**Separation of Concerns**

Separation of concerns allows us to deal with different aspects of a problem, so that we can concentrate on each individually. Separation of concerns is a commonsense prac­tice that we try to follow in our everyday life to overcome the difficulties we encounter. The principle should be applied also in software development, to master its inherent complexity.

More specifically, there are many decisions that must be made in the develop­ment of a software product. Some of them concern features of the product: functions to offer, expected reliability, efficiency with respect to space and time, the product's relationship with the environment (i.e., the special hardware or software resources required), user interfaces, etc. Others concern the development process: the develop­ment environment, the organization and structure of teams, scheduling, control procedures, design strategies, error recovery mechanisms, etc. Still others concern economic and financial matters. These different decisions may be unrelated to one another. In such a case, it is obvious that they should be treated separately.

Very often, however, many decisions are strongly related and interdependent.

For instance, a design decision (e.g., swapping some data from main memory to disk) may depend on the size of the memory of the selected target machine (and hence, the cost of the machine), and this, in turn, may affect the policy for error recovery. When different design decisions are strongly interconnected, it would be useful to take all the issues into account at the same time and by the same people, but this is not usually pos­sible in practice.

The only way to master the complexity of a project is to separate the different concerns. First of all, we should try to isolate issues that are not so closely related to the others. Then, we consider issues separately, together with only the *relevant* details of related issues.

There are various ways in which concerns may be separated. First, one can separate them in *time*. As an everyday example, a university professor might apply separation of concerns by scheduling teaching-related activities such as holding classes, seminars, office hours, and department meetings from 9 A.M. to 2 PM. Monday through Thursday; consulting on Friday; and engaging in research the rest of the time. Such temporal sepa­ration of concerns allows for the precise planning of activities and eliminates overhead that would arise through switching from one activity to another in an unconstrained way. Separation of concerns in terms of time is the underlying motivation of the software life cycle models, each of which defines a sequence of activities that should be followed in software production.

Another type of separation of concerns is in terms of *qualities* that should be treated separately. For example, in the case of software, we might wish to deal sepa­rately with the efficiency and the correctness of a given program. One might decide first to design software in such a careful and structured way that its correctness is expected to be guaranteed a *priori* and then to restructure the program partially to improve its efficiency. Similarly, in the verification phase, one might first check the functional correctness of the program and then its performance. Both activities can be done rigorously, applying some systematic procedures, or even formally (i.e., using for­mal correctness proofs and complexity analysis).

Another important type of separation of concerns allows different *views* of the software to be analyzed separately. For example, when we analyze the requirements of an application, it may be helpful to concentrate separately on the flow of data from one activity to another in the system and the flow of control that governs the way dif­ferent activities are synchronized. Both views help us understand the system we are working on better, although neither one gives a complete view of it.

Still another type of separation of concerns allows us to deal with *parts* of the same system separately; here, separation is in terms of size. This is a fundamental con­cept that we need to master to dominate the complexity of software production. Indeed, it is so important that we prefer to detail it shortly as a separate point under modularity.

There is an inherent disadvantage in separation of concerns: By separating two or more issues, we might miss some global optimization that would be possible by tackling them together. While this is true in principle, our ability to make "optimized" decisions in face of complexity is rather limited. If we consider too many concerns simultaneously, we are likely to be overwhelmed by the amount of detail and complexity we face. Some of the most important decisions in design concern which aspects to consider together and which separately. System designers and architects often face such trade-offs.

