Chapter 1: Key Concepts of Programming and Software Engineering

Software Engineering

- Coding without a solution design increases debugging time - known fact!
- A team of programmers for a large software development project requires
  - An overall plan
  - Organization
  - Communication
- **Software engineering** provides techniques to facilitate the development of computer programs. It is the use of technologies and practices from computer science, project management, and other fields in order to specify, design, develop, and maintain software applications. There is no single best way to build software, no unifying theory about how to do this, but software engineers do share certain common ideas about software development.
- *Software engineering is not just for large projects.* There are principles of software engineering that are applicable to small-scale program development as well. They include methods of design, testing, documentation, and development.

Object-Oriented Problem Solving

- **Object-oriented analysis and design** (OOA & D) is one particular method of problem solving. In OOA & D,
  - A problem solution is a program consisting of a system of interacting classes of **objects**.
  - Each object has characteristics and behaviors related to the solution.
  - A **class** is a set of objects having the same type.
- A solution is a C++ program consisting of modules, each of which contains one or more of
  - A single, stand-alone function
  - A class
  - Several functions or classes working closely together
  - Other blocks of code

Abstraction and Information Hiding

Abstraction

- Abstraction separates the purpose of an object from its implementation
- Specifications for each object or module are written before implementation

Functional Abstraction

Function abstraction is the separation of what a program unit does from how it does it. The idea is to write descriptions of what functions do without actually writing the functions, and separate the what from the how. The client software, i.e., the software calling the function, only needs to know the parameters to pass to it, the return value, and what the function does; it should not know how it actually works. In this sense, functions become like black boxes that perform tasks.

Data abstraction

Data abstraction separates what can be done to data from how it is actually done. It focuses on the operations of data, not on the implementation of the operations. For example, in data abstraction, you might specify that a set of numbers provides functions to find the largest or smallest values, or the average value, but never to display all of the data in the set. It might also provide a function that answers yes or no to queries such as, "is this number present in the set?" In data abstraction, data and the operations that act on it form an entity, an object, inseparable from each other, and the implementation of these operations is hidden from the client.

Abstract data type (ADT)

- An **abstract data type** is a representation of an object. It is a collection of data and a set of operations that act on the data.
- An ADT's operations can be used without knowing their implementations or how data is stored, as long as the interface to the ADT is precisely specified.
- A **data structure** is a data storage container that can be defined by a programmer in a programming language. It may be part of an object or even implement an object. ***It is not an ADT!***

Information Hiding

- **Information hiding** takes data abstraction one step further. Not only are the implementations of the operations hidden within the module, but the data itself can be hidden. The client software does not know the form of the data inside the black box, so clients cannot tamper with the hidden data.
- There are two views of a module: its **public view**, called its **interface**, and its **private view**, called its **implementation**. The parts of a class (object) that are in its public view are said to be exposed by the class.

Principles of Object-Oriented Programming (OOP)

- Object-oriented languages enable programmers to build classes of objects
- A class combines
  - Attributes (characteristics) of objects of a single type, typically data, called data members
  - Behaviors (operations), typically operate on the data, called methods or member functions
- The principles of object-oriented programming are
  - Encapsulation
    - Objects combine data and operations

  - Information hiding
    * Objects hide inner details
  - Inheritance
    * Classes can inherit properties from other classes. Existing classes can be reused
  - Polymorphism
    * Objects can determine appropriate operations at execution time

Object-Oriented Analysis and Design

- **Analysis** is the process of breaking down the problem in order to understand it more clearly. It often uses domain knowledge and theory to cast the problem into a perspective that makes it easier to understand.
- The goal is to express what the solution must do, not how it must do it.
- **Object-oriented analysis** tries to cast the problem into a collection of interacting objects. It expresses an understanding of the problem and the requirements of a solution in terms of objects within the problem domain. It does not express a solution – just its requirements.
- **Object-oriented design**, in contrast, is the act of using the results of an object-oriented analysis to describe a solution as a set of objects and how they interact. The interactions between a pair of objects are called collaborations. One object sends a message to another as a means of asking the latter to perform some service or task on its behalf.

