Software quality

EXTERNAL AND INTERNAL FACTORS

External quality factors are properties such as speed or ease of use, whose presence or absence in a software product may be detected by its users. Other qualities applicable to a software product, such as being modular, or readable, are internal factors, perceptible only to computer professionals who have access to the actual software text.

In the end, only external factors matter, but the key to achieving these external factors is in the internal ones: for the users to enjoy the visible qualities, the designers and implementers must have applied internal techniques that will ensure the hidden qualities.

EXTERNAL FACTORS

Correctness: The ability of software products to perform their tasks, as defined by their specification.

Correctness is the prime quality. If a system does not do what it is supposed to do, everything else about it — whether it is fast, has a nice user interface… — matters little. But this is easier said than done. Even the first step to correctness is already difficult: we must be able to specify the system requirements in a precise form, by itself quite a challenging task.

Robustness: The ability of software systems to react appropriately to abnormal conditions.

Robustness complements correctness. Correctness addresses the behavior of a system in cases covered by its specification; robustness characterizes what happens outside of that specification. There will always be cases that the specification does not explicitly address. The role of the robustness requirement is to make sure that if such cases do arise, the system does not cause catastrophic events; it should produce appropriate error messages, terminate its execution cleanly, or enter a so-called “graceful degradation” mode.

Extendibility: The ease of adapting software products to changes of specification.

Traditional approaches to software engineering did not take enough account of change, relying instead on an ideal view of the software lifecycle where an initial analysis stage freezes the requirements, the rest of the process being devoted to designing and building a solution.

Change is pervasive in software development: change of requirements, of our understanding of the requirements, of algorithms, of data representation, of implementation techniques. Support for change is a basic goal of object technology.

Two principles are essential for improving extendibility:

• *Design simplicity*: a simple architecture will always be easier to adapt to changes than a complex one.
• *Decentralization*: the more autonomous the modules, the higher the likelihood that a simple change will affect just one module, or a small number of modules, rather than triggering off a chain reaction of changes over the whole system.

The object-oriented method is, before anything else, a system architecture method which helps designers produce systems whose structure remains both simple (even for large systems) and decentralized.

Reusability: The ability of software elements to serve for the construction of many different applications.

The need for reusability comes from the observation that software systems often follow similar patterns; it should be possible to exploit this commonality and avoid reinventing solutions to problems that have been encountered before. By capturing such a pattern, a reusable software element will be applicable to many different developments.

Compatibility: The ease of combining software elements with others.

Compatibility is important because we do not develop software elements in a vacuum: they need to interact with each other. But they too often have trouble interacting because they make conflicting assumptions about the rest of the world. An example is the wide variety of incompatible file formats supported by many operating systems. A program can directly use another's result as input only if the file formats are compatible. Lack of compatibility can yield disaster

The key to compatibility lies in homogeneity of design, and in agreeing on standardized conventions for inter-program communication. Approaches include:

• Standardized file formats, as in the Unix system, where every text file is simply a sequence of characters.

• Standardized data structures, as in Lisp systems, where all data, and programs as well, are represented by binary trees (called lists in Lisp).

• Standardized user interfaces, as on various versions of Windows, OS/2 and MacOS, where all tools rely on a single paradigm for communication with the user, based on

standard components such as windows, icons, menus etc.

More general solutions are obtained by defining standardized access protocols to all important entities manipulated by the software. This is the idea behind abstract data types and the object-oriented approach

Efficiency: The ability of a software system to place as few demands as possible on hardware resources, such as processor time, space occupied in internal and external memories, bandwidth used in communication devices.

The constant improvement in computer power, impressive as it is, is not an excuse for overlooking efficiency.

Software construction is difficult precisely because it requires taking into account many different requirements, some of which, such as correctness, are abstract and conceptual, whereas others, such as efficiency, are concrete and bound to the properties of computer hardware.

Efficiency is only one of the factors of quality; we should not (like some in the profession) let it rule our engineering lives. But it is a factor, and must be taken into consideration, whether in the construction of a software system or in the design of a programming language. If you dismiss performance, performance will dismiss you.

Portability: The ease of transferring software products to various hardware and software environments.

Ease of use: The ease with which people of various backgrounds and qualifications can learn to use software products and apply them to solve problems. It also covers the ease of installation, operation and monitoring.

Functionality: The extent of possibilities provided by a system.

One of the most difficult problems facing a project leader is to know how much functionality is enough. The pressure for more facilities constantly there. Its consequences are bad for internal projects, where the pressure comes from users within the same company, and worse for commercial products, as the most prominent part of a journalist's comparative review is often the table listing side by side the features offered by competing products.

