ADDENDUM Z Software quality

In previous chapters, we mentioned the difficulty involved in dealing with the evolving nature and complexity of a software system. Our approach to software design and construction is largely dictated by the goal of managing complexity while producing a system that can be easily adapted to changing needs. In essence, we want to produce “quality software.” In this chapter, we step back from our detailed discussion of system development and take a brief look at what we mean by “quality software.” Specifically, we enumerate some of the important properties we want our systems to have. Of particular importance are the attributes of correctness and maintainability: that is, the system must do what it is supposed to do, and be able to be modified easily when necessary. We also identify some of the general characteristics our designs must exhibit if the systems we build are to achieve these standards of quality.

z.1 External qualities

The qualities of a software system that are our ultimate goal – the qualities that the user is concerned with – are sometimes called *external qualities*. The attributes of a system that help achieve these external qualities are *internal qualities*. For example, the external qualities important in an automobile are things like reliability, safety, fuel efficiency, speed, good handling, smooth ride, *etc*. These are what the customer – the user of the automobile – is concerned with. To achieve a smooth ride, the designers might attempt to minimize unsprung weight. Low unsprung weight is an internal quality. The customer is not particularly interested in unsprung weight: the customer wants a smooth ride. Minimizing unsprung weight helps achieve the external quality that the customer wants.

Some of the attributes we want systems to have are quantitative and measurable; others are qualitative. With an automobile, for instance, fuel efficiency is a well-defined, empirically measurable notion. (Of course, what counts as *acceptable* fuel efficiency is relative and qualitative.) On the other hand, safety is a less precise, relative notion. To deal with such qualitative ideas, we generally define a set of measurable attributes that we expect will corollate closely with the qualitative attribute. For example, to “measure” automobile safety, we might design a set of crash tests and carefully specify a measurable set of criteria that constitute acceptable performance in these tests.

What are the important external qualities of a software system – the attributes that are important to the user? Probably the most obvious is *correctness*. A system is correct if it behaves as expected in all cases; that is, if it *conforms to its specifications*. This is not such a simple idea as it first seems. To begin with, we must first know what it is supposed to do in all possible circumstances, completely and precisely. This is given by the system's *functional specifications*. But developing a complete, precise, and consistent set of specifications for a large system in no trivial matter. Furthermore, assuming we have an adequate specification, how do we tell if the system satisfies it? We can test the system, but we cannot conceivably test all situations. Verifying that a systems satisfies a set of specifications is also not an easy matter.

A second quality that usually comes to mind is *efficiency*. By efficiency, we sometimes mean the amount of memory space required to run the system, but more often we mean the time it takes for a system to perform a particular function. For some systems, called *real-time systems*, response time, the time required for a system to respond to some stimulus, is a matter of correctness. Think of a small computer in an anti-lock brake system. When the system detects that a wheel is not rotating, it must respond by releasing the brake. If the time between detecting the wheel lock and releasing the brake is too long, the vehicle will skid. That is, *correctness* requires that the system respond within a specified period of time. As another example, imagine a computer controlling a deceleration thruster on a planetary lander. When the lander comes within a certain distance of a planet's surface, the thruster must be fired to slow the lander. Again, if the system does not respond within a carefully specified amount of time, disaster will result.

For most systems, though, efficiency is a relative matter. If we click a button that instructs the system to balance this month's accounts, it probably doesn't matter to us whether it takes a half second or five seconds. On the other hand, we would not find it acceptable if five seconds elapsed between the time we pressed a key and time the character appeared on the screen.

Another important quality is *ease of use* or *user friendliness*. This is a very relative, hard to measure notion. Indeed, what qualifies as "easy to use" for one person can be annoyingly constraining for another, and impossibly obscure for a third. We will not attempt to discuss the many measures, tests, and principles that have been suggested in this area.

A key aspect of ease of use is *robustness*. A system is robust if it behaves "reasonably" when encountering unexpected circumstances or incorrect input. Suppose, for instance, a system asks for a user's age (expecting an integer), and the user keys in his name instead. A system that crashes in this situation is not very robust. We would like the system to help the user along, perhaps with a more detailed explanation of what was expected, and give him a chance to try again. (An old adage, however, says that it is impossible to make a system idiot-proof because idiots are too damned clever.)

