

Software Engineering Principles

Software Engineering A disciplined approach to the design, production, and maintenance of computer programs that are developed on time and within cost estimates, using tools that help to manage the size and complexity of the resulting software products.

Software project life cycle

- *Problem analysis* Understanding the nature of the problem to be solved
- *Requirements definition* Specifying exactly what the program must do
- *High- and low-level design* Recording how the program meets the requirements, from the "big picture" overview to the detailed design
- *Implementation of the design* Coding a program in a computer language
- *Testing and verification* Detecting and fixing errors and demonstrating the correctness of the program
- *Delivery* Actually using the program
- *Maintenance* Making changes to fix operational errors and to add or modify functions of the program.

Goals of Quality Software

1. It works.
2. It can be read and understood.
3. It can be modified without excruciating time and effort.
4. It is completed on time and within budget.

Algorithm A logical sequence of discrete steps that describes a complete solution to a given problem computable in a finite amount of time.

More About Design

A part of the design is to simplify the problem. Divide a large problem into small manageable parts called modules (functions) which will be self-contained units of code. Modules should be as independent or loosely coupled as possible, except for their interfaces with other modules. Each module should perform one well-defined task; that is, it should be **highly cohesive**. This modularity describes a program that is organized into **loosely coupled**, highly cohesive modules. When the emphasis of your problem is on algorithms and not data, a top-down design will lead to a modular solution. The philosophy of a top-down design is that you should address a task at successively lower levels of detail. Consider the example of computing the median among a collection of test scores. We will build a **structure chart** to illustrate the hierarchy of and interaction among the modules that solve this problem. At first, each module is little more than a statement of what it needs to solve and is devoid of detail. You refine each module into smaller modules, which solve smaller problems and contain more detail on how to solve the problem. This process continues until the modules at the bottom of the hierarchy are simple enough to translate into isolated modules that solve very small, independent problems.

When you design a modular solution, each module begins as a box that states what it does, but not how it does it. No box may "know" how any other box performs its task---it may know only what that task is.

Abstraction separates the purpose of a module from its implementation. Modularity and abstraction complement each other. Modularity breaks a solution into modules; abstraction specifies each module clearly *before* you implement it in a programming language. Such specifications clarify the design of your solution.

As the problem-solving process proceeds, you gradually refine the boxes until eventually you can implement their actions in some programming language (C++ in our case). Each box typically becomes a C++ function in your program. You should separate the purpose of a function from its implementation. This process is known as **procedural abstraction**. Once a function is written, you can use it without knowing the particulars of its algorithm as long as you have a statement of its purpose and a description of its arguments.

Procedural abstraction is essential to team projects. After all, in a team situation, you will have to use functions written by others, frequently without knowledge of their algorithms. Actually, you do this all the time whenever you use a function from a standard library, such as *sqrt*.

Data Abstraction

Consider now a collection of data and a set of operations on the data. The operations might include ones that add new data to the collection, remove data from the collection, or search for some data. **Data abstraction** focuses on what the operations do instead of on how they will be implemented. The other modules of the solution will "know" *what* operations they can perform, but they will not know *how* the data are stored or *how* the operations are performed.

Most of this course is about data abstraction. To enable you to think abstractly about data---that is, to focus on what operations you will perform on the data instead of how you will perform---we must define an **abstract data type** or **ADT**. An ADT is a collection of data and a set of operations on that data. You can use an ADT's operations, if you know their specifications, without knowing how the operations are implemented or how the data are stored.

Consider the ordered list of integers 2, 7, 12, 15, 16, 19, 21 with the operations *insert*, *delete*, *find*.

Within problem solving, abstract data types support algorithms, and algorithms are part of what constitutes an abstract data type. As you design a solution to a problem, you should develop algorithms and ADTs in tandem.

Abstraction tells you to write, for each module, functional specifications that describe its outside, or public, view. However, abstraction also helps you to identify details that you should hide from public view---details that should not be in the specifications but should be **private**. The principle of **information hiding** tells you not only to hide such details within the module, but also to ensure that no other module can tamper with these hidden details.

Information hiding limits the ways in which you need to deal with functions and data. As a user of a module, you do not worry about the details of its implementation. As an implementer of a module, you do not worry about its uses.

Testing The process of executing a program with data sets designed to discover errors.

Debugging The process of removing known errors.

Assertion A logical proposition that can be true or false.

Preconditions Assertions that must be true on entry into an operation or function for the postconditions to be guaranteed.

Postconditions Assertions that state what results are to be expected at the exit of an operation or function, assuming the preconditions are true.

```
int FindSmallIndex(int A[], int size)

// Returns the index of the smallest element in the array, A.

// Pre:  size > 0, A contains valid data in A[0] to A[size - 1]
// Post: A value between 0 and size-1 is returned.  If there are
//       two "smallest" values, the smaller of the two indices is
//       returned.
```

Test plans

A test plan is a document showing the test cases planned for a program or module, their purposes, inputs, expected outputs, and criteria for success. Below is a test plan for the Divide module. (This is not a good test plan. We'll improve it in class.)

```
void Divide(int dividend, int divisor, bool & error, float & result)

// Pre: None
// Post: This function sets error to true if the divisor is 0 and does
//       not set result to any value.  If the divisor is not 0, error
//       will be false and result will be dividend divided by divisor.
```

Reason for test case	Input Values	Expected Output	Observed Output
divisor is zero	dividend = 8 divisor = 0	error = true result = anything	
divisor is nonzero	dividend = 8 divisor = 2	error = false result = 4.0	

Fail-Safe Programming

A fail-safe program is one that will perform reasonably no matter how anyone uses it. This is difficult to attain. A more realistic goal is to anticipate the ways that people might misuse the program and guard carefully against abuses.