes **loose coupling**
- Enables **extensibility** as new versions of an object can be developed later.

Comparable Interface

- Collections.sort can sort any list that contains objects that implement the Comparable interface.
- Library for Collections.sort
  java.util.collections
- Interface

```
public interface Comparable<T> {
    public int compareTo(T other);
}
```

Rules of Return for CompareTo

object1.compareTo(object2)

- **Negative number** – object 1 should appear **before** object 2.
- **Zero** – object 1 and object 2 are interchangable in order.
- **Positive Number** – object 1 should be **after** object 2.

Note: Comparable is implemented on the object's class not only the list containing it.

Comparator Interface

- More flexible than comparator since you can have multiple sorting algorithms.

Interface

```
public interface Comparator<T> {
    public int compare(T first, T second)
}
```

- Same return rules as the Comparable interface.
- Using Comparator

```
Collections.sort(list, comp);
```

- where comp implements the comparator interface for the type of objects in the list.

Anonymous Object – An object not stored in a variable.
- **Example:** Any string used outside a variable is an anonymous object.

Anonymous Class – A class without a name. When an anonymous class is used, you must also declare a variable for that class at the same time.

Inner Class – A class defined within another class.

Example of an Anonymous Class

```
Comparator<T> comp = new Comparator<T>() {
        public void compare(T object1, T object2) {
            ...
            return comparison;
        }
};  // -- Need semicolon at the end.
```

AWT - Abstract Windowing toolkit

Java Library
`java.awt.*;`

- AWT was the Java's first attempt at cross platform graphic
- In J2SE 1.2, AWT was superseded by Swing.
- Mostly used today for low level classes.

Event
ActionListener
BasicStroke

Components in AWT

Shapes
- Point, Rectangle, Shape, Dimension

Drawing
- Graphics, Graphics2D

Component
- Inherited by Swing's JComponent

Layout Managers
- FlowLayout
- BorderLayout
- GridLayout
- GridBagLayout

LayoutManagers Not in AWT
- BoxLayout
- SpringLayout

Swing - Java's most common GUI ToolKit.
- Relies heavily on the Model-View-Controller framework.
- Most common controls in Swing.

JFrame - JFrame("Name") sets the frame's name.

JComponent

JButton - JButton("Button Text") sets the button's text.

JTextField - JTextField("Text") adds text to the field.
JTextField(numbColumns) Creates a blank textfield with the specified number of columns.
JTextField("Text", numbColumns) - Creates a textfield with "Text" in it and the specified number of columns.

JLabel - JLabel("Label Text")

JTextArea - Multiline text area.

Layout Managers

BoxLayout, SpringLayout.

Timer

UIManager, JMenuBar, JMenu, JMenuItem BorderFactory Box AbstractAction

---

JFrame - Foundation for all Swing GUI's.
- JFrame is a window.

Creating a simple JFrame

```
JFrame myFrame = new JFrame("Frame Name)"
...
myFrame.setSize(width, height);
myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
myFrame.setVisible(true);
myFrame.setLayout(new BorderLayout());
```

Additional Method
- `getContentPane()` - Get the component container in the window
- `add` - Adds a component to the JFrame.

JComponent - Excluding JFrame, all swing components inherit JFrame.

Provides common functionality
- Tool tips
- Painting and borders
- Double Buffering
- Support for Layout Managers

Container - All GUI components must be placed in a container to be visible.

TopLevel Containers
JFrame, JDialog, JApplet

Other Containers
JPanel, JRootPane, JLayeredPane.

All containers extend:
`java.awt.Container`.

JButton - Encapsulation of a button interface.

```
JButton myButton = new JButton("Click Me!")

JButton btn2=new JButton(new ImageIcon
                         ("Twitter.png"));
```

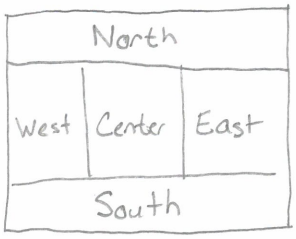
System Look and Feel

Uses the javax.swing.UIManager toolkit.