A system should be *maintainable*. It might seem a bit odd to use the term "maintenance" in regard to software. After all, software doesn't "wear out" or "break down" like an automobile or washing machine. Software maintenance generally refers to activities that modify a system after the system has been put into production. Maintenance activities can be classified according to the objective of the modifications:

- *corrective*: taken to correct errors detected in the system;
- *adaptive*: taken to adapt the system to changes in the environment or in the user's requirements;
- *perfective*: taken to improve the quality of the system.

Even with thoroughly tested software, bugs – programming errors – are occasionally discovered after the system is put in use. Some errors go undiscovered for years, showing up only when some very unusual circumstance occurs. One aspect of software maintenance involves tracking down and correcting errors discovered in a production system. A second aspect has to do with the fact that software exists in a dynamic environment. As needs change, the software must be modified and extended. (Software that can be easily adapted to changing circumstances is sometimes called *extendible*.) Modifying production systems to meet evolving specifications is another important part of software maintenance.

We should note that software maintenance is a rather substantive issue. It is generally accepted that about 70% of software cost is due to maintenance, and studies have indicated that the upwards of 40% of maintenance costs result from changes in user requirements. The approach to software design we adopt is rooted in the goal of making the system maintainable: in particular, we want to build systems for which a small change in requirements necessitates only a small change in the system.

Though we could extend this list of desirable attributes for many pages, we'll mention just a couple more. *Portability* measures the ease with which a software system can be transferred to run on different kinds of computers and under different operating systems – "different platforms." The advantage here is that the user is not confined to a particular vendor, and can take advantage of price/performance advances in the technology.

Compatibility means that one software system is able to work with others. For instance, if we use a particular computer aided design (CAD) system to produce technical drawings, a spreadsheet to do cost analysis, and publishing software to create reports, we may well want to incorporate CAD drawings and spreadsheet graphs in a report created with the publishing software. This can be trivial or almost impossible, depending on the compatibility of the three software products.

Note that these qualities are not necessarily implied by the functional requirements of the system. For instance, two distinct implementations of the same set of functional requirements may differ dramatically efficiency, maintainability, portability, and so on.

z.2 Complexity

Now that we've listed a few of the external qualities we'd like our software to have, we'll take a very brief look at some of the internal qualities that have proven indispensable in achieving these goals. It might be difficult, at this stage, to appreciate the significance of these points fully. In fact, the points themselves may be a bit obscure and hard to comprehend. Nevertheless, we feel obligated at least to sketch the goals toward which so much of our approach is directed, even if their import is not yet completely evident.

As we have emphasized, complexity is the principal obstacle to the creation of correct, reliable, maintainable software. Software systems are, of course, inherently complex. A program performs millions of actions a second, each of which must be specified at some level by system designers. A good design, however, can reduce complexity considerably. We must build our systems from components that are simple enough to be comprehensible. The problem is that the number of possible interactions between components grows as the square of the number of components. Thus an important key to managing complexity is to control the interactions between components. Components should be as self-contained as possible; interactions should be minimal and carefully defined.

z.3 Modularity

To say that a software system is *modular* is simply to say that it is made up of components: *i.e.*, of *modules*. This is somewhat tautological: every piece of software is in some way made up of pieces. To achieve the benefits we seek in managing complexity, modules must be self-contained and coherent. What, specifically, does this mean?

In the first place, we'd like the logical modularity of our design to be supported by the programming language and development system. That is, the logical components of the design should correspond to physically separate syntactic software components. We'd like to be able to manage these software components independently during system development. For instance, we'd like to be able to keep separate components in separate files, have different programmers develop and test them, and so forth. If we are not able to reflect the logical structure of the system with the physical structure of the program, we're starting with a considerable handicap.

Cohesion is a measure of the degree to which a module *completely* encapsulates a *single* notion. Note that there are two aspects to this definition: a module has to do with only one notion, and it in some sense completely represents it. When we say "notion," we mean a facet of the system's functionality or data. If we're designing functionally, we want a module to describe a single function of the system completely – for instance, the operation of producing a student's transcript in a student record system. (This is sometimes referred to as *functional strength*.) If our design is object oriented, we want a module to describe a single kind of data object completely. For instance, an object representing a student in a student record system should completely encapsulate the data and functionality associated with the student, and nothing else.