Featurism is actually the combination of two problems, one more difficult than the other. The easier problem is the loss of consistency that may result from the addition of new

features, affecting its ease of use. The more difficult problem is to avoid being so focused on features as to forget the other qualities.

Timeliness: The ability of a software system to be released when or before its users want it.

Other qualities

Other qualities beside the ones discussed so far affect users of software systems and the people who purchase these systems or commission their development. In particular: Verifiability, Integrity, Repairability, and Economy.

ABOUT DOCUMENTATION

In a list of software quality factors, one might expect to find the presence of good documentation as one of the requirements. But this is not a separate quality factor; instead, the need for documentation is a consequence of the other quality factors seen above. We may distinguish between three kinds of documentation:

• The need for *external* documentation, which enables users to understand the power of a system and use it conveniently, is a consequence of the definition of ease of use.

• The need for *internal* documentation, which enables software developers to understand the structure and implementation of a system, is a consequence of the extendibility requirement.

• The need for *module interface* documentation, enabling software developers to understand the functions provided by a module without having to understand its implementation, is a consequence of the reusability requirement. It also follows from extendibility, as module interface documentation makes it possible to determine whether a certain change need affect a certain module.

TRADEOFFS

In this review of external software quality factors, we have encountered requirements that may conflict with one another. How can one get *integrity* without introducing protections of various kinds, which will inevitably hamper *ease of use*? *Economy* often seems to fight with *functionality*. Optimal *efficiency* would require perfect adaptation to a particular hardware and software environment, which is the opposite of *portability*, and perfect adaptation to a specification, where *reusability* pushes towards solving problems more

general than the one initially given. *Timeliness* pressures might tempt us to use "Rapid Application Development" techniques whose results may not enjoy much *extendibility*.

Although it is in many cases possible to find a solution that reconciles apparently conflicting factors, you will sometimes need to make tradeoffs. Too often, developers make these tradeoffs implicitly, without taking the time to examine the issues involved and the various choices available; efficiency tends to be the dominating factor in such silent decisions. A true software engineering approach implies an effort to state the criteria clearly and make the choices consciously.

Necessary as tradeoffs between quality factors may be, one factor stands out from the rest: correctness. There is never any justification for compromising correctness for the sake of other concerns such as efficiency. If the software does not perform its function, the rest is useless.

KEY CONCERNS

All the qualities discussed above are important. But in the current state of the software industry, four stand out:

• **Correctness and robustness**: it is still too difficult to produce software without defects (bugs), and too hard to correct the defects once they are there. Techniques for improving correctness and robustness are of the same general flavors: more systematic approaches to software construction; more formal specifications; built-in checks throughout the software construction process (not just after-the-fact testing and debugging); better language mechanisms such as static typing, assertions, automatic memory management and disciplined exception handling, enabling developers to state correctness and robustness requirements, and enabling tools to detect inconsistencies before they lead to defects. Because of this closeness of correctness and robustness issues, it is convenient to use a more general term, **reliability**, to cover both factors.

• **Extendibility and reusability**: software should be easier to change; the software elements we produce should be more generally applicable, and there should exist a larger inventory of general-purpose components that we can reuse when developing a new system. Here again, similar ideas are useful for improving both qualities: any idea that helps produce more decentralized architectures, in which the components are self-contained and only communicate through restricted and clearly defined channels, will help. The term **modularity** will cover reusability and extendibility.

The object-oriented method can significantly improve these four quality factors — which is why it is so attractive. It also has significant contributions to make on other aspects, in particular:

• *Compatibility*: the method promotes a common design style and standardized module and system interfaces, which help produce systems that will work together.

• *Portability*: with its emphasis on abstraction and information hiding, object technology encourages designers to distinguish between specification and implementation properties, facilitating porting efforts. The techniques of polymorphism and dynamic binding will even make it possible to write systems that automatically adapt to various components of the hardware-software machine, for example different window systems or different database management systems.

• *Ease of use*: the contribution of O-O tools to modern interactive systems and especially their user interfaces is well known, to the point that it sometimes obscures other aspects (ad copy writers are not the only people who call "object-oriented" any system that uses icons, windows and mouse-driven input).

• *Efficiency*: as noted above, although the extra power or object-oriented techniques at first appears to carry a price, relying on professional-quality reusable components can often yield considerable performance improvements.

• *Timeliness*, *economy* and *functionality*: O-O techniques enable those who master them to produce software faster and at less cost; they facilitate addition of functions, and may even of themselves suggest new functions to add.