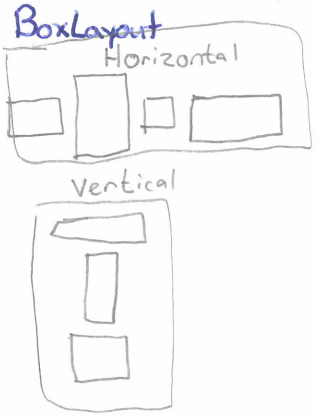
```
try {
  UIManager.setLookAndFeel(
    UIManager.getSystemLookAndFeelClassName());
} catch (Exception ex) { }
```

Layout Manager - Specifies methods for organizing GUI components in a container.

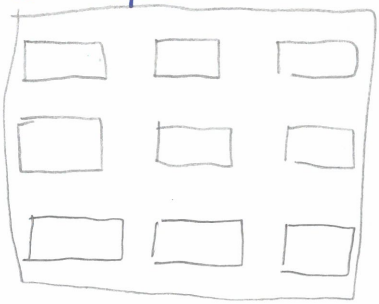
Border Layout



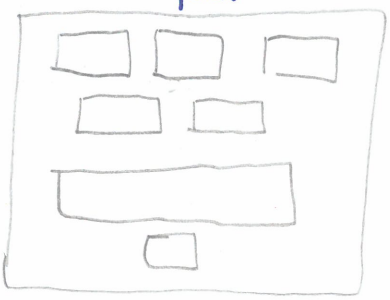
BoxLayout



GridLayout



Flow Layout



BoxLayout Example

```
BoxLayout myLayout= new BoxLayout(myFrame.getContentPane(), BoxLayout.X_AXIS);

JPanel myPanel= newJPanel();
BoxLayout myLayout2 = new BoxLayout(myPanel, BoxLayout.Y_AXIS);
```

Default Layout Managers

JPanel - Flow Layout

JFrame and Content Panes - Border Layout

Adding Spaces

```
myWindow.add ( Box.createRigidArea(new Dimension(10,10));
```

Java library for Box: javax.swing.Box

Java Library for Dimension: java.awt.Dimension

JPanel - Allows you to make Layouts within layouts.

```
JPanel myPanel=new JPanel();
myPanel.add(new JLabel("User Information"));

myFrame.add(myPanel);
```

JLabel - Generic Label Element

JTextField - Generic text entry element

JTextArea - Multiline text entry element

Manual Position

- Pass **null** to **setLayout**
- Use method **setLocation(x,y)**
- Use the **setSize(width, height)** method.

Example

```
JLabel myLabel=new JLabel("User Information");
myWindow.add(myLabel);
myWindow.setLayout(null);
myLabel.setLocation(10,50);
myLabel.setSize(100,300);
```

Extending JComponent - Custom visulization is possible by extending swing's JComponent.

Graphics and Graphics2D

Java Libraries
java.awt.Graphics java.awt.Graphics2D

Steps:
1) In the paint method cast the Graphics object to a Graphics2D object.

```
paint (Graphics g) {
  Graphics2D g2D = (Graphics2D)g;
```

2) Methods
- setColor (Color c)
  java.awt.Color
- drawString
- drawRect, fillRect(int x, int y, int width, int height)
- drawImage(Image i, int x, int y, ImageObserver obs)
- setStroke (new BasicStroke (width));
  java.awt.BasicStroke

Events and Listeners

Events are actions taken by actors (a user, another program) outside of the software.

Listener
- Interface types
- Are collected by components that listen for events.