Coupling is a measure of the degree to which a module interacts with and depends on other modules. That is, it is a measure of inter-module dependency. Clearly, a module that is part of a system must have something to do with other parts of the system. But as we stated above, the number of possible interactions between components grows in proportion to the square of the number of components. If every module in a system can directly affect every other, the complexity of the system is bound to be unmanageable. We want a module to interact with as few other modules as possible, and we want the interaction to be through a minimal, well-defined interface. If two modules need to exchange data, for instance, they should exchange only what is absolutely necessary. Interactions between modules should be clearly defined: one module should not depend on another in subtle and unclear ways. To get a better idea of what we mean, consider a computer workstation and the diesel generator that provides its electrical power. Both of these are very complex physical systems. But their interface is through a simple, well-defined, standard mechanism. The power generator provides electricity with certain well defined characteristics to a receptacle of specific configuration, and the computer has a power cord that plugs into the receptacle.

Clearly, we want to design modules with *high cohesion* and *weak coupling*. If our modules are well-designed, then affecting a small change in the system should require modification of only a few closely related modules. As we've mentioned, experience has shown that this can be best accomplished by organizing the modular structure of the system around the data rather than around functionality. Furthermore, well-defined modules will localize the effects of run-time errors, simplifying the maintenance process. For example, a module responsible for getting input data might also be responsible for validating that the data is reasonable and has the proper format. Thus bad data doesn't spread through the system, causing hard to trace errors in disparate parts of the system. Problems from the bad data are constrained to the input module, where they can be more easily identified and addressed.

The modular construction of a hardware system, in which extensive use is made of standard components from a parts catalogue, is often contrasted with the construction of a typical software system, where components are hand-crafted from scratch. There would be clear advantages, in terms of cost and reliability, to having a collection of reliably correct software components that could be used in the construction of new systems. Much of the focus of object-oriented methodologies is toward creating *reusable software modules*. By "reusable software components," we mean something a little different from prefabricated blocks we can plug together. While it's handy to have a library of standard pieces that can be plugged into commonly occurring situations, a better description of what we want to achieve might be "tailorable software components." Like the basic black shift that we can take up or let down, appoint with a collar or belt, and generally adjust to suit a myriad of situations, we'd like modules that we can get off the rack and use in our system with only a few minor alterations. Note that the same qualities that make a module easy to adjust for use in a new system, also make it easy to adjust to meet changing needs in an existing system.

z.4 Three principles

Before we complete our discussion of software quality, we present three design principles, articulated by Bertrand Meyer in his excellent text *Object Oriented Software Construction* [Meyer 88, 97]. These are the *information hiding principle*, the *principle of continuity*, and the *open-closed principle*. We'll explore these ideas and their consequences thoroughly in the remainder of the text.

Let's suppose that you and I work for a large university in the days when records were kept manually. You work in the Student Records Office, and I work in Financial Aid and

Scholarships. In order to determine who should be awarded particular scholarships, I need to get student data from your files. To simplify and standardize this procedure, a form has been devised that I use to request information. For instance, I may request a list of all juniors in Mathematics with a grade point average greater than 3.5. Now whether or not this works well depends on a number of circumstances: how well the form has been designed, how responsive your staff is to my requests, how reasonable I am in my expectations, and so forth. But our interface is simple and well-defined. It doesn't matter if you happen to be out for a day; any clerk in the Records Office can satisfy my request, and I don't particularly care who does. Also if I happen to get a better offer from the school across town and leave, it shouldn't take my replacement long to figure out how to get data from the Records Office.

Now suppose that your files are in an accessible location, and I get tired of using the standard procedures. When I need some information, I rummage through your files myself. The more I use the files, the more details I learn about them. For instance, I discover that there are no files for Math majors in the green cabinet against the wall. I might even begin to make my own notes on your files. What I've done is create a set of rather subtle unspecified dependencies between what I do and your file system. In software terms, I've violated the rule that modules should be weakly coupled: that is, have minimal interaction through a well-defined interface. An innocent change on your part – for instance, you move some Math files to the green cabinet – can cause the system to fail.