KEY CONCEPTS

• The purpose of software engineering is to find ways of building quality software.

• Rather than a single factor, quality in software is best viewed as a tradeoff between a set of different goals.

• External factors, perceptible to users and clients, should be distinguished from internal factors, perceptible to designers and implementors.

• What matters is the external factors, but they can only be achieved through the internal factors.

• A list of basic external quality factors was presented. Those for which current software is most badly in need of better methods, and which the object-oriented method directly addresses, are the safety-related factors correctness and robustness, together known as reliability, and the factors requiring more decentralized software architectures: reusability and extendibility, together known as modularity.

• Software maintenance, which consumes a large portion of software costs, is penalized by the difficulty of implementing changes in software products, and by the over-dependence of programs on the physical structure of the data they manipulate.

Modularity

From the goals of extendibility and reusability follows the need for flexible system architectures, made of autonomous software components.

Modular programming was once taken to mean the construction of programs as assemblies of small pieces, usually subroutines. But such a technique cannot bring real extendibility and reusability benefits unless we have a better way of guaranteeing that the resulting pieces — the **modules** — are self-contained and organized in stable architectures. Any comprehensive definition of modularity must ensure these properties.

Five Criteria: A design method worthy of being called "modular" should satisfy five fundamental requirements:

- Decomposability
- Composability
- Understandability.
- Continuity.
- Protection.

Modular decomposability

> **A software construction method satisfies Modular Decomposability if it helps in the task of decomposing a software problem into a small number of less complex subproblems, connected by a simple structure, and independent enough to allow further work to proceed separately on each of them**

A corollary of the decomposability requirement is *division of labor*: once you have decomposed a system into subsystems you should be able to distribute work on these subsystems among different people or groups. This is a difficult goal since it limits the dependencies that may exist between the subsystems:

• You must keep such dependencies to the bare minimum; otherwise the development of each subsystem would be limited by the pace of the work on the other subsystems.

• The dependencies must be known: if you fail to list all the relations between subsystems, you may at the end of the project get a set of software elements that appear to work individually but cannot be put together to produce a complete system satisfying the overall requirements of the original problem.

The most obvious *example* of a method meant to satisfy the decomposability criterion is **top-down design**. This method directs designers to start with a most abstract description of the system's function, and then to refine this view through successive steps, decomposing each subsystem at each step into a small number of simpler subsystems, until all the remaining elements are of a sufficiently low level of abstraction to allow direct implementation.

A typical *counter-example* is any method encouraging you to include, in each software system that you produce, a global initialization module. Many modules in a system will need some kind of initialization — actions such as the opening of certain files or the initialization of certain variables, which the module must execute before it performs its first directly useful tasks. It may seem a good idea to concentrate all such actions, for all modules of the system, in a module that initializes everything for everybody. However, this method would endanger the autonomy of modules: you will have to grant the initialization module authorization to access many separate data structures, belonging to the various modules of the system and requiring specific initialization actions. This means that the author of the initialization module will constantly have to peek into the internal data structures of the other modules, and interact with their authors. This is incompatible with the decomposability criterion.

In the object-oriented method, every module will be responsible for the initialization of its own data structures.

Modular composability

> **A method satisfies Modular Composability if it favors the production of software elements which may then be freely combined with each other to produce new systems, possibly in an environment quite different from the one in which they were initially developed.**

Where decomposability was concerned with the derivation of subsystems from overall systems, composability addresses the reverse process: extracting existing software elements from the context for which they were originally designed, so as to use them again in different contexts.

A modular design method should facilitate this process by yielding software elements that will be sufficiently autonomous — sufficiently independent from the immediate goal that led to their existence — as to make the extraction possible.

Composability is directly connected with the goal of reusability: the aim is to find ways to design software elements performing well-defined tasks and usable in widely different contexts. This criterion reflects an old dream: transforming the software design process into a construction box activity, so that we would build programs by combining standard prefabricated elements.