Uniform Modeling Language (UML) (Optional)

- UML is a modeling langauge used to express an object-oriented design. Unlike programming languages and spoken languages, its elements include many pictures, or diagrams, that depict the solution. It is therefore easier for most people to understand than descriptions consistsing entirely of words.
- Object-oriented analysis can be made easier using **use cases**, which in turn lead to the creation of UML **sequence diagrams**.
- A use case consists of one or more **scenarios**, which are plain text descriptions of what the solution to the problem should do in response to actions by its user. Some scenarios describe "good" behaviors (the user enters two numbers, and the system displays their sum), whereas some represent "error" behaviors (the user enters one number, and the system displays an error message.)
- Scenarios represent what the system must do, i.e., its responsibilities, not how it should do it.
- Scenarios are converted to sequence diagrams during object-oriented design. There are specific rules for drawing sequence diagrams, such as that all objects are represented in a single row as rectangles, and that solid lines mean one thing and dashed lines another. This set of notes does not elaborate on this topic.
- The sequence diagrams are annotated with text that eventually gets converted to class methods and data, as well as ordinary variables and parameters.
- The sequence diagrams lead to the design of actual classes, in a language independent way.
- UML **class diagrams** represent the classes derived from the object-oriented design and sequence diagrams. A class diagram is a rectnagle with the name of the class and possibly its methods and data, in a very specific syntax. It does not show how the class is implemented, but only its interface.

- Class diagrams can consist of multiple classes connected by various kinds of lines indicating the relationships among the classes. A relationship might be containment (this class is a part of the other) or associations, which are less constrained and just indicate that one class might require the services of the other. Yet another relationship is called generalization, which means that one class inherits from the other.
- UML is just one approach to object-oriented analysis and design.

The Software Life Cycle

Software follows a pattern of development known as the **software life cycle**, which is the sequence of events that take place from the moment software is conceived until it is removed from service. There are many ways to view this life cycle. This is one traditional view of the cycle:

1. **User Requirements** document
   (a) Specifies informally the user's understanding of the problem
   (b) Example: I want a program that displays a text file on the screen.
2. **Requirements Analysis** -> problem specification document
   (a) Specifies in unambiguous, complete, and consistent logical sentences, how the program is supposed to respond to inputs and other external events that happen while it is running.
   (b) Example: should the program wrap long lines, or try to justify lines on the screen; should it display tabs as tabs or replace them as sequences of blanks; what if the file does not exist, or if it is not a text file, or what if the user does not have permission to access that file?
3. **Design Analysis** -> program design document
   (a) Specifies how the program is to be constructed. What are the different classes? How do they interact with each other? What are their interrelationships? How is the development effort to be divided up, i.e., which logical elements will be in the same physical files or even within the same classes? What are the dependencies? What is the data? How will error conditions be handled? These are the types of questions answered during design analysis.
   (b) Example: The main program will prompt the user for the name of the input file and check that the user has permissions. The module that reads from the input file stream will have "switches" to control how it behaves. For example, it will have a parameter that tells it whether to convert tabs to spaces and if so, how many, and it will have a parameter that tells it whether to wrap lines, and another on whether to justify lines.
4. **Implementation** -> program source code and documentation
   (a) The program source code is the human readable executable code that gets compiled, linked, and loaded for execution. It is the result of tedious logical design and understanding.
5. **Test Design** -> Test plan document
   (a) Before you test your code, you should plan out the entire set of tests. The plan includes expected outputs, parts of the specification that the test covers, and so on. This document is required by many government agencies if you are building software for controlled applications like medical, pharmaceutical, military, or financial products.
6. **Testing and Validation** -> test cases documentation

   (a) This contains the actual test results, i.e., what really happened and what was done to fix errors if errors were found.

   (b) Example: When the program was given a file whose last character was a tab, and tab translation was turned on, it did not convert the last tab to a sequence of blanks. The error was traced to a loop condition that had the wrong comparison operator.

   (c) If tests fail, return to whichever step must be repeated. (E.g., sometimes it is a design flaw or a specification flaw)

7. **Production** -> released software

   (a) The software is placed into production, so it is now in use by the user, and exposed to potential risks and losses.

8. **Maintenance** -> changes in software

   (a) The users discover bugs or weaknesses such as confusing interfaces or hard to understand forms and these are reported and eventually fixed.

Software Quality

- The "goodness" of software is its quality. The factors that contribute to the worth or value of software from a user's perspective include:
  - Cost, including cost of acquisition, efficiency, resource consumption
  - Correctness or reliability
  - Safety and security
  - Ease of use
- From the developer's perspective, the factors are:
  - Cost of development
  - Correctness or reliability
  - Security
  - Modularity
  - Modifiability
  - Readability and style
  - Quality of documentation
  - Robustness

Software Correctness and Reliability

- Programs must be correct. They often are not. When they are not, we can measure their relative correctness by their software reliability, which is a statistical assessment of the probability that they will be correct in an arbitrary run. Measuring reliability is a complicated matter, subject to much error.
- In general, software must undergo a rigorous development process, including testing, debugging, and verification when possible. Verification is hard; testing is something anyone can and should do.