Note that if two issues associated with one problem are intrinsically intertwined (i.e., the problem is not immediately decomposable into separate issues), it is often possi­ble to make some overall design decisions first and then effectively separate the different issues. For example, consider a system in which on-line transactions access a database concurrently. In a first implementation of the system, we could introduce a simple lock­ing scheme that requires each transaction to lock the entire database at the start of the transaction and unlock it at the end. Suppose now that a preliminary performance analy­sis shows that some transaction, say, ti (which might print some complex report extract­ing many data from the database), takes so long than we cannot afford to have the database unavailable to other transactions. Thus, the problem is to revise the implemen­tation to improve its performance yet maintain the overall correctness of the system. Clearly, the two issues-functional correctness and performance-are strongly related, so a first design decision must concern both of them: ti is no longer implemented as an atomic transaction, but is split into several subtransactions ti1, ti2, …, tin, each of which is atomic itself. The new implementation may affect the correctness of the system, because of the interleaving that may occur between the executions of any two sub trans­actions. Now, however, we have separated the two concerns of checking the functional correctness of the system and analyzing its performance; we may, then, do the analyses independently, possibly even by two different designers with different levels of expertise.

Perhaps the most important application of separation of concerns is to separate problem-domain concerns from implementation-domain concerns. Problem-domain properties hold in general, regardless of the implementation environment. For exam­ple, in designing a personnel-management system, we must separate issues that are true about employees in general from those which are a consequence of our imple­mentation of the employee as a data structure or object. In the problem domain, we may speak of the relationship between employees, such as "employee A reports to employee B," and in the implementation domain we may speak of one object pointing to another. These concerns are often intermingled in many projects.

As a final remark, notice that separation of concerns may result in separation of responsibilities in dealing with separate issues. Thus, the principle is the basis for divid­ing the work on a complex problem into specific assignments, possibly for different people with different skills. For example, by separating managerial and technical issues in the software process, we allow two types of people to cooperate in a software pro­ject. Or, having separated requirements analysis and specification from other activities in a software life cycle, we may hire specialized analysts with expertise in the applica­tion domain, instead of relying on internal resources. The analyst, in turn, may concen­trate separately on functional and nonfunctional system requirements.

Ghezzi, C., Jazayeri, M., and Mandrioli, D. (2002), *Fundamentals of Software Engineering,* Prentice Hall.

**Modularity**

A complex system may be divided into simpler pieces called *modules*. A system that is composed of modules is called *modular*. The main benefit of modularity is that it allows the principle of separation of concerns to be applied in two phases: when deal­ing with the details of each module in isolation (and ignoring details of other modules) and when dealing with the overall characteristics of all modules and their relationships in order to integrate them into a coherent system. If the two phases are executed in sequence first by concentrating on modules and then on their composition, then we say that the system is designed *bottom up*; the converse-when we decompose modules first and then concentrate on individual module design-is *top-down* design.

Modularity is an important property of most engineering processes and products. For example, in the automobile industry, the construction of cars proceeds by assem­bling building blocks that are designed and built separately. Furthermore, parts are often reused from model to model, perhaps after minor changes. Most industrial processes are essentially modular, made out of work packages that are combined in simple ways (sequentially or overlapping) to achieve the desired result.

Modularity, however, is not only a desirable design principle; it also permeates the whole of software production. In particular, modularity provides four main benefits in practice:

1. the capability of decomposing a complex system into simpler pieces,
2. the capability of composing a complex system from existing modules,
3. the capability of understanding the system in terms of its pieces, and
4. the capability of modifying a system by modifying only a small number of its pieces.

The *decomposability* of a system is based on dividing the original problem top down into subproblems and then applying the decomposition to each subproblem recursively. This procedure reflects the well-known Latin motto *divide et impera* (divide and conquer), which describes the philosophy followed by the ancient Romans to dom­inate other nations: Divide and isolate them first, and then conquer them individually.

The *composability* of a system is based on starting bottom up from elementary components and combining them in steps towards finally producing a finished sys­tem. As an example, a system for office automation may be designed by assembling together existing hardware components, such as personal workstations, a network, and peripherals; system software, such as the operating system; and productivity tools, such as document processors, databases, and spreadsheets. A car is another obvious example of a system that is built by assembling components. Consider first the main subsystems into which a car may be decomposed: the body, the electrical system, the power system, the transmission system, etc. Each of them, in turn, is made out of standard parts; for example, the battery, fuses, cables, etc., form the electrical system. When something goes wrong, defective components may be replaced by new ones.

