Lecture Notes 2 – Software engineering, C++
CSS 501 – Data Structures and Object-Oriented Programming – Professor Clark F. Olson

Reading for this lecture: Carrano, Chapter 3, C++ Interlude 1

Software engineering principles
Let's start by discussing some of the fundamental principles that are useful in building complex computer programs and that form the basis for software engineering. Software engineering is a branch of computer science that ***provides techniques to facilitate the development of computer programs*** (one textbook definition). It defines the process by which programs are designed, coded, tested, and maintained (among other steps). Rather than just starting coding at some arbitrary point, a problem solving approach is used for the developing programs to accomplish some goal.

The Life Cycle of Software
While you may have (so far) written only simple programs (for example, for courses) that were discarded once it demonstrated that you could use certain skills, good software undergoes continuing process that is called its life cycle. One version of this life cycle consists of:

1. **Specification:** You must first understand exactly what the problem is that you are solving. You may get an initial program specification from a customer or non-technical person, so it may at first be imprecise. Specification requires that you have a complete understanding of many issues including: what the **input** data is, **what types of data are valid**, what sort of **interface** the program will have, what input **errors** need to be detected, what **assumptions** are made, what **special cases** need to be handled, what is the **output**, what will be documented, what changes might be made after completion, etc. Note that the specification should not include the method of solving the problem.

2. **Design**: After you know exactly what problem is to be solved, a solution to the problem can be designed, encompassing both **algorithms and abstract data types**. For large programs, this is usually broken down into well-defined **smaller problems** that can be solved using **modules that are reusable** and (somewhat) **independent.** (For example, a sorting routine can be used by many programs but, aside from the interface, is independent of them.) A module can be a single function or a group of functions. The **input, output, purpose, and assumptions** of each module should be specified. The design should be **independent of the implementation.**

3. **Risk analysis**: All software projects have some risk, at least in terms of **cost and schedule**. Many have additional risks (for example, software to control a power plant). This course will not discuss risk analysis in detail, but you may see it more in a future course.

4. **Verification**: It is possible, in some cases, to prove that an algorithm is correct. For example, **loop invariants** can be used to prove that a loop performs the correct operation.

   Example:

   ```
   /* Code fragment to sum the items in a list */
   int sum = 0;
   int j = 0;
   while (j < n)
           {
           sum += item[j];
           j++;
           }
   ```

   Loop invariant: sum is the sum of elements item[0] through item[j-1]
   To prove the correctness of the loop, you have to show that:
   - **The invariant must be true initially (this is a pre-condition)**
   - **Execution of the loop preserves the invariant**
   - **The invariant captures the correctness of the algorithm**
   - **The loop must terminate**

5. **Coding**: In this phase, the algorithm and abstract data types are translated into a **particular programming language**. When software is well designed, this is a **small part** of the software life cycle.

6. **Testing:** This phase is used to **detect** as many **errors** in the design and coding as possible. While the program should be tested **as a whole, each module** should also be tested separately, if possible, using **valid data** for which the correct result is known. Testing should encompass as many **special cases** as possible, including **invalid input**, unless the specification allows the assumption of valid input.

7. **Refining**: You now have a working program. What refinements are necessary? Maybe your initial design accomplished everything the customer needed, but not everything that customer wanted. Sometimes a good design strategy is to make simplify the problem, solve the simple problem, and later add refinements to remove the simplifications. However, care must be taken to ensure that the refinements do not require completely redesigning the system.

8. **Production**: This phase includes **distribution, installation, and use** of the software. We will not discuss this phase in detail.

9. **Maintenance**: Once the software is in use, people will find **bugs**, suggest **changes,** and ask for **new features.**

The software life cycle is usually drawn as a wheel with nine spokes. Note that **documentation** forms the center of wheel, since it is a key aspect of all phases of the software life cycle. This is one of the most important aspects of software. For large programs, different people are often responsible for different parts of the life cycle and documentation is crucial to understanding the phases that other people have worked on. Even if you are the only person who will ever use a program (unlikely, except for trivial programs), when you come back to it a week, a month, or a year later, will you remember the specifications, the design details, or other steps? **Document each phase** so that you don't need to!

Each function should have a set of *pre-conditions* specifying the conditions that must exist at the beginning of the function, and a set of *post-conditions* that specify the conditions after the function has completed.

Example:

```
// sort:  function to sort an array of integers
// preconditions:  n is the number of values to sort and is no less than zero.
//                 array has been allocated and holds at least n integers.
// postconditions: the first n values of array are sorted into non-decreasing order.
//                 specifically array[i] <= array[i+1] for 0 <= i < n-1
void sort(int n, int array[])
```

This is the kind of documentation that I want to see in this class. Each of the function parameters (including return values) should be mentioned in documentation. Note that the preconditions and postconditions say nothing about how the sorting is achieved. For a complex function, you should also say something about how the function works and include comments inside the function for each code block.

C++ concepts

See the C++ notes linked on the course web site:
http://courses.washington.edu/css342/zander/css332/
In particular, the following pages:
C/C++ data types, basic operators, and control structures
Defining consts
C++ I/O
C++ functions and pass by value vs. pass by reference
Arrays as parameters to functions

For next lecture:
The Rational class header file
The Rational class .cpp
Using Rational objects, sample driver