Object-oriented Design

Reading Assignment
Chapters 1-2

Designing Software

- How do we solve a large problem?
  - break it down into smaller ones
    - Top-down design
    - Divide and conquer
    - *Separation of concerns*
    - *Stepwise refinement*
  - if some of the problems have already been solved, identify existing *components* that can be reused
  - when the problems are small enough, solve them directly
- How do we divide a problem?
  - By identifying functionality
  - By identifying classes (data and code)
  - By identifying reusable components
  - By separating concerns

What is good design?

- Objective
  - Identifying classes (i.e., fields and methods)
- Software engineering principles
  - Encapsulation
    - Package data and access functions
    - Abstract data types (ADTs)
    - Localize changes
  - High cohesion among the members of a class
    - Lots of dependencies among the methods and fields of a class
  - Low coupling among classes
    - Classes communicate by calling each others' methods
  - Small interfaces
    - Keep parameters lists short (i.e., 0 to 3 parameters)
    - For longer lists, pack the list into a class/object
  - Information hiding
    - Keep the fields as private, protected, or package as possible
    - Provide get() and set() functions for private, protected, and package fields

Design techniques

- Identifying responsibilities and behaviours
  - Divide the work among different actors, each with a different responsibility (i.e., verb)
  - Striving for independence; define the work for each class to be as independent from each other as possible; each class should have some autonomy
  - Define the behaviour (i.e., methods) of class so that it is easily understood by other classes; the set of methods of a class constitutes the *protocol* of a class
- CRC
  - Responsibility-collaborator cards
  - Index cards
  - Left: responsibilities of the component
  - Right: collaborators of the component
- UML
  - Unified Modeling Language
  - Industry standard
- Goodrich & Tamassia, p. 42-43

CRC cards

Class Name:	
Superclasses:	
Subclasses:	
Responsibilities:	Collaborators

A Simple Object-Oriented Design Technique

- Start with a statement of the problem to be solved.
  - Circle or colour all the important-looking *nouns*
    - they become candidates for *classes* and *fields*
  - Underline or colour all the important-looking *verbs*
    - they become candidates for *methods*
  - Put them together into coherent *classes*
    - write a short description for each class
  - Identify *relationships* among the classes
  - If classes share characteristics, extract them into superclasses (i.e., *generalization)*
- If your descriptions or relationships use vocabulary not yet identified, iterate through the process.

Design Quiz

The registrar's office is upgrading its course registration system to be written in Java. Students can enroll in courses being offered, but each course has an enrollment limit: once this is reached, all further registrants are placed on a waiting list. Students can also withdraw from a course they are enrolled in.

Design Quiz

The registrar's office is upgrading its course registration system to be written in Java. Students can enroll in courses being offered, but each course has an enrollment limit: once this is reached, all further registrants are placed on a waiting list. Students can also withdraw from a course they are enrolled in.

Nouns - candidate classes, object, fields

Verb – candidate methods

Registrar's Office

- Classes
  - Student
  - StudentList
  - Course
  - CourseList
  - Registrar
- Fields
  - Student
    - id
    - first
    - last
  - StudentList
    - noStudents
    - sl
  - Course
    - id
    - limit
    - rl
    - wl
  - CourseList
    - noCourses
    - cl
  - Registrar
    - cl
    - sl

Registrar's Office

- Classes
  - Student
  - StudentList
  - Course
  - CourseList
  - Registrar
- Methods
  - Student
    - Student()
    - getId()
    - getFirst()
    - getLast()
  - StudentList
    - StudentList()
    - getNoStudents()
    - add(Student)
    - remove(Student)
    - getSL()
  - Course
    - Course()
    - ful()
    - add(Student)
    - remove(Student)
    - getId()
    - getLimit()
    - getRL()
    - getWL()
  - CourseList
    - CourseList()
    - getNoCourses()
    - getCL()
    - add(Course)
    - remove(Course)
  - Registrar
    - Registrar()
    - getCL()
    - getSL()
    - add(Course)
    - remove(Course)
    - add(Student)
    - remove(Student)

Strategies for Unit Testing

- *Unit testing*
  - The practice of testing a single method or class, separately from the overall program in which it is used
- Important things to test for
  - API of a class (methods, parameters)
  - Proper initialization of fields
  - Boundary conditions (e.g., array bounds, off by one)
  - Error conditions
  - Execution paths (*statement coverage*)
- Using `println()` and `toString()` for debugging purposes
  - Design this ability in from the start like other requirements
  - Write/override the `toString()` method for each class
  - You can then print before-after pictures in your test code
- Code inspection and code walk-throughs

Bottom-Up Testing

- Why is bottom-up testing a useful approach?
  - the smaller the piece of code being tested, the easier it is to locate and fix bugs
  - if the code being tested has dependencies, those dependencies are also tested
  - so start at the bottom, with the smallest possible modules and fewest dependencies
- Classes are tested from the bottom to the top of the class hierarchy
- If a group of modules forms a dependency cycle, test the cluster as a whole—so avoid creating dependency cycles!

> Bottom-Up Testing Order Principle:
> Whenever possible, before testing a given method X, test all methods that X calls or that prepare data that X uses.

Top-Down Testing

- Why would you want to do this?
  - when working in a team, layers are often implemented in parallel
  - A component may depend on others that aren't available yet
  - Don't wait for others before starting testing; use stubs
- How to do it?
  - *stub out* any dependencies of your component: fake realistic results with a minimum of effort
  - the stub might:
    - return a very small number of hard-coded items
    - only be able to deal with your specific test data
- Stubbing out components can also be useful in breaking dependency cycles, allowing the co-dependent components to be tested individually

Integration, Acceptance and Regression Testing

- When unit testing is complete, you must test the interactions of the classes with *integration testing*
- When the project is complete, you often have to run a final *acceptance test* before the customer officially accepts your work
- Once a system is released into service, it enters the *maintenance phase* of its lifecycle
  - in this phase, more bugs are discovered and fixed, and new features added
  - as things change, you want to do *regression testing* to make sure that the changes don't break previously working code
  - it's useful to have a suite of test drivers that can automatically run all unit and integration tests, and report on the results
  - note: it's very common for fixes or upgrades to interfere with seemingly unrelated code

Assertions

- An *assertion* is the statement of a fact that should be true at a given point in the execution of a program
  - assertions can be written as comments, to document the code:

```
count--;
// assert:  count >= 0
```

  - they can be written as code, to verify assumptions at runtime:

```
count--;
if (!(count >= 0)) throw new AssertionFailure();
```

- An assertion at the beginning of a method is called a *precondition*
  - it will often validate the method's arguments
- An assertion at the end of a method is called a *postcondition*
  - it will often validate the method's work and/or result
- When assertions are stated using a formal logical language, it's sometimes possible to prove a program's correctness; this is called *verification*

Basic Idea

- The class:
  - It has attributes that uniquely define an object.
    - instance variables
  - It allows some access to other classes
    - through public methods
  - It allows internal activities
    - through private methods and variables
- An object can be anything we imagine it to be:
  - A computer interpretation of a tangible thing
    - car, cartoon drawing, check out line
  - A collection of data
  - A single data item