[an error occurred while processing this directive]

# Principles of Software Engineering

## Separation of Concerns

Separation of concerns is a recognition of the need for human beings to work within a limited context. As descibed by G. A. Miller [Miller56], the human mind is limited to dealing with approximately seven units of data at a time. A unit is something that a person has learned to deal with as a whole - a single abstraction or concept. Although human capacity for forming abstractions appears to be unlimited, it takes time and repetitive use for an abstraction to become a useful tool; that is, to serve as a unit.

When specifying the behavior of a data structure component, there are often two concerns that need to be dealt with: basic functionality and support for data integrity. A data structure component is often easier to use if these two concerns are divided as much as posible into separate sets of client functions. It is certainly helful to clients if the client documentation treats the two concerns separately. Further, implementation documentation and algorithm descriptions can profit from separate treatment of basic algorithms and modifications for data integrity and exception handling.

There is another reason for the importance of separation of concerns. Software engineers must deal with complex values in attempting to optimize the quality of a product. From the study of algorithmic complexity, we can learn an important lesson. There are often efficient algorithms for optimizing a single measurable quantity, but problems requiring optimization of a combination of quantities are almost always NP-complete. Although it is not a proven fact, most experts in complexity theory believe that NP-complete problems cannot be solved by algorithms that run in polynomial time.

In view of this, it makes sense to separate handling of different values. This can be done either by dealing with different values at different times in the software development process, or by structuring the design so that responsibility for achieving different values is assigned to different components.

As an example of this, run-time efficiency is one value that frequently conflicts with other software values. For this reason, most software engineers recommend dealing with efficiency as a separate concern. After the software is design to meet other criteria, it's run time can be checked and analysed to see where the time is being spent. If necessary, the portions of code that are using the greatest part of the runtime can be modified to improve the runtime. This idea is described in depth in Ken Auer and Kent Beck's article "Lazy optimization: patterns for efficient smalltalk programming" in [VCK96, pp 19-42].

## Modularity

The principle of modularity is a specialization of the principle of separation of concerns. Following the principle of modularity implies separating software into components according to functionality and responsibility. Parnas [Parnas72] wrote one of the eariest papers discussing the considerations involved in modularization. A more recent work, [WWW90], describes a responsibility-driven methodology for modularization in an object-oriented context.

## Abstraction

The principle of abstraction is another specialization of the principle of separation of concerns. Following the principle of abstraction implies separating the behavior of software components from their implementation. It requires learning to look at software and software components from two points of view: what it does, and how it does it.

Failure to separate behavior from implementation is a common cause of unnecessary coupling. For example, it is common in recursive algorithms to introduce extra parameters to make the recursion work. When this is done, the recursion should be called through a non-recursive shell that provides the proper initial values for the extra parameters. Otherwise, the caller must deal with a more complex behavior that requires specifying the extra parameters. If the implementation is later converted to a non-recursive algorithm then the client code will also need to be changed.

Design by contract is an important methodology for dealing with abstraction. The basic ideas of design by contract are sketched by Fowler and Scott [FS97]. The most complete treatment of the methodology is given by Meyer [Meyer92a].

## Anticipation of Change

Computer software is an automated solution to a problem. The problem arises in some context, or *domain* that is familiar to the users of the software. The domain defines the types of data that the users need to work with and relationships between the types of data.

Software developers, on the other hand, are familiar with a technology that deals with data in an abstract way. They deal with structures and algorithms without regard for the meaning or importance of the data that is involved. A software developer can think in terms of graphs and graph algorithms without attaching concrete meaning to vertices and edges.

Working out an automated solution to a problem is thus a learning experience for both software developers and their clients. Software developers are learning the domain that the clients work in. They are also learning the values of the client: what form of data presentation is most useful to the client, what kinds of data are crucial and require special protective measures.

The clients are learning to see the range of possible solutions that software technology can provide. They are also learning to evaluate the possible solutions with regard to their effectiveness in meeting the clients needs.

If the problem to be solved is complex then it is not reasonable to assume that the best solution will be worked out in a short period of time. The clients do, however, want a timely solution. In most cases, they are not willing to wait until the perfect solution is worked out. They want a reasonable solution soon; perfection can come later. To develop a timely solution, software developers need to know the requirements: how the software should behave. The principle of acticipation of change recognizes the complexity of the learning process for both software developers and their clients. Preliminary requirements need to be worked out early, but it should be possible to make changes in the requirements as learning progresses.

Coupling is a major obstacle to change. If two components are strongly coupled then it is likely that changing one will not work without changing the other.

Cohesiveness has a positive effect on ease of change. Cohesive components are easier to reuse when requirements change. If a component has several tasks rolled up into one package, it is likely that it will need to be split up when changes are made.

## Generality

The principle of generality is closely related to the principle of anticipation of change. It is important in designing software that is free from unnatural restrictions and limitations. One excellent example of an unnatural restriction or limitation is the use of two digit year numbers, which has led to the "year 2000" problem: software that will garble record keeping at the turn of the century. Although the two-digit limitation appeared reasonable at the time, good software frequently survives beyond its expected lifetime.

For another example where the principle of generality applies, consider a customer who is converting business practices into automated software. They are often trying to satisfy general needs, but they understand and present their needs in terms of their current practices. As they become more familiar with the possibilities of automated solutions, they begin seeing what they need, rather than what they are currently doing to satisfy those needs. This distinction is similar to the distinction in the principle of abstraction, but its effects are felt earlier in the software development process.

## Incremental Development

Fowler and Scott [FS97] give a brief, but thoughtful, description of an incremental software development process. In this process, you build the software in small increments; for example, adding one use case at a time.

An incremental software development process simplifies verification. If you develop software by adding small increments of functionality then, for verification, you only need to deal with the added portion. If there are any errors detected then they are already partly isolated so they are much easier to correct.

A carefully planned incremental development process can also ease the handling of changes in requirements. To do this, the planning must identify use cases that are most likely to be changed and put them towards the end of the development process.

## Consistency

The principle of consistency is a recognition of the fact that it is easier to do things in a familiar context. For example, coding style is a consistent manner of laying out code text. This serves two purposes. First, it makes reading the code easier. Second, it allows programmers to automate part of the skills required in code entry, freeing the programmer's mind to deal with more important issues.

At a higher level, consistency involves the development of idioms for dealing with common programming problems. Coplien [Coplien92] gives an excellent presentation of the use of idioms for coding in C++.

Consistency serves two purposes in designing graphical user interfaces. First, a consistent look and feel makes it easier for users to learn to use software. Once the basic elements of dealing with an inteface are learned, they do not have to be relearned for a different software application. Second, a consistent user interface promotes reuse of the interace components. Graphical user interface systems have a collection of frames, panes, and other view components that support the common look. They also have a collection of controllers for responding to user input, supporting the common feel. Often, both look and feel are combined, as in pop-up menus and buttons. These components can be used by any program.

Meyer [Meyer94c] applies the principle of consistency to object-oriented class libraries. As the available libraries grow more and more complex it is essential that they be designed to present a consistent interface to the client. For example, most data collection structures support adding new data items. It is much easier to learn to use the collections if the name `add` is always used for this kind of operation.

[an error occurred while processing this directive]