- Verification is a mathematical proof that software is correct. In general, it is impossible to prove that software is correct, for the same reason that it is impossible to prove that a program is guaranteed never to enter an infinite loop. It is theoretically impossible. In spite of this, many program fragments and pieces of code can be proved to do what they are intended to do. This is through the use of pre-conditions, post-conditions, and loop invariants. All of these are logical statements involving the relationships of program variables to each other or to fixed constants. The following is a trivial example:

```
int x,y;
y = 0;
x = 2;
// y == 0 && x == 2     is an assertion that is true here
```

- Here is a slightly more interesting example:

```
x = 0;
cin >> n;
if ( n < 0 )
    n = -n;
// n >= 0      an assertion about n
while ( x < n )
    x = x + 1;
// x == n   an assertion about x and n
```

- After a loop it is always true that the loop entry condition is false, so its negation is always true!
- **Loop invariants** are special assertions that are true at specific points in the execution of the program. Specifically, a loop invariant is true

  1. Immediately preceding the evaluation of the loop entry condition before the loop is ever entered, and
  2. Immediately preceding the evaluation of the loop entry condition after each execution of the loop body.

- In other words, just at the point that the loop entry condition is to be evaluated, no matter when that happens, the loop invariant must be true, regardless of how many times the body has been executed. If an assertion is true at this point, it is a loop invariant.
- Some invariants are useful and others are useless. Here are examples:

```
int sum = 0;
int j = 0;
while ( j < n) //  j >= 0    true but not very useful
{
j++;
    sum = sum + j;
}
```

- It is true that j >= 0 but it does not help us in understanding what the loop does. This is better:

```
int sum = 0;
int j = 0;
while ( j < n) //  sum >= j    better but still not very useful
{
    j++;
    sum = sum + j;
}
```

- This is a little more interesting but still not useful enough. It is a little harder to prove that this is true. First, it is true before the loop is ever entered because `sum` and `j` are each 0. Second, if it is true when the loop is entered, it is true the next time the condition is evaluated because `sum` is always increased by the value of `j`, so it has to be at least `j`. This is what we really need:

```
int sum = 0;
int j = 0;

while ( j < n)  //  sum == 0 + 1 + 2 + ... + j
{
    j++;
    sum = sum + j;
}
```

- This one is strong enough to prove that the loop computes the *arithmetic sum* of the numbers from 1 to n. First we prove that the loop invariant is true by using mathematical induction on the number of times the loop body is entered.

  Proof. If the loop is never entered, then □

  `sum = j = 0`

  so it is true. Assume that it is true before the loop is entered the kth time. Then `j` has the value k-1 and by assumption,

  `sum = 1 + 2 + ... + (k-1).`

  After incrementing `j`, `j` has the value k. After the assignment to `sum`,

  `sum = 1 + 2 + 3 + ... + k`

  so

  `sum = 1 + 2 + ... + j`

  remains true. By induction on k it follows that it is true for all values of `j`. Since the loop invariant is true when the loop ends and since `j` == n when the loop ends, it follows that `sum` = 1 + 2 + ... + n.

The Interface as a Contract

- The **pre-condition** of a function is a logical condition that must be satisfied just prior to execution of the function.
- The **post-condition** of a function is a logical condition that must be true when the function terminates.
- Pre- and post-conditions together form a **contract** that states that if the pre-conditions are true, then after the function executes, the post-conditions will be true.
- The **signature** of a function is what some people call the prototype. It includes
  - return value type and qualifiers (e.g., const)
  - function name
  - argument list with types and qualifiers
- The interface to a class or a module is also a contract, if every function has been given pre-and post-conditions in the interface.

