Principles for Program Development

Principle 1 *A program and its proof should be developed hand-in-hand, with the proof usually leading the way.*

It is often difficult to prove that an existing program meets its specification. It is better to use proof-of-correctness ideas throughout the programming process.

Principle 2 *Use theory to provide insight; use common sense and intuition where it is suitable, but fall back on the formal theory when difficulties and complexities arise.*

Intuition and common sense are needed for successful programming. However, complete reliance upon them often leads to poorly designed, error-plagued programs. Formal proofs of correctness put programming upon a firmer foundation, but excessive attention to formalism may lead to incomprehensible detail. A programmer needs a balance between theory and intuition. Obvious facts should be left implicit, important points should be stressed, and detail should be presented to allow the reader to understand a program as easily as possible. But, when the going gets rough, more formalism is required. Maintaining this balance requires intelligence, taste, knowledge, and PRACTICE.

I assume that the students in this class are adept at applying intuition, common sense, and knowledge of computer technology in the development of programs, but are less adept at applying formal methods. To overcome this imbalance, we will perhaps be more formal than may be required in practical programming situations.

Principle 3 *Know the properties of the objects that are to be manipulated by the program.*

The more properties you know about the objects, the greater chance you have of creating an efficient (and elegant) algorithm.

Principle 4 *Never dismiss as obvious any fundamental principle, for it is only through conscious application of such principles that success can be achieved.*

Ideas may be simple and easy to understand, but their application may require effort. Recognizing a principle and applying it are two different things.

Principle 5 *Before attempting to solve a problem, make absolutely sure that you know what the problem is.*

Programming corollary: Before developing a program, make the precondition and the postcondition precise, and refine them to give insight into possible computational strategies.

A problem is sometimes specified in a way that lends itself to several interpretations. Hence, it is reasonable to spend some time making the specification as clear and unambiguous as possible. Moreover, the form of the specification can influence algorithmic development, so that striving for simplicity and elegance should be helpful.

Principle 6 *Programming is a goal-oriented activity.*

The postcondition (i.e, the desired result or goal) plays a more important role in the development of a program than the precondition. The precondition is used, but the postcondition usually gives us more insight. (The weakest precondition calculus was developed to accommodate the goal-oriented nature of programming—supports working backward from the postcondition.)

Principle 7 *The solution of a complex problem is not something that can be understood all at once; elements of it must be grasped over a period of time and fitted into the overall framework.*

A solution to a complex problem usually requires the identification and solution of a set of subproblems. These solutions can then be combined and adapted to construct a solution to the original problem. Possible strategies:

- Solve a set of subproblems, each of which solves the original problem for some of the initial states (i.e., for some stronger precondition).
- Solve a sequence of intermediate problems, the composition of which is equivalent to the original problem. (The problems in the sequence, except the last, have postconditions weaker than than the original problem.)
- Solve a "harder" problem, that is, a problem with a stronger postcondition than the original problem. The stronger specification often allows a more efficient solution to be found.
- Solve an equivalent problem, that is, reformulate the problem in some way that is easier to manipulate and solve.

Principle 8 *Don't stop working on a problem when you have found a solution—reexamine, reconsider, and reflect upon the solution.*

No problem is ever completely exhausted. There always remains something to do; with sufficient study, we can improve almost any solution, and, in any case, we can always improve our understanding of the solution and our problem-solving methodology. Perhaps a more elegant, more general, or more efficient solution can be found—or perhaps a better derivation. What steps did you go through in the process of solving the problem? Have you discovered some general technique that can be applied to other similar problems?

Caveat: Although reconsideration of solutions is a "good thing," one we too often omit when faced with a deadline, excessive concern for perfection is sometimes counter-productive. Every good engineer—whether of software or of hardware—must learn to accept solutions that are "good enough" to meet the constraints of the problem and move on to other problems that need solutions. As the old saying goes, don't be "penny wise and pound foolish."

Acknowledgement: The above principles and comments are adapted from Gries' *The Science of Programming* (Springer-Verlag, 1981), Dromey's *Program Derivation: The Development of Programs from Specifications* (Addison-Wesley, 1989), Polya's *How to Solve It* (Princeton University Press, 1957), and other sources.