Lecture 01: Software Design and Implementation
[CMPU-203]
August 31, 2017

Sameer S Pradhan
Vassar College
Course Logistics
A Few Announcements

- Course website

- There were no readings for today.

- Attendance (only for a few initial classes).

- Has everyone brought their laptops? We may not use it today, but it would be good to bring it with you.
  - In case someone does not have a laptop, please contact me. They can check one from the college for the semester.
What is it?

- Software engineering: useful programming
- Scale (useful programs are big).
  - Working with other programmers.
  - Dividing programs into manageable pieces.
  - Building re-usable components.
- Correctness (useful programs must work).
  - Robustness (tolerate errors).
  - Maintainability.
  - Security (sometimes).
Implementing Program Modules

- Big systems must be built of smaller parts.
- Each programmer works on one at a time.
  - With many programmers, a distributed process.
  - Less time spent on communication, better.
  - Less sensitivity to each other’s schedules, better.
  - Method: Agree what each module must do; go separate ways to work out how.
- Parts should be re-usable when it makes sense.
  - In different projects. Hard, takes extra work.
  - In new releases of same project.
  - In other words, should tolerate change.
It’s not enough that your program (class, method,...) works now; what matters is how well it keeps working when conditions change.
Abstraction

- Rules of language give meaning to code.
  - Know language + read code $\rightarrow$ know what code means.
  - This meaning is concrete.
- Program modules should also have intent, or abstract meaning.
  - Example 1:
    - Concrete: Function returns 0.15 times its argument.
    - Abstract: Given check total, function returns tip amount.
  - Example 2:
    - Concrete: This object consists of an array of objects and an integer.
    - Abstract: This object represents a stack.
- “Tip” and “Stack” are abstract ideas, called abstractions.
- Corresponding function or class implements the abstraction.
Decomposition and Abstraction

- **Decomposition**
  - When to decompose
  - Identifying components
  - Modelling components

- **Abstraction**
  - Abstraction by parameterization
  - Abstraction by specification
  - Pre-conditions and Post-conditions
Decomposition

- Large problems can be tackled with “divide and conquer”
- Decompose the problem so that:
  - Each subproblem is at (roughly) the same level of detail
  - Each subproblem can be solved independently
  - The solutions to the subproblems can be combined to solve the original
- Advantages
  - Different people can work on different subproblems
  - Parallelization may be possible
  - Maintenance is easier
- Disadvantages
  - The solutions to the subproblems might not combine to solve the original problem
  - Poorly understood problems are hard to decompose
Decomposition Examples

- Decomposition can work well:
  - E.g. designing a restaurant menu

  ![Restaurant Menu Diagram]

- Decomposition doesn’t always work
  - E.g. writing a play:

  ![Play Writing Diagram]

- Decomposition isn’t always possible
  - For very complex problems (e.g. Managing the economy)
  - For impossible problems (e.g. Turning water into wine)
  - For atomic problems (e.g. Adding 1 and 1)
How to decompose

- Step 1: Identify components
  - A good decomposition minimizes dependencies between components
    - A coupling - a measure of inter-component connectivity
    - A cohesion - a measure of how well the contents of a component go together
  - Information hiding
    - Having modules keep their data private
    - Provide limited access procedures
    - This reduces coupling
Step 2: Model the components

At the design level

structure charts

object diagrams

dataflow diagrams

At the coding level

procedure declarations

float sqrt(int x)

/* requires: x is a positive integer
   effects: returns an approximation of the square root of x to within ±10^-4 */
Abstraction

- Abstraction is the main tool used in reasoning about software.
- Why? It allows you to:
  - Ignore inconvenient detail
  - Treat different entities as though they are the same
  - Simplify many types of analysis

Example abstractions:

- A file
- A sequence of bits on a disk
- A program that takes an integer and a list returns the index of the first occurrence of the element or null if the element does not occur in the list.
Can I replace A with B?

A

```java
found = false;
i = lowbound(a);
while (i < highbound(a)+1){
    if (a[i] == e) {
        z = i;
        found = TRUE;
    }
    i = i + 1;
}
```

B

```java
found = false;
i = highbound(a);
while (i > lowbound(a)-1){
    if (a[i] == e) {
        z = i;
        found = TRUE;
    }
    i = i - 1;
}
```
Using Abstraction

- Abstraction can help with Decomposition
  - e.g. To manage the economy, try focussing on some abstracted features such as inflation, growth, GDP, etc.
  - Abstraction allows us to ignore inconvenient details

- In programming
  - Abstraction is the process of naming compound objects and dealing with them as single entities
    - (i.e. ignoring their details)

- Abstraction doesn’t solve problems...
  - ...but it allows us to simplify them