• *Example 1*: *subprogram libraries.*
• *Example 2*: *Unix Shell conventions.*

Composability is independent of decomposability. In fact, these criteria are often at odds. Top-down design, for example, which we saw as a technique favoring decomposability, tends to produce modules that are *not* easy to combine with modules coming from other sources. This is because the method suggests developing each module to fulfill a specific requirement, corresponding to a subproblem obtained at some point in the refinement process. Such modules tend to be closely linked to the immediate context that led to their development, and unfit for adaptation to other contexts. The method provides neither hints towards making modules more general than immediately required, nor any incentives to do so; it helps neither avoid nor even just detect commonalities or redundancies between modules obtained in different parts of the hierarchy. That composability and decomposability are both part of the requirements for a modular method reflects the inevitable mix of top-down and bottom-up reasoning

Modular understandability

> **A method favors Modular Understandability if it helps produce software in which a human reader can understand each module without having to know the others, or, at worst, by having to examine only a few of the others**

The importance of this criterion follows from its influence on the maintenance process. Most maintenance activities, whether of the noble or not-so-noble category, involve having to dig into existing software elements. A method can hardly be called modular if a reader of the software is unable to understand its elements separately.

This criterion, like the others, applies to the modules of a system description at any level: analysis, design, implementation.

• *Counter-example*: *sequential dependencies.*

Modular continuity

> **A method satisfies Modular Continuity if, in the software architectures that it yields, a small change in a problem specification will trigger a change of just one module, or a small number of modules**

This criterion is directly connected to the general goal of extendibility. Continuity means that small changes should affect individual modules in the structure of the system, rather than the structure itself.

• *Example 1*: *symbolic constants*.
• *Example 2*: *the Uniform Access principle*. Another rule states that a single notation should be available to obtain the features of an object, whether they are represented as data fields or computed on demand. This property is sufficiently important to warrant a separate discussion later in this chapter.
• *Counter-example 1*: *using physical representations*.
• *Counter-example 2*: *static arrays*.

Modular protection

> **A method satisfies Modular Protection if it yields architectures in which the effect of an abnormal condition occurring at run time in a module will remain confined to that module, or at worst will only propagate to a few neighboring modules.**

The underlying issue, that of failures and errors, is central to software engineering. The errors considered here are run-time errors, resulting from hardware failures, erroneous input or exhaustion of needed resources (for example memory storage). The criterion does not address the avoidance or correction of errors, but the aspect that is directly relevant to modularity: their propagation.

• *Example*: *validating input at the source*.
• *Counter-example*: *undisciplined exceptions*.

FIVE RULES

From the preceding criteria, five rules follow which we must observe to ensure modularity:

- Direct Mapping.
- Few Interfaces.
- Small interfaces (weak coupling).
- Explicit Interfaces.
- Information Hiding.

The first rule addresses the connection between a software system and the externals systems with which it is connected; the next four all address a common issue — how

modules will communicate. Obtaining good modular architectures requires that communication occur in a controlled and disciplined way.

Direct Mapping

Any software system attempts to address the needs of some problem domain. If you have a good model for describing that domain, you will find it desirable to keep a clear correspondence (mapping) between the structure of the solution, as provided by the software, and the structure of the problem, as described by the model. Hence the first rule:

> **The modular structure devised in the process of building a software system should remain compatible with any modular structure devised in the process of modeling the problem domain.**

This advice follows in particular from two of the modularity criteria:

• Continuity: keeping a trace of the problem's modular structure in the solution's structure will make it easier to assess and limit the impact of changes.

• Decomposability: if some work has already been done to analyze the modular structure of the problem domain, it may provide a good starting point for the modular decomposition of the software.

Few Interfaces

The Few Interfaces rule restricts the overall number of communication channels between modules in a software architecture:

> **Every module should communicate with as few others as possible.**

Communication may occur between modules in a variety of ways. Modules may call each other (if they are procedures), share data structures etc. The Few Interfaces rule limits the number of such connections.

Small Interfaces

The Small Interfaces or "Weak Coupling" rule relates to the size of intermodule connections rather than to their number:

If two modules communicate, they should exchange as little information as Possible

Explicit Interfaces

With the fourth rule, we go one step further in enforcing a totalitarian regime upon the society of modules: not only do we demand that any conversation be limited to few participants and consist of just a few words; we also require that such conversations must be held in public and loudly!

Whenever two modules *A* and *B* communicate, this must be obvious from the text of *A* or *B* or both.

Behind this rule stand the criteria of decomposability and composability (if you need to decompose a module into several submodules or compose it with other modules, any outside connection should be clearly visible), continuity (it should be easy to find out what elements a potential change may affect) and understandability (how can you understand *A* by itself if *B* can influence its behavior in some devious way?).

One of the problems in applying the Explicit Interfaces rule is that there is more to intermodule coupling than procedure call; data sharing, in particular, is a source of indirect coupling: Assume that module *A* modifies and module *B* uses the same data item *x*. Then *A* and *B* are in fact strongly coupled through *x* even though there may be no apparent connection, such as a procedure call, between them.

