Principles of Software Engineering[1]

Separation of Concerns

Separation of concerns is an acknowledgement for people working within a limited context. As described by G. A. Miller [Miller56], the human mind is limited to dealing with approximately seven (7) units of data at a time. A unit is a measurement of something that a person has learned to deal with as a whole -- a single abstraction or concept. Although the 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: 1) the basic functionality and 2) the support for data integrity. A data structure component is often easier to use if these two concerns are divided, as much as possible, into separate client functions sets. It is certainly helpful if the client documentation treats the two concerns separately. Furthermore, software implementation documents and algorithm annotations are enhanced when using the separate treatment of basic algorithms and modifications for data integrity and exception handling.

Another reason for the importance of separation of concerns presents itself. Software engineers must deal with complex interrelationship when optimizing a product's quality. From algorithmic complexity analysis, we learn an important lesson. There are often efficient optimizing algorithms for a single measurable quantity, but problems requiring optimization of a combination of quantities are almost always NP-complete (NP meaning "**nondeterministic polynomial time**"). Although it is not a proven fact, most experts in computational 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 the handling of different values. This can be done either by dealing with different values at different times during the software development process or by structuring the architecture so that the 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 designed to meet other criteria, it's runtime can be checked and analyzed to see where the computational processing is spent. If necessary, the portions of the code, that are using the greatest part of the runtime, could be modified to improve their runtime. This idea is described in depth in Ken Auer "Lazy optimization: patterns for efficient Smalltalk programming".

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

[1] Original copy from https://www.d.umn.edu/~gshute/softeng/principles.html edited and adapted for game software engineering.

functionality and responsibility. Single Responsibility Principle, 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: 1) what it does, and 2) how it does it.

Failure to separate behavior from the software implementation is a common cause of unnecessary coupling. For example, it is common in recursive algorithms to introduce extra parameters to assist the recursion operation. When this is done, the recursion should be called through a non-recursive shell that provides the proper initialization values for those extra parameters. Otherwise, the caller function 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 require changes.

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

Anticipation of Change

Computer software is an automated solution to a given problem. The problem arises in some context or “domain” that is familiar to the users of the software. A domain defines the relationships between the types of data with which users are required to work.

Software developers, on the other hand, are familiar with a technology that deals with data in an abstract way. They deal with data structures and algorithms without regard for interpretation of the involved data. A software developer thinks in terms of graphic representations and graphing algorithms without attaching translations 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 in which the clients work. They are also learning those items their client values: what manner of data presentation is most beneficial for their client, what kinds of information are crucial and require special security measures.

The clients are exposed to the range of software technology solutions possible. They are also learning to evaluate those solutions regarding their business operations effectiveness and business drivers.

If the problem is complex, then it is an irrational assumption an optimal solution could be discovered in a short period of time. The clients do, however, want a timely solution. In most cases, they are not willing to wait until a perfect solution is presented. They want a reasonable solution soon; perfection can come later. To develop a timely solution, software developers need to know the prioritized requirements: how the software should behave. The principle of “anticipation of change” encompasses the complexity of the discovery 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 functional package, it is likely that it will be deconstructed 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 "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 like 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 newly added portions. If errors are 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 an acknowledged fact that it is easier to do things in a familiar context. For example, coding style is the consistent manner of laying and formatting source code text. This serves two purposes. Firstly, it aids in easily reading the code. Second, it allows automation of code entry for programmers, thus freeing their minds to deal with more abstract 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. Firstly, a consistent look and feel makes facilitates users' learning of the software tools and Integrated

Development Environments (IDE). Once familiarity of dealing with an interface is achieved, they do not have to be relearned for a different software application. Second, a consistent user interface promotes reuse of the interface 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 the appearance is combined, as a pop-up menus or 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 are designed to present a consistent interface. 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.