Example

First attempt at writing a contract for a sorting function:

```
sort(int anArray[], int num);
// Sorts an array.
// Precondition: anArray is an array of num  integers; num > 0.
// Postcondition: The integers in anArray are  sorted.
```

Second refinement at writing a contract for sorting function:

```
void sort(/*inout*/ int anArray[], /*in */ const int num);
// Sorts anArray into ascending order.
// Precondition: anArray is an initialized array of num integers;
// 1 <= num <= MAX_ARRAY, where
// MAX_ARRAY is a global constant that specifies
// the maximum size of anArray.
// Postcondition: anArray[0] <= anArray[1] <= ...
// <= anArray[num-1], num is unchanged.
```

Modularity

- The modularity of software is measured by its cohesion, coupling, and completeness.
- The **cohesion** of a module is the degree to which the parts of the module are related.
  - A highly cohesive module performs one well-defined task
  - High cohesion implies that it contains only methods and data that are related to its narrow purpose.
- The **coupling** of a set of modules is the degree to which the modules depend on each other.
  - Modules with low coupling are almost independent of one another
  - Low coupled modules are
    - Easier to change: A change to one module won't affect another
    - Easier to understand
  - Coupling cannot be and should not be eliminated entirely
- Classes should be easy to understand, and have as few methods as possible (idea of minimality). but should be **complete** – providing all methods that are necessary.

Documentation

- Documentation is critical for all programs, because
  - Other people have to read your program and understand it, including your supervisor or teacher or colleagues, or even a client.
  - You may have to revise a program you did a year ago and without good documentation you will not remember what it does or how it works.
  - Government and other agencies will require certain software to be properly documented. Software in various products requiring certifications needs proper documentation.
- What is proper documentation? It must include, at a minimum:

1. a preamble for every file, as described in the Programming Rules document.
2. pre-and post-conditions for all function prototypes, including descriptions of each parameter
3. descriptions of how error conditions are handled,
4. a revision history for every file (list of dates of modifications, who made them, and what they fixed or changed),
5. data dictionary (descriptions of most variables)
6. detailed descriptions of all complex algorithms.

- This is an example of a suitable preamble:

```
/*****************************************************************************
Title :       draw_stars.c
Author :      Stewart Weiss
Created on :  April 2, 2010
Description : Draws stars of any size in a window, by dragging the mouse to
              define the bounding rectangle of the star
Purpose :     Reinforces drawing with the backing-pixmap method, in which the
              application maintains a separate, hidden pixmap that gets drawn
              to the drawable only in the expose-event handler. Introduces the
              rubberbanding technique.
Usage :       draw_stars
              Press the left mouse button and drag to draw a 5-pointed star
Build with :  gcc -o drawing_demo_03 draw_stars.c \
                 `pkg-config --cflags --libs gtk+-2.0`
Modifications :
*****************************************************************************/
```

Readability and Style

- Programs should be **readable**. Factors contributing to readability include:
- The systematic use of white space (blank lines, space characters, tabs) to separate program sections and keep related code close together.
- A systematic identifier naming convention for program entities such as variables, classes, functions, and constants.
- Using lines of asterisks or other punctuation to group related code and separate code sections from each other.
- Indentation to indicate structural elements.

Other Stylistic Tips

- Use global constants for all program parameters and numeric and string constants, such as

```
int WINWIDTH  = 420 // width of our window
int WINHEIGHT = 400 // height of our window
```

- Use `define` macros for constants and functions also:

```
#define EPSILON 1.0e-15
#define ALMOSTZERO(x) (((-EPSILON)<(x))&&((x)<(EPSILON)))
```

- Try to design general solutions, not specific ones, to make functions very general.

Modifiability

- The modifiability of a program is the ease with which it can be modified. If a modification requires making changes to many parts of a program, it is not easy and can introduce errors when something is overlooked. A good program is one in which a simple modification requires only a change to one or two places in the program.
- Examples of things that make a program more modifiable include
  - Using named constants and macros instead of numeric and string literals.
  - Using `typedef` statements to define frequently used structured types.
  - Creating functions for common tasks.
  - Making modules very cohesive.
  - Keeping coupling low.

Ease of Use

- In an interactive program, the program should prompt the user for input in a clear manner.
- A program should always echo its input.
- The output should be well labeled and easy to read.

Robustness

- **Robust** (fail-safe) programs will perform reasonably no matter how anyone uses them. They do not crash. They do not produce nonsense, even when the user misuses them. Examples of things to do to make programs robust:
  - Test if input data, including program command line arguments, is invalid before trying to use it.
  - Prevent the user from entering invalid data when possible.
  - Check that all function parameters meet preconditions before using them.
  - Handle all error conditions that can arise, such as missing files, bad inputs, wrong values of variables, and so on, and exit gracefully when the program cannot continue.