Design

Introduction to the Section:

This may be for advanced students who already know about Procedural Programming. Could be a nice lecture for them as to *why* we teach OO.

This is from a lecture at Georgia Tech, the 7th lecture that they give. Students are assumed to have programming knowledge from either a previous course or coming into school. The course this is adapted from is the second course given in the CS program. Thus, students all know Procedural already, and are being taught OO in Java. Java syntax is already taught, and everything from Objects to full data structures are covered in this course.

Hopefully this adaptation is a possible way of teaching design to non-newbies, as it follows with a design example in both OO and Procedural. Currently the example is straight from their slides, unedited to meet 15's programming style.

Design

Central OO Benefit:

Large OO programs are easier to understand and maintain . . . because they:

- are written in terms of real world objects (not data structures)
- feature strong encapsulation (details are hidden)
- feature strong modularity (making code reuse easy)

Central OO Costs

- slows things down slightly, though this is being negated somewhat by faster hardware
- might be a little too much for small programs that won't ever be reused/modified
- greater need for appropriate analysis and design.

Design

Two phases: *OO Analysis* and *OO Design*

In OOA, ask *"what?"*
- What does the program need to do?
- What classes are needed?
- What does each class need to do?

In OOD, ask *"how?"*
- How will this class handle its responsibility?
- How can I make sure this class has the info it needs?
- How will the various classes communicate?

Switch back and forth between OOA and OOD whenever it makes sense to do so.

Don’t get stuck in one or the other

OO Analysis

Do three things in OOA:

1. Determine functionality of the system (a.k.a. requirements analyis)

2. Create list of classes that are part of the system (and a list of those that aren't!)

3. Distribute functionality of the system among the classes.

Won't say much about #1, it's generally taken care of for you in the specifications.

OO Analysis

2. Creating candidate classes:

Brainstorm:

- think of as many classes as you can, quickly.
- ignore consequences (pretend you don’t have to code it).
- don’t throw things out, nothing is stupid

Where to look for classes:

- go through requirements, underline the nouns.
- think about the system: look for persons,places, things, transactions to be remembered.

Dangers to watch:

- avoid "CS-ey" terms (data-structure)
- question objects w/names ending in "-er"
- avoid class that represents the "whole thing"
- don’t include things you don’t need.

OO Analysis

2. Selecting among candidate classes:

Split candidate classes in 3 groups:
- those that are clearly needed.
- those that you don't need.
- those that you aren't sure about.

For those that you're unsure about, ask:
- Does it encapsulate what would normally be done by person outside the system? (lose it, unless a simulation)
- Is it really identical to some other class?
- Does it have more than 1 possible state; will its state change over life of program? (if not, then it's best an attribute of another class)
- Will it have distinct responsibilities?
- Will it have unique knowledge?
- Is it needed by other classes
- Is it really just an attribute of another class?

OO Analysis

3. Distribute functional responsibilities:

Responsibilities come from *requirements*.

Role play: pretend that you are each object, trace interactions to see *what you need to know and do*.

Focus on *what*, not *how* (for now)

Look for commonalities that you can abstract out via *inheritance* relationships.

Principles:

- Assign responsibility to class(es) that has the knowledge to do it.
- An object is responsible for all things that would be done to it in the real world.
- Distribute responsibility; nobody should be THE center
- Each class should have some responsibility.

OO Analysis

Checkpoint:

- Classes are relatively small.
- Responsibility and control are distributed.
- Few assumptions about language or platform.
- OOA describes the world, not CS jargon.
- Objects all have some responsibility.
- No object is a manager of another's data.
- If requirements extended to include more things, then minimal change to existing part.
- If I/O methods were to change radically, then minimal change to existing part.
- Zero redundancy.

OO Design

Goal: Convert OOA into something you can implement.
Decide: What each class needs to know and what other classes need to know about it.

- Attributes: state of the object, "what it knows"
- Ask if attribute is really its own class.
- Values that object calculates from other attributes are best services, not attributes them-selves.
- Refer to closely related values as one (e.g., name)
- Will every instance need this attribute?

Does it describe me? (add it to me)
Does it describe an object I know? (add to it)
Does it describe something we share (might need an object to encapsulate the interaction)

OO Design

Assigning capabilities:

A *capability* is something the object knows how to do, implemented as a method.

Basic capabilities (*get*, *set*, *add*, *remove*, *init*) are ubiquitous, often not explicit in design.

Most capabilities come from responsibilities already made clear.

Ask "will every object of this class need it?"

Consider moving a capability to another class:
- If capability contains name of another object.
- If capability takes another object as input.

OO Design

Assigning Links (i.e., Relationships):

Inheritance:
<*subclass*> is a (kind of) <*superclass*>

Composition:
<*whole*> has a <*part*>

Broadcast:
<*sender*> sends a message to <*receiver*>

Are you ready to Rumbaugh!!!

(insert Rumbaugh slides)

OO Design

<u>OOD Checkpoint:</u>

- It’s clear from OOD how to write your code.
- Every class has at least one attribute or object connection.
- Every class has at least one service.
- No object knows about everybody in the sys-tem.
- Objects connect to others only if they need info from them.
- No unnecesary middlemen.
- Attributes and services are as high as can be in inheritance hierarchy.