Types of Listener Implementation
1) An object itself may be its own listener
2) Inner class
3) Anonymous class
4) Container class
   - May listen for all components inside the container.

```
public interface ActionListener {
    public void actionPerformed (ActionEvent e);
}
```

Anonymous Class Action Listener

```
myButton = new ActionListener( new ActionListener() {

        public void ActionPerformed (ActionEvent e)
            JOptionPane.showMessageDialog
                ("Hello World!");

    });
```

Java Imports
java.awt.event.ActionListener
java.awt.event.ActionEvent

Inner Class Listener

```
myButton = new JButton("button text");
class myActionListener implements ActionListener {
  @Override
  public void actionPerformed (ActionEvent e) {
      //--Do Something
  }
}
myButton.addActionListener (new myActionListener());
```

Polymorphism in GUIs

- GUI's are highly decoupled in Java due to both interfaces and inheritance.
- Any class can respond to an ActionEvent.
- Any class can be drawn as a GUI component as long as it extends JComponent.

Icon Interface Type

Three Primary Methods
- getIconWidth
- getIconHeight
- paintIcon

Example

```
public class BoxIcon extend JComponent {
    public BoxIcon(Color boxColor) {
        color = boxColor;
    }
    public void paintIcon(Component c, Graphics g, int x, int y) {
        Graphics g2D = (Graphics2D)g;
        g2D.setColor(color);
        g2D.drawRect (x, y, 99, 99); // Box 100 by 100
    }
```

Design Pattern – A way of solving a problem that occurs repeatedly in a program.

Antipatterns – Solutions that are commonly used but may be ineffective or counterproductive.

Disadvantages of Design Patterns

1) May decrease a program's performance
2) Require effort to organize the code and to keep it organized.

Benefits of Software Architecture

1) Allows for easier modification
2) Increases code flexibility
3) Reduces coupling

Iterator Pattern

- Iterator allows for the traversal of a list while keeping its own state.

Benefits of an Iterator

1) Does not expose the internal structure of the list.
2) Allow for multiple clients to access the list simultaneously (unlike a cursor).

Typical Implementation

- Usually implemented as an inner class.

Interface Methods

`hasNext`, `next`, `remove`

Factory Method Pattern

Problem: A program uses many aggregator class (i.e. storage class, LinkedList, ArrayList, etc), and must get iterators for each one.

Solution: Design an interface with a single method that returns an Iterator.

Factory Method Pattern means a calling class need not know anything about classes to do an activity. All they know is they have a specific name that performs the task.

Example: The Iterable interface has one method for this "`iterator()`"

Command Pattern

- Summary: User interactions can be managed and interpreted more easily if we wrap them in objects.

Solution: Turn method calls into objects.

Benefit of the command pattern

- Enables undo and redo as actions can be kept in a queue
- Stores **what** needs to be done not how to do it.

AbstractAction

- Class in Java that implements the action interface.

Example:

```
class ConsoleAction extends AbstractAction {
    public ConsoleAction(String name, String desc){
        super(name);
        putValue(SHORT_DESCRIPTION, desc);
    }
    public void ActionPerformed(ActionEvent e){
        System.out(getValue(NAME)+" "+getValue
                (SHORT_DESCRIPTION
    }
}
```

Adding an Abstract Action

```
JButton myButton = new JButton("Button Text")
myButton.setAction(new ConsoleAction("Name", "Description"
```

JMenuBar

```
JFrame myFrame = new JFrame("Frame Name");
JMenuBar myMenuBar = new JMenuBar();
myFrame.setJMenuBar(myMenuBar);

JMenu fileMenu = new JMenu("File");
JMenuItem newMenuItem = new JMenuItem("New");
fileMenu.add(newMenuItem);
myMenuBar.add(fileMenu)

newMenuItem.setAction(new ConsoleAction("...", "..."));
//-- Better to put at the end
```

Disadvantages of Patterns

1) **Increased code/development complexity**
   - Every where you implement a pattern you are speculating you will need that flexibility later.
   - Too many levels of abstraction must themselves be learned.
2) **Increased overhead and runtime**
   - Abstract class and interfaces do have runtime effects (although small).

Observer Pattern

Event Driven Architecture – Away of triggering code based on actions from the user or some other entity.
- Architecture used in all modern GUI's.

Problem/Context: An object A is the source of events and object B should respond to these events.

Solution: Define an observer interface the objects of which will be aggregated by A and will notify B.