Information Hiding

The rule of Information Hiding may be stated as follows:

The designer of every module must select a subset of the module's properties as the official information about the module, to be made available to authors of client modules.

The fundamental reason behind the rule of Information Hiding is the continuity criterion. Assume a module changes, but the changes apply only to its secret elements, leaving the public ones untouched; then other modules who use it, called its *clients*, will not be affected. The smaller the public part, the higher the chances that changes to the module will indeed be in the secret part. We may picture a module supporting Information Hiding as an iceberg; only the tip — the interface — is visible to the clients.

Information hiding implies that users of a procedure should be independent of the particular implementation chosen. That way client modules will not suffer from any change in implementation. Information hiding emphasizes separation of function from implementation. Besides continuity, this rule is also related to the criteria of decomposability, composability and understandability. You cannot develop the modules of a system separately, combine various existing modules, or understand individual modules, unless you know precisely what each of them may and may not expect from the others.

More precisely, it should be impossible to write client modules whose correct functioning depends on secret information

FIVE PRINCIPLES

From the preceding rules, and indirectly from the criteria, five principles of software construction follow:

- The Linguistic Modular Units principle.
- The Self-Documentation principle.
- The Uniform Access principle.
- The Open-Closed principle.
- The Single Choice principle.

Linguistic Modular Units: Modules must correspond to syntactic units in the language used

The Linguistic Modular Units principle expresses that the formalism used to describe software at various levels (specifications, designs, implementations) must support the view of modularity retained: The language mentioned may be a programming language, a design language, a specification language etc. In the case of programming languages, modules should be separately compilable.

Self-Documentation principle: The designer of a module should strive to make all information about the module part of the module itself.

Uniform Access: All services offered by a module should be available through a uniform notation, which does not betray whether they are implemented through storage or through computation.

Although it may at first appear just to address a notational issue, the Uniform Access principle is in fact a design rule which influences many aspects of object-oriented design and the supporting notation. It follows from the Continuity criterion; you may also view it as a special case of Information Hiding.

The Open-Closed principle: Modules should be both open and closed

The contradiction between the two terms is only apparent as they correspond to goals of a different nature:
• A module is said to be open if it is still available for extension..
• A module is said to be closed if it is available for use by other modules.

Single Choice: Whenever a software system must support a set of alternatives, one and only one module in the system should know their exhaustive list.

By requiring that knowledge of the list of choices be confined to just one module, we prepare the scene for later changes: if variants are added, we will only have to update the module which has the information — the point of single choice. All others, in particular its clients, will be able to continue their business as usual.

Once again, traditional methods do not provide a solution; once again, object technology will show the way, here thanks to two techniques connected with inheritance: polymorphism and dynamic binding. No sneak preview in this case, however; these techniques must be understood in the context of the full method.

The Single Choice principle prompts a few more comments:

• The number of modules that know the list of choices should be, according to the principle, exactly one. The modularity goals suggest that we want *at most one* module to have this knowledge; but then it is also clear that *at least one* module must possess it. You cannot write an editor unless at least one component of the system has the list of all supported commands, or a graphics system unless at least one component has the list of all supported figure types, or a Pascal compiler unless at least one component "knows" the list of Pascal constructs.

• Like many of the other rules and principles studied in this chapter, the principle is about **distribution of knowledge** in a software system. This question is indeed crucial to the search for extendible, reusable software. To obtain solid, durable system architectures you must take stringent steps to limit the amount of information available to each module. By analogy with the methods employed by certain human organizations, we may

call this a **need-to-know** policy: barring every module from accessing any information that is not strictly required for its proper functioning.

KEY CONCEPTS

• The choice of a proper module structure is the key to achieving the aims of reusability and extendibility.

• Modules serve for both software decomposition (the top-down view) and software composition (bottom-up).

• Modular concepts apply to specification and design as well as implementation.

• A comprehensive definition of modularity must combine several perspectives; the various requirements may sometimes appear at odds with each other, as with decomposability (which encourages top-down methods) and composability (which favors a bottom-up approach).

• Controlling the amount and form of communication between modules is a fundamental step in producing a good modular architecture.

• The long-term integrity of modular system structures requires information hiding, which enforces a rigorous separation of interface and implementation.

• Uniform access frees clients from internal representation choices in their suppliers.

• A closed module is one that may be used, through its interface, by client modules.

• An open module is one that is still subject to extension.

• Effective project management requires support for modules that are both open and closed. But traditional approaches to design and programming do not permit this.

• The principle of Single Choice directs us to limit the dissemination of exhaustive knowledge about variants of a certain notion.