# Lecture 12
# CSE11 Fall 2013

# Introduction to Program Design and Testing

- So far, we have discussed Java syntax, modified existing classes, created some new classes
- We haven't discussed the prospect of “starting from scratch”
  - Given a problem,
    - how do you determine how to structure a solution?
    - What are good ways to break the problem up into manageable pieces?
    - In Java, what classes should I construct and why?

# Designing Programs

- There is no “right” way, no “wrong” way.  Some methods work better than others for you, the group/company work for, and others
- Two Ends of the Spectrum
  - Carefully design everything, up front. Think about as many possible interactions as possible, use cases. Code nothing until you have very complete blueprint
  - Design nothing upfront. Start writing code and hope things work out
    - This is often called “Extreme Programming” (Dr. P. is most definitely NOT a fan).

# Some Background

- You must have a defined goal in mind if you are going to design anything
  - What you would like a program to accomplish for you, e.g.,
    - calculate the mean, average, standard deviation of a column of numbers
    - Store all my movie collection and allow me to find what I own by title, actor, and/or year.
    - Plot a histogram of “red-light camera” tickets given in San Diego over the last 24 months
    - …
- **<u>You have to define the problem before you can design</u>**

# Some more Background

- Going from Abstract to Specific
- Going from Specific to General
- Abstract ==> Specific
  - Designing code and classes
    - Start with words, comments, ultimately actual code
- Specific ==> General
  - Applicability of a program. It needs to solve a specific problem, can it be easily “generalized” to solve a wider range of problems?

- These two may seem in conflict, but they are not.
  - At the beginning we know the **specific problem** we want to solve and we have a **general (abstract/vague) idea how to solve it.**

# The biggest mistake many programmers make

Trying to solve the problem all at once

# Building up in Stages

- Object-oriented (OO) design encourages
  - Thinking about small pieces
  - Assembly into larger functionality
  - Hiding details so that in design one can think more abstractly
- Good OO design is delightful and sensible
- Bad OO design is very painful

# Book Chapter 21

- Suppose we want to play the “shell” game on the computer
  - One players hides a marble under one of three cups and shuffles them on the table. After shuffling player 2 guesses which cup has the marble.

# How do we start

- What problem are we trying to solve?
  - Play the shell game on the computer
- What functionality do we want to support?
  - Begin with 3 cups on and 1 ball and tabletop/screen
  - Allow player1 to hide the marble in a particular cup
  - Allow player1 to shuffle the cups
    - How?
      - Dragging cups?
      - Random animated shuffle?
  - Allow player2 to select which cup he/she believes holds the hidden marble, and then reveal its contents

# Now we have some possible items (objects) to model

- Cups
- Ball
- ~~Tabletop~~
  - ~~Maybe later~~
- ~~Players~~

# Properties and Behaviors of Particular Objects

- Cups
  - Should all have the same appearance
  - Need to be raised to reveal contents
  - Need to have ability to have marble placed into them
  - Must be able to “shuffle” (move on screen, dragged by a mouse)
- Marble
  - Place it in a cup
  - Move it (when it is in a cup, and when it is on the table)
- Overall (control)
  - Reset the game
  - Keep Score?

# Three Classes

```
public class Cup {
}

public class Marble {
}

public class ShellGame extends WindowController {
}
```

# Which comes first behaviors or properties of the classes?

- Honestly, depends on how you reason about problems
  - (I tend believe you really do these together)

```
public class Cup {
    // Properties:  (~Nouns)
    // Position on table
    // Contains a Marble
    // Image of the cup (or shapes)
    // A marble (if one exists)

    // Behaviors:  (~Verbs)
    // Can be moved
    // Raise the cup (to reveal contents)
    // Lower a cup (to hide)
    // add a Marble
    // determine if cup is empty.
}
```

# Translate English to Java (Instance Variables)

```
public class Cup {

    // Properties:
    private Location coordinates; // Position on table
    private boolean empty;        // Contains a Marble
    private VisibleImage theCup;  // Image of the cup
    private Marble theMarble;     // Marble


    // Behaviors:
    // Can be moved
    public void move(double dx, double dy) {}
    // Raise the cup (to reveal contents)
    public void reveal() {}
    // Lower a cup (to hide)
    public void hide(Marble aMarble) {}
    // Add a Marble to the cup
    public void addMarble( Marble aMarble)
    // is empty
    public boolean empty() {}
}
```

# Next Step?

- Some choices
  - Outline the other two classes (Marble and ShellGame)
  - Fill in the details of the Cup class:
    - build a constructor
    - code basic functionality of some/all methods
    -
- Eventually you have to get to some working code *so that you can **test***

# Design Choices and some Refinement

- Think about the Marble class behaviors
  - Show the marble (on the canvas)
  - Hide the marble
  - Move it with the mouse
- VisibleImages from Objectdraw
  - hide() method already defined
  - show() method already defined
  - move() and moveTo() methods already defined
  - ➔ Maybe simpler just to use a VisibleImage object instead of defining a new Marble class?
    - ➔ Trade generality for already-existing code. (judgment call)

# All at once or Piece-by-Piece?

- At this point, you have to take a step back and think about your approach to coding. Let's look at the Cup class
- What is the simplest (useful) piece of controller code that I could write to manipulate a Cup class?
  - e.g., Something that creates -a- cup, drags it around the screen
  - We've built code like that. Re-use it! Perhaps even call it `FakeGame (WeightTest, TestCemetery serve this identical purpose)`

# Practical Approach

1) create Cup class with constructor and a move (or moveTo) method
   - immediately discover that we need a method to determine if we have clicked on the cup (contains()). Add it to your class
2) Build first edition of `FakeGame`, when you click it creates a cup. When you click-and-drag on the cup, it moves.
3) At this point, you some very basic working code. Debug it so that it works. It's small and manageable.
4) Modify Cup.java to be able to add a Marble to the cup. Also to reveal() a Marble. At this point you can decide if a Marble is its own class or native Objectdraw object. It could even be a `Drawable2DInterface`
5) Modify FakeGame to add a Marble and then reveal() it. You could choose different methods to create and test.

# The is a general model of development

- Look at the whole problem, design at a high-level the objects/pieces you need.
- Determine what is a minimal and usable set of methods to create a  particular Class that has some useful (but incomplete) functionality
- Build  a small program to incrementally test each stage of development.
  - As the stages progress, you discover
    - Methods that are missing
    - Other Classes that must be built to move forward.

# Encapsulation

- The ability to hide details of *implementation* for a class, consider:

```
Cup cup1, cup2, cup3;
cup1 = new Cup(.....);
cup2 = new Cup(.....);
cup3 = new Cup(.....);

if (cup1.empty()) {
.... }
```

- As a “user” of a Cup class, Do we care **HOW** the Cup instance determines if it is empty?
  - It could keep an empty/full instance variable
  - It could decide if `theMarble` instance variable is `null`
  - The Cup class could have a static variable called theMarble and it could determine `theCup` and `theMarble overlap()`
- Those Choices of implementation are up to the person who wrote the class

# Their Animated Shell Game

- http://eventfuljava.cs.williams.edu/sampleProgs/ch2
- Let's practice Reading code to determine HOW it works.  In other words, can be go from the specific book implementation back to a high-level view.
- Reading Code and Understanding what it does is an acquired skill. Can't do it without practice
  - You can learn some good (and really bad!) programming techniques/approaches if you do this.