Model-View-Controller Architecture

- Based off the observer pattern.
- **Model** – Hold the actual data
- **View** – Displays the visible parts of the model.
- **Controller** – Processes user interaction.

Composite Pattern

- **Summary** – Combine several objects into a single object that has the same behavior as its part.

Example: JPanel can aggregate other JPanels.

Antipatterns – Used to lead develops away from bad solutions.
- Often used by are usually ineffective or counter productive.

Blob – System class that takes on unrelated responsibilities.

God object – System class that takes on too many responsibilities.

Poltergeist – Short lived class with no significant responsibilities. Examples can be data formatting classes.

Magic Numbers – Unexplained numbers in a method.

Object Orgy – Public properties and methods exposing the internal state of objects to other objects.

Error Hiding – Using exception handling to hide bugs from develops and users.

Sequential Coupling – A class that requires its public methods be called in a certain order.

Strategy Pattern

Summary – Wrap an algorithm in a class.

Example: **Comparator** interface type.

Problem:
A class can benefit from many variants of an algorithm.

Solution: Provide an interface type that will define the operation. This is the **strategy**. Concrete classes then implement the strategy interface type

Strategy pattern is used to specify **how** something should be done.

Decorator Pattern

Summary: Applies when a class enhances the functionality of another class while preserving the other class' interface.
- Forms a **combined result** of the decoration and the original component.

Border in Swing

```
JPanel myPanel= new JPanel();
myPanel.setBorder(BorderFactory.createLineBorder
                  (Color.BLACK));
```

Java import
javax.swing.BorderFactory

JPanel Constructors

JPanel(boolean isDoubleBuffered)
JPanel(LayoutManager layout)
JPanel(LayoutManger layout, boolean isDoubleBuffered)

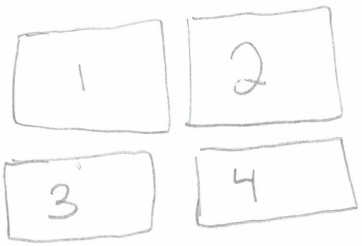
Enumerated Type Java

```
public enum Currency
    PENNY, NICKEL, DIME, QUARTER
};
```

Grid Layout Constructors

```
new GridLayout()
new GridLayout(int rows, int columns)
new GridLayout(int rows, int cols, int hgap int vgap)

myPanel= new JPanel();
GridLayout layout = new GridLayout(2,2);
myPanel.setLayout(layout);
myPanel.add(new JButton("1"));
myPanel.add(new JButton("2"));
myPanel.add(new JButton("3"));
myPanel.add(new JButton("4"));
```



Result

Stroke Example

```
paint(Graphics g) {
  Graphics2D g2D= (Graphics2D) g;
  g2D.setStroke(new BasicStroke(width));
```

Java Import
java.awt.BasicStroke

Programming Notes

- main is public static void
- Always import
  - java.awt.*
  - javax.swing.*
  - java.awt.event.*
- A good option is to have the class extend JPanel or JFrame.
- Do not forget semicolons.

Javadoc Notation

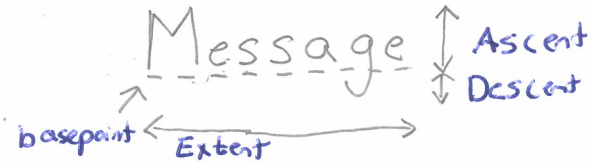
@param – Input parameter
@return – Return value
@precondition
@postcondition

```
/**
    Entry for javadoc
*/
```

Graphics 2D Commands

drawString(String str, int x, int y)
drawLine(int x1, int y1, int x2, int y2)
drawRect(int x, int y, int width, int height)
drawOval(int x, int y, int width, int height)
setStroke
setColor(Color.COLOR_NAME)
fillRect
fillOval

Font Terms



Proportionally Based Font – Different characters have different widths.

