Object-Oriented Design

How to outline the classes that you need to create and/or use.
Handy Observation: Nouns and Verbs

Classes are nouns.
- objects (a thing)
  - Cow, Trumpet, House, Zoo

Methods are verbs.
- behavior (an action)
  - calculate, reduce, print, grunt

Gross generalizations!
- But useful.
- Can start with this when given a project.
Many approaches to OO design! Try this if feeling lost.

1. Write verbal description of problem.
2. Pick nouns and verbs.
3. Outline classes with nouns.
4. Outline methods with verbs.
5. Add state and behavior appropriate to each class.
6. Add other missing classes and methods.
8. Fill in missing details.
Example: Pick Potential Classes

- Suppose I want to create a program that handles sales at my small store. The program would find out what item was sold, remove it from the inventory, order more of the item when necessary, calculate the tax, and print a receipt.

- What are the potential classes?
  - Nouns!
  - Later we may decide to add more or delete others.

- What are the potential methods?
  - Verbs!
  - Later we’ll add more.

- Which class has which method?
  - Closest association between noun and verb.
Create class for each noun.

- Get rid of plurals – Sales to Sale.
- Add comments so know what each class will do.

Consider if base class or interface useful.

/** The class that handles events during a transaction. */
public class Sale {}

/** The class that represents the thing being sold – e.g., a lawnmower. */
public class Item {}

/** A list of all the things in the store. In other words, a collection of Items. */
public class Inventory implements Iterator {}

/** A class that handles tax related issues. */
public class Tax {}

/** A class that handles the customer’s receipt. */
public class Receipt {}
Outline the Methods (Verbs)

- Create method for each verb.
  - Will probably add more methods later.
  - Add comments.

/** A list of all the things in the store. In other words, a collection of Items. */
public class Inventory implements Iterator
{
    ...
    /** remove the specified item from the inventory */
    public void remove(Item thing)
    {
    }

    /** add the specified item to the inventory */
    public void add(Item thing)
    {
    }
    ...
    ... also iterator methods…
    ...etc.
}
/** A class that handles tax related issues. */
public class Tax
{
    public static double calculate(double cost)
    {
    }
}

/** A class that handles the customer’s receipt. */
public class Receipt
{
    public void print()
    {
    }
}
Outline Object State and Behavior

- Now add instance variables, constructors, getters and setters.
  - Add details that make the object have the expected state
    - state
    - behavior

/** The class that represents the thing being sold – e.g., a lawnmower. */
public class Item
{
    private String name = null;
    private String description = null;
    private double cost = 0.0;

    public Item(String name, String description, double cost)
    {
        ...
    }

    public double getCost()
    {
        ...
    }

    ...etc.
Outline Interactions

What class calls what methods?

- e.g.,
  - Receipt calls Tax
  - Receipt calls Item to get the cost.

/** A class that handles the customer's receipt. */
public class Receipt
{
    Item[] purchasedItems = null;
    double tax = 0.0;

    public Receipt(Item[] purchasedItems)
    {
        ...
    }

    public void print()
    {
        ...
        for(int i=0; i<purchasedItems.length; i++)
        {
            totalCost += Item[i].getCost();
        }
        double tax = Tax.calculate(totalCost);
        ...
    }
}
Outline Other Details

- Missing classes?
- Missing inheritance/interface?
- Missing behavior?

- Fill in details for the methods.
Describe a real world program you might need.

- Find classes.
- Find methods.

- Which methods in which class?

- Anything obvious missing?
  - There usually is!
Try the following CRC (Class, Responsibility, Collaboration) approach if still stumped.

- Take a set of index cards.
  - On each card
    - Write one class name.
    - Write class responsibilities
      - methods/instance variables
    - Write class collaborators.
      - classes it uses as instance variables.
      - classes passed in as variables to constructor, methods.

- Go through cards again and again, to simplify, add missing responsibilities, add missing classes, etc.

- Convert to real classes.
If card is too complicated (too many responsibilities), then break into sub-classes
Complicated classes

- **Remember that classes should do as little as possible.**
  - Classes blueprints for objects.
  - Objects should have intuitive behavior.
    - A Dog doesn’t do Cat things.
  - **Classes that do too much won’t be intuitive.**
    - And should probably be broken down into subclasses.
    - Car should have instance variables for the Tire, Engine, Windshield, Door, etc. – i.e., composed of simpler classes.

- **Consider graphics**
  - Graphics classes can easily be too complicated!
  - Solution: next page.
Model-View-Controller Approach

Even a simple graphics button seems complex.

- Bad habit to make one class do too much.

- So separate into three classes.
  - Model – which stores component’s content.
    - text that is in JTextArea / whether or not highlighted / etc.
    - label that is in a JLabel
    - label on a JButton
  - View – which displays component’s content.
    - font of the text / size of the text / etc.
    - icon image used for JButton
  - Controller – which handles user input.
    - action caused by mouse click
    - text is being highlighted
    - button is depressed
Example of M-V-C

- **JButton**
  - Contains instance of a class called `DefaultButtonModel`.
    - The Model.
    - Called “composition”
      - composed of the `DefaultButtonModel`.
  - Contains instance of `DefaultButtonUI`.
    - UI for user interface.
    - The View.
  - Events handled by the `ActionListener`.
    - The Controller.

Three classes!
Huh? Why “composed of” ActionListener? Thought this always had to be extended to be used. Yes, but then passed in as a parameter and stored as an instance variable. i.e., public void addActionListener(ActionListener a) {...} Polymorphism!
The Model-View-Controller is an example of a Design Pattern.

- Take Software Engineering!
  - Learn lots of Design Patterns.
    - Patterns of classes that solve recurring OO problems.

- We also address general OO design.
- Learn software lifecycle.
- Learn UML.
Example: Singleton Design Pattern

**Purpose:** Allow a class to be instantiated only once.

**When useful:** Only want one of a certain object

- e.g.,
  - only one login allowed – single user license
  - can’t have two people simultaneously using the same online bank account; both people could withdraw all the money – instantiate account once.

- don’t want to have to keep track somewhere else
- So make the class keep track itself – won’t allow itself to be instantiated more than once.
public class Loner
{

  //static, so can be accessed before class is officially instantiated
  private static Loner lonerInstance = null;

  //constructor – private so can’t accidentally use it
  private Loner();

  //use this in place of the constructor … Loner student = Loner.getInstance();
  public static Loner getInstance()
  {
    //create a new instance only if one doesn’t already exists
    if(lonerInstance == null)
      lonerInstance = new Loner();

    //returns instance stored in the static variable – might be an old instance
    return lonerInstance;
  }

  //other methods go here
  public void…
}
Like this OO stuff?

- Take Software Engineering!
  - Extends OO with design patterns.

- Take Artificial Life.
  - Uses OO. All of it!