Ideally, in software production we would like to be able to assemble new applica­tions by taking modules from a library and combining them to form the required prod­uct. Such modules should be designed with the express goal of being reusable. By using reusable components, we may speed up both the initial system construction and its fine-tuning. For example, it would be possible to replace a component by another that performs the same function, but differs in computational resource requirements.

The *capability of understanding* and modifying a system are related to each other as understanding is often the first step to applying modifications. We have stressed evolvability as a quality goal because software engineers are often required to go back to previous work to modify it. If the entire system can be understood only in its entirety, modifications are likely to be difficult to apply, and the result will probably be unreliable. When it is necessary to repair a defect or enhance a feature, proper modu­larity helps confine the search for the fault or enhancement to single components. Modularity thus forms the basis for software evolution.

To achieve modular composability, decomposability, understandability, and mod­ifiability, the software engineer must design the modules with the goal of *high cohesion* and *low coupling.*

A module has high cohesion if all of its elements are related strongly. Elements of a module (e.g., statements, procedures, and declarations) are grouped together in the same module for a logical reason, not just by chance: They cooperate to achieve a common goal, which is the function of the module.

Whereas cohesion is an internal property of a module, coupling characterizes a module's relationship to other modules. Coupling measures the interdependence of two modules. If two modules depend on each other heavily, they have high coupling. Ideally, we would like modules in a system to exhibit low coupling, because it will then be possible to analyze, understand, modify, test, or reuse them sep­arately.

A good example of a system that has high cohesion and low coupling is the elec­tric subsystem of a house. Because it is made out of a set of appliances with clearly definable functions and interconnected by simple wires, the system has low coupling. Because each appliance's internal components are there exactly to provide the service the appliance is supposed to provide, the system has high cohesion.

Modular structures with high cohesion and low coupling allow us to see modules as black boxes when the overall structure of a system is described and then deal with each module separately when the module's functionality is described or analyzed. In other words, modularity supports the application of the principle of separation of concerns.

Ghezzi, C., Jazayeri, M., and Mandrioli, D. (2002), *Fundamentals of Software Engineering,* Prentice Hall.

**Abstraction**

Abstraction is a fundamental technique for understanding and analyzing complex problems. In applying abstraction, we identify the important aspects of a phenomenon and ignore its details. Thus, abstraction is a special case of separation of concerns wherein we separate the concern of the important aspects from the concern of the less important details.

What we abstract and consider as a detail that may be ignored depends on the purpose of the abstraction. For example, consider a digital watch. A useful abstrac­tion for the owner is a description of the effects of pushing its various buttons, which allow the watch to enter various modes of functioning and react differently to sequences of commands. A useful abstraction for the person in charge of maintaining the watch is a box that can be opened in order to replace the battery. Still other abstractions of the device are useful for understanding the watch and performing the activities that are needed to repair it (let alone design it). Thus, there may be many different abstractions of the same reality, each providing a view of the reality and serving some specific purpose.

Abstraction is a powerful technique practiced by engineers of all fields for mas­tering complexity. For example, the representation of an electrical circuit in terms of resistors and capacitors, each characterized by a set of equations, is an idealized abstraction of a device. The equations are a simplified model that approximates the behavior of the real components.

This example illustrates an important general idea: The models we build of phe­nomena-e.g., the equations for describing devices-are an abstraction from reality, ignoring certain facts and concentrating on others that we believe are relevant. The same holds for the models built and analyzed by software engineers. For example, when the requirements for a new application are analyzed and specified, software engineers build a model of the proposed application. This model may be expressed in various forms, depending on the required degree of rigor and formality. No matter what language we use for expressing requirements-be it natural language or the formal language of mathematical formulas-what we provide is a model that abstracts away from a number of details that we decide can be ignored safely.