Timer

Java Import
javax.swing.Timer

Setting a listener to a timer

```
ActionListener listener = ... ;
int delay = 1000; //-- Time in milliseconds
Timer t = new Timer (delay, listener);
t.start(); //-- Start the timer
```

All times are in milliseconds

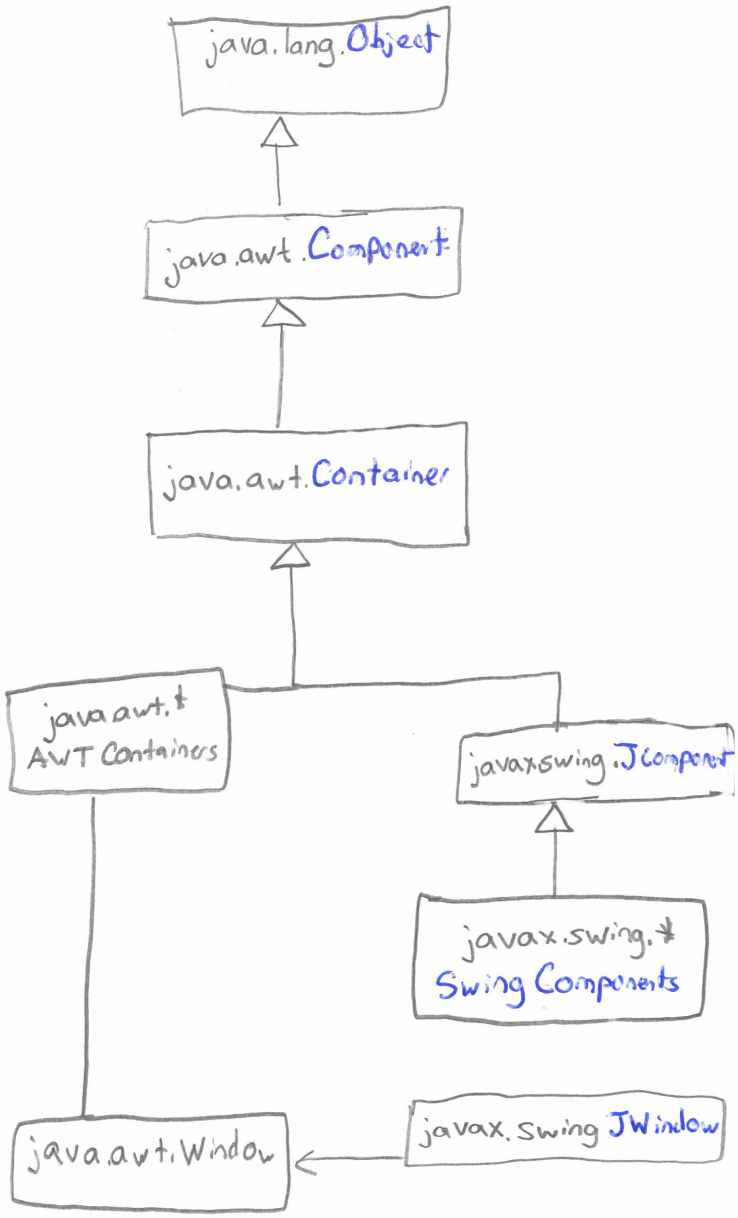
Class BigBox

```
public class BigBox extends JComponent
{

   public BigBox() {
      super();
      setPreferredSize(100,100);
   }

   @Override
   public void paint(Graphics g){
     Graphics2D g2D = (Graphics2D)g;
     g2D.setColor(Color.RED);
     g2D.fillRect(0, 0, getWidth(),
                        getHeight());
   }

}
```



Relationship Between AWT and Swing

Iterator

```
LinkedList<String> list = new LinkedList<String>();

list.add("...")
:
ListIterator<String> li = list.iterator();

while (li.hasNext()){
   System.out.println(li.next());
}

while(String item : list){
   System.out.println(item);
}
```

Total Ordering

Transitivity
$xPy \wedge yPz \rightarrow xPz$

Reflexivity
xPx

Antisymmetry
$xPy \wedge yPx \rightarrow x = y$

Totality
xPy OR yPx