Design Example

Battleship Game via *Structured* (non-OO):

<u>Data structures</u>:

- ShipType:
    - name
    - numHitPoints
    - numHitsTaken
    - xLocation
    - yLocation
    - boardOrientation

- Board
    - array[1d, 2d] of boolean hitsTaken
    - array[maxShips] of ShipType

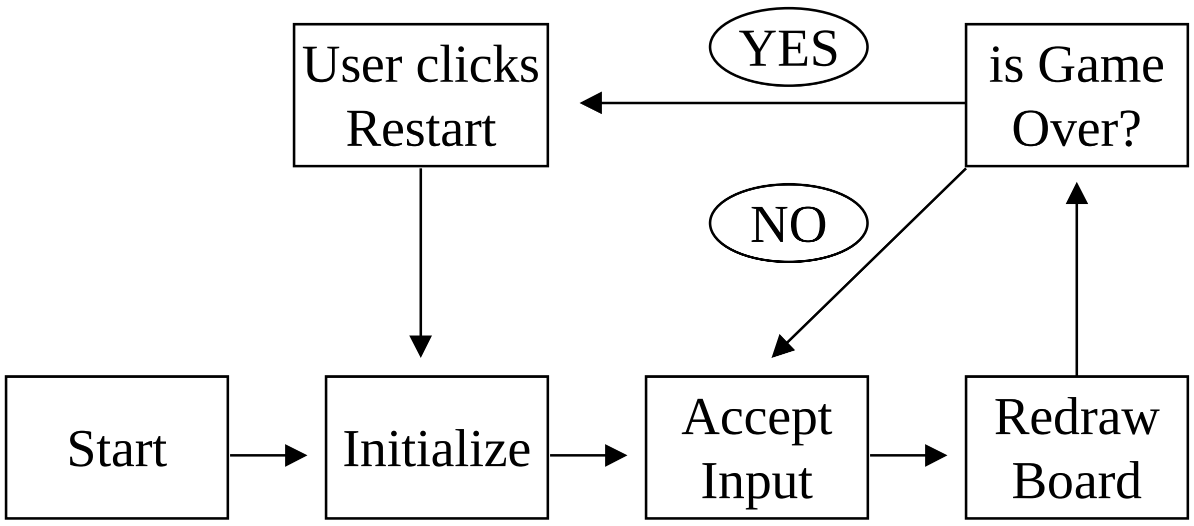
<u>Procs/Funcs</u>:

- initializeTheBoard
- acceptInput
- redrawBoard
- isGameOver

Design Example

Battleship Game via *Structured* (non-OO):

<u>*Control flow diagram:*</u>



Design Example

Battleship Game via *OO*:

```
public abstract class Ship
// Constants:
//      Version (1.0)
//      Debug flag (false)
//      Horizontal orientation (1)
//      Vertical orientation (0)

// Instance variables:
//  strName
//  iHumHitPoints
//  boolean array of iHitsTaken
// iXStartCoord
// iYStartCoord
// iOrientation

// Constructors;
//    set init values

// Accessors:
//     obvious “get the various values”, plus
//     isHit
//     isSunk
```

Design Example

Battleship Game via *OO*:

```
public abstract Ship (continued)

// Modifiers:
//      setHit

// toString
//     convert info re: object to string for output

}  // of class Ship

subclasses of Ship
//  as appropriate given different names,
// different sizes and different hits to sink
```

Design Example

Battleship Game via *OO*:

```
public class Board extends Canvas implements
                              MouseListener, ActionListener

// Constants:
//      Version (1.0)
//      DEBUG (false)
//      NUM_SHIPS (5)
//      BOARD_SIZE (15)
//      GAME_PLAYING (0)
//      GAME_OVER (1)

// Instance variables:
//   array of Ships
//   2d array of boolean for locations ShotAt
//   gameState (on)

// Constructors;
//    set init values, calls Modifer initShips

// Accessors:
//     obvious "get the various values"

// Modifiers:
//      initShips to reset game
```

Design Example

Battleship Game via *OO*:

```
public class Board extends Canvas implements
                              MouseListener, ActionListener

// Private methods:
//     checkNewShip
//     aShipHitAt
//     setShipHitAt
//     isGameOver

// Events and Handlers:
//     paint (draw the board)
//     mouseClicked (handle User interaction)
//     actionPerformed (handle buttons, e.g., StartOver)
```

Design Example

Battleship Game via *Hybrid* (i.e., *poor*) *OO*:

<u>*Data structures*</u>:

- class Ship: with appropriate constructors, accessors, and modifiers for:
    - name
    - numHitPoints
    - numHitsTaken
    - xLocation
    - yLocation
    - boardOrientation
- class Board: with appropriate constructors, accessors, and modifiers for:
    - array[1d, 2d] of boolean hitsTaken
    - array[maxShips] of ShipType

<u>*Procs/Funcs*</u>:

- initializeTheBoard
- acceptInput
- redrawBoard
- isGameOver

Using classes as data structures, that’s all...