Abstraction permeates the whole of programming. The programming languages that we use are abstractions built on top of the hardware: They provide us with useful and powerful constructs so that we can write (most) programs ignoring such details as the number of bits that are used to represent numbers or the specific computer's addressing mechanism. This helps us concentrate on the solution to the problem we are trying to solve, rather than on the way to instruct the machine on how to solve it. The programs we write are themselves abstractions. For example, a computerized pay­roll procedure is an abstraction of the manual procedure it replaces: It provides the essence of the manual procedure, not its exact details.

As an example of the use of abstraction in software processes, consider the case of cost estimation for a new application. One possible way of doing cost estimation consists of identifying some key factors of the new system-for example, the number of engineers on the project and the expected size of the final system-and extrapo­lating from the cost profiles of previous similar systems. The key factors used to per­form the analysis are an abstraction of the system for the purpose of cost estimation.

Ghezzi, C., Jazayeri, M., and Mandrioli, D. (2002), *Fundamentals of Software Engineering,* Prentice Hall.

**Generality**

The principle of generality may be stated as follows:

Every time you are asked to solve a problem, try to focus on the discovery of a more gen­eral problem that may be hidden behind the problem at hand. It may happen that the gen­eralized problem is not more complex-indeed, it may even be simpler-than the original problem. Moreover, it is likely that the solution to the generalized problem has more potential for being reused. It may even happen that the solution is already provided by some off-the-shelf package. Also, it may happen that, by generalizing a problem, you end up designing a module that is invoked at more than one point of the application, rather than having several specialized solutions.

On the other hand, a generalized solution may be more costly, in terms of speed of execution, memory requirements, or development time, than the specialized solu­tion that is tailored to the original problem. Thus, it is necessary to evaluate the trade­offs of generality with respect to cost and efficiency, in order to decide whether it is worthwhile to solve the generalized problem instead of the original problem.

For example, suppose you are asked to merge two sorted sequential files into one. On the basis of the requirements, you know that the two source files do not con­tain any records with identical key values. Clearly, then, if you generalize your solution to accept source files that may contain different elements with the same key value, you provide a program that has a higher potential for reusability. Also, you may be able to use a merge program that is available in your system library.

As another example, suppose that you are asked to design an application to handle a small library of cooking recipes. Suppose also that the recipes have a header-containing information such as a name, a list of ingredients, and cooking information-and a textual part describing how to apply the recipes. Apart from storing recipes in the library, it must be possible to do a sophisticated search for recipes, based on their available ingredients, maximum calories, etc. Rather than designing a new set of facilities, these searches can be viewed as a special case of a more general set of text-processing facilities. Before starting with the design of the specialized set of routines, the designer should consider whether the adoption of a generalized text-processing tool would be more useful. The generalized tool is undoubtedly more reliable than the specialized program to be designed, and it would probably accommodate changes in the requirements or even new requirements. On the negative side, however, there may be a cost of acquisition, and possibly overhead, in the use of the generalized tool.

Ghezzi, C., Jazayeri, M., and Mandrioli, D. (2002), *Fundamentals of Software Engineering,* Prentice Hall.

**Anticipation of Change**

Software undergoes changes constantly. Changes are due both to the need for repairing the software-eliminating errors that were not detected before releasing the application-and to the need for supporting evolution of the application as new requirements arise or old requirements change. This is why we identified main­tainability as a major software quality.

The ability of software to evolve does not happen by accident or out of sheer luck-it requires a special effort to anticipate how and where changes are likely to occur. Designers can try to identify likely future changes and take special care to make these changes easy to apply. Basically, likely changes should be isolated in specific portions of the software in such a way that changes will be restricted to such small portions. In other words, anticipation of change is the basis for our modularization strategy.

Anticipation of change is perhaps the one principle that distinguishes software the most from other types of industrial productions. In many cases, a software applica­tion is developed when its requirements are not entirely understood. Then, after being released, on the basis of feedback from the users, the application must evolve as new requirements are discovered or old requirements are updated. In addition, applica­tions are often embedded in an environment, such as an organizational structure. The environment is affected by the introduction of the application, and this generates new requirements that were not known initially. Thus, anticipation of change is a principle that we can use to achieve evolvability.

