ICS 111
Object Design and Implementation

- Testing a class
  - unit tests
- Design and implementation
- Tracing objects
- Problem solving and common patterns

Testing a Class

- our programs so far always have a main method, so testing means to run the code
- how do you test a class that doesn't have a main method?
- if you have an interactive IDE, you can build objects and invoke their methods through the IDE
  - the book mentions bluej.org and drjava.org
- otherwise, you can write a **test program** to create objects and use methods from the class, and make sure it works
  - this program is often separate from the code that will eventually use your new class
  - the program either checks or prints the results of method calls
- to test a method, you need to know what it should do
  - what outputs and results are correct
  - for the current values of parameters and instance variables

Unit Test

- in some development environments, every class that is developed must be delivered with other code that tests the functionality of the code
- if the class is X, the testing code is called a **unit test** for X
  - X is the unit being tested
- this at the other extreme from having stub methods: a unit test tests the methods first, before testing the code that uses the methods
- a unit test should:
  - call every method of the class
  - try to make sure every code **execution path** is executed at least once
    - for example, if possible, make sure that you are testing both the if and the else part of every conditional statement (including every else if part)
  - also test common cases likely to be used in practice, and that the class is particularly designed for
- the unit test can be used during development to verify functionality
- the unit test should be run again after any changes, to verify that the changes have not introduced bugs

Class Design and Implementation

1. design all the methods that the class will provide
   - for each method, choose a name, return type, and parameter list
   - and provide a javadoc explanation of what each method does
   - same for each constructor
   - if helpful, can write sample code that would call these methods
2. design instance variables that will allow you to implement these methods
   - you may have to try different instance variables, different types, or different combinations of instance variables
3. implement methods and constructors
4. test the class
   - a very simple test program might be enough initially

- each of these is a creative activity
  - like cooking, it is a creative activity designed to deliver useful outcomes
- in many cases, there is more than one way to do each of these steps

Class Design Philosophy

- If you have trouble describing what a method does, or choosing a name for it, it may be a good idea to consider alternatives
- while the design usually guides the implementation
  - what we want should determine what we do!
- sometimes the implementation can also guide the (re)design
  - what we can do may determine what we want
- as always, compiler errors suggest areas where we haven’t been thinking clearly, and can be used to focus our redesign or debugging efforts

Class Design Example

- Let's say you want to design a class to keep track of grades in a class
- there may be a single method to add a grade:

  `void addGrade(int value, String gradeType)`

  - the gradeType could be a String such as "lecture" or "quiz"
  - or the gradeType could be an int, with 1 representing lecture, 2 representing quiz, and so on

    `void addGrade(int value, int gradeType)`
- or you may have separate methods:

  `void addLecture(int value)`

  `void addQuiz(int value)`
- you have to decide whether you want to add a date for each value, and if so, in what format
  - may use `java.util.Date` or `java.util.Calendar`
- then of course you'll want a method to calculate the final grade

  `double finalGrade()`
- and a method to print all the grades

  `String toString()`
- Then choose instance variables:
  - perhaps an ArrayList of integers
  - and perhaps an ArrayList of Date or Calendar values
  - or perhaps an ArrayList of an object (maybe `Grade`) that you also define

Tracing Objects

- the **state** of each object is the collection of the values of all its instance variables
- if you know the state of an object and the values of the parameters, you can trace any method
- it is good to group all the instance variables for a single object together on a page
  - the textbook suggests index cards
  - a word processor file or text file in a computer is also fine
- multiple objects of the same type each have their own instance variables

Common patterns: keeping a total

- everything from grade objects to bank account objects need to keep a total
  - as well as, perhaps, a history
- the total will be a numeric value of type int, long, or double
- class methods can read the value, change the value, or both
- the total value must be initialized by the constructor
  - perhaps to a default value such as 0
  - perhaps to an initial value, such as when opening an account
- there may be a mutator method to reset or change this total value

Common patterns: counters

- computers are good at counting
- it is natural to want to keep track of the number of times something happens
  - needed for computing averages
  - may be useful for other things
- a counter may be incremented by a specific method call
- or as a side effect of calling a method whose main purpose is to do something else (e.g. record a transaction)
- you probably want an accessor method to return the value of the counter
- and may want a mutator method to reset the counter

Common patterns: storing values

- it is fairly normal for an object to keep a collection of values of some type
  - a shopping cart object holds a collection of intended purchases
    - each purchase could be represented as a string
    - or each purchase could be represented by a separate Purchase object, with its own methods
- if you have a fixed number of values to store, you can use an array
- otherwise, it is usually more convenient to use an ArrayList
  - for example, ArrayList<Purchase> cart;
- the constructor must initialize this ArrayList
  - with cart = newArrayList<Purchase>();
  - otherwise cart defaults to null
  - and all your methods have to test for the possibility of cart being null

Common patterns: specific properties

- the object `Purchase`, described in the previous slide, has several properties:
  - name of the item
  - price
  - quantity
  - perhaps other properties, such as color, size, or weight
- generally the constructor will take all these properties and for each initialize a corresponding instance variable
- some properties may be optional:
  - if the instance variable is an Object, it can be set to `null`
  - if it is a basic type, you can provide a boolean companion instance variable to record whether the property is valid

    ```
    boolean hasColor = false;
    ```
- for most properties, it is a good idea to have at least an accessor methods
- often mutator methods are also useful
- a `String toString()` method is always useful for debugging, should normally be provided
- such a pattern applies to many different possible objects
  - including those representing people or objects in any database

Common patterns: objects in different states

- in Java, the state of an object is the current values of its instance variables
  - does not refer to U.S. states!
- some objects have a small number of different states, in which they do distinctly different things
  - for example, an item in an online store could be either available, on order, or unavailable
- the methods do different things in different states
  - and might even throw `java.lang.illegalStateException`,
  - e.g. if you try to purchase an item that is unavailable
- this state is usually kept in an instance variable
  - the variable may be an int, e.g. 1 for available, 2 for on order, 3 for unavailable
  - or may be an enum
- calling different methods may cause **state transitions**
  - accomplished by assigning a new value to the state variable
- there is a theory of state machines, theoretical objects in which events cause actions and state transitions
  - the combination of event and current state determines the action and next state
- state machines are useful for modeling real-world objects
- state machines are useful for designing networking protocols
  - if curious, you can look at page 23 of the original specification of TCP

Common patterns: object positions

- sometimes objects represent things in the real world
- and sometimes they represent things on a screen
- in either case, the object should record a position
- positions may be 2-dimensional (x, y) or 3-dimensional (x, y, z)
- the object correspondingly needs two or three numeric instance variables
  - int, long, or double may each be suitable, depending on the application
- if a velocity needs to be recorded, it can be recorded as a direction (e.g. an angle) and a speed, or an x-speed and a y-speed
  - for three dimensions, we need a speed and two angles, or three dimensions of speed
- similarly for acceleration
- mutator methods that record an object’s motion will change the position
  - if a speed is defined, these methods may use speed and time to compute a new position

Summary

- testing and tracing of objects
  - unit tests can give you confidence in the implementation of a method
- it is best to first design an interface for a class, then implement the interface
  - interfaces evolve, and the implementation has to follow
- but sometimes the implementation dictates parts of the interface
- many common patterns, all reflected in the instance variables and the methods that use and update those instance variables