Reusability is another software quality that is strongly affected by anticipation of change. As we saw, a component is reusable if it can be directly used to produce a new product. More realistically, the component might undergo some changes before it can be reused. Hence, reusability may be viewed as low-grain evolvability-that is, evolv­ability at the component level. If we can anticipate the changes of context in which a software component might be embedded, we may then design the component in a way that such changes may be accommodated easily.

Anticipation of change requires that appropriate tools be available to manage the various versions and revisions of the software in a controlled manner. It must be possible to store and retrieve documentation, source modules, object modules, etc., from a database that acts as the central repository of reusable components. Access to the database must be controlled. A consistent view of the software system must always be available, even as we apply changes to some of its components. The discipline that studies this class of problems is called *configuration management*.

In our discussion of anticipation of change, we focused attention more on soft­ware products than on software processes. Anticipation of change, however, also affects the management of the software process. For example, managers should anticipate the effects of personnel turnover. Also, when the life cycle of an application is designed, it is important to take maintenance into account. Depending on the anticipated changes, managers must estimate costs and design the organizational structure that will support the evolution of the software. Finally, managers should decide whether it is worthwhile investing time and effort in the production of reusable components, either as a by-prod­uct of a given software development project or as a parallel development effort.

Ghezzi, C., Jazayeri, M., and Mandrioli, D. (2002), *Fundamentals of Software Engineering,* Prentice Hall.

**Incrementality**

Incrementality characterizes a process that proceeds in a stepwise fashion, in increments. We try to achieve the desired goal by successively closer approximations to it. Each approximation is an increment over the previous one.

Incrementality applies to many engineering activities. When applied to software, it means that the desired application is produced as a result of an evolutionary process.

One way of applying the incrementality principle consists of identifying useful early subsets of an application that may be developed and delivered to customers, in order to get early feedback. This allows the application to evolve in a controlled man­ner in cases where the initial requirements are not stable or fully understood. The motivation for incrementality is that in most practical cases there is no way of getting all the' requirements right before an application is developed. Rather, requirements emerge as the application-or parts of it-is available for practical experimentation. Consequently, the sooner we can receive feedback from the customer concerning the usefulness of the application, the easier it is to incorporate the required changes into the product. Thus, incrementality is intertwined with anticipation of change and is one of the cornerstones upon which evolvability may be based.

We may progressively add functions to the application being developed, starting from a kernel of functions that would still make the system useful, although incomplete. For example, in some business automation systems, some functions would still be done manually, while others would be done automatically by the application.

We can also add performance in an incremental fashion. That is, the initial ver­sion of the application might emphasize user interfaces and reliability more than per­formance, and successive releases would then improve space and time efficiency.

When an application is developed incrementally, intermediate stages may con­stitute prototypes of the end product; that is, they are just an approximation of it. The idea of rapid prototyping is often advocated as a way of progressively develop­ing an application hand in hand with an understanding of its requirements. Obviously, a software life cycle based on prototyping is rather different from the typical waterfall model described earlier, wherein we first do a complete require­ments analysis and specification and then start developing the application. Instead, prototyping is based on a more flexible and iterative development model. This dif­ference affects not only the technical aspects of projects, but also the organizational and managerial issues.

As we mentioned in connection with anticipation of change, evolutionary soft­ware development requires special care in the management of documents, programs, test data, etc., developed for the various versions of software. Each meaningful incre­mental step must be recorded, documentation must be easily retrieved, changes must be applied in a controlled way, and so on. If these are not done carefully, an intended evolutionary development may quickly turn into undisciplined software development, and all the potential advantages of evolvability will be lost.

Ghezzi, C., Jazayeri, M., and Mandrioli, D. (2002), *Fundamentals of Software Engineering,* Prentice Hall.