A principle that helps minimize the interdependency of modules is the *information hiding principle*. Basically this rule says that the only information one module should be able to obtain about another is that required by the well-defined interface between them. In our example, this means that you shouldn't give me direct access to your files. Then I can't depend on any particular organization of your records. If you change the way you handle student records, it cannot possibly effect me as long as you continue to satisfy my formal requests.

Suppose that a situation arises where cultural considerations need to be taken into account in the awarding of certain scholarships. The Lithuanian Society, for instance, funds several scholarships to encourage students of Lithuanian descent to pursue engineering degrees. It may well be the case that student records do not include information such as national heritage. Adding this kind of information should be a relatively minor change in the system. One would not expect, for instance, that the Accounting Department would have to take notice of such a change. The *principle of continuity* states that a small change in the specifications of a system should result in a small change in the system. In particular, a small change should effect only a few modules in the system, and not the entire structure of the system itself. When each module needs to know details of all the principle data objects, or when knowledge about any specific data item is spread throughout the system, a small change will propagate through the entire system. This is exactly what occurs when the system design is organized around functionality. As we have mentioned a number of times, structuring the system around data rather than around functionality measurably enhances a system's continuity.

A module that can still be modified is said to be *open*. A module that is available for use by other modules is said to be *closed*. We would like to design and build our systems so that the modules are both open and closed: the *open-closed principle*. The idea is that we would like to be able to modify or extend a module without effecting other modules – even those that interact with the one being modified – that are not directly concerned with the modification. In the context of our example, we would like to be able to modify student records to include information regarding national heritage without this change effecting the Accounting Department. That is, the Accounting Department should be able to continue using information from student records without taking notice that the records have been modified to include additional information. Clearly, the open-closed principle is closely related to the continuity principle discussed above.

z.5 What is a software system, revisited

We addressed the question of what we meant by a software system in the first chapter. Now that we've taken a look at some of the attributes we want our systems to have, we return to this question and offer three views of what software should be. We want to consider these views as defining criteria for evaluating our software systems and for guiding our developmental efforts.

z.5.1 Software = a temporary solution to an evolving problem

When we build software systems, we cannot simply look for specific solutions to specific problems. We must produce solutions which are amenable to change. Appreciating the fact that malleability must be an essential attribute of our software helps identify fundamental activities that should be part of the design and implementation process. Specifically, we must *design* a system that adequately models the problem as currently perceived, yet *implement* the system in such a way as to minimize dependency on the particularities of the problem at hand.

These two seemingly contradictory goals are addressed through *object design* and *algorithm design*. Object design produces the collection of objects that not only model the problem at hand, but also provide a set of fixed conceptual constituents in terms of which system evolution can be expressed. Object design also defines how the objects interact to solve the problem at hand. Algorithm design involves providing the detailed functionality of the objects. Put simply, as the system changes we expect the behavior of the individual objects to change, and we expect the way in which the objects interact to change. But we expect the set of objects in terms of which the system is defined to remain stable.

You should note that this view of software is intimately related to the principles outlined in the previous section. From this perspective, the quality of a model used to produce a solution depends on how successfully the stable components, as objects, are identified and isolated from those aspects of the problem likely to change. A successful approach segments the most volatile aspects of the system, so that change and modification can be localized.

The term “temporary” is a key one to keep in mind. Problems and their solutions are not rigid, well-defined, well-behaved. Version 1.0 is never the final product. By qualifying a solution as temporary, we intend to encourage a perspective in which emphasis will be placed on identifying those fundamental components, central to the problem, that serve as a supporting framework for an evolving system.

z.5.2 Software = data + algorithms

Niklaus Wirth introduced the formula “Algorithms + Data Structures = Programs” in his 1975 book of that name [Wirth 75]. Wirth’s point was that since every program handles data, attention must be paid to how the representation of the data can affect the algorithm. Developments in object-oriented methods have given a twist to Wirth’s equation. Experience has shown that data is the fundamental and most stable aspect of a software system. Not only must we isolate data representation as a separate concern, we must carefully design the abstract view of the data and the relationships between component data objects. What we mean to say is that “software = design of data representation followed by algorithm design.” The design of an abstract view of the data, followed by the implementation of this abstract view by means of algorithms, is the essence of programming.

z.5.3 Software = English document and a mathematical document

Though software is executed by machines, it is written and maintained by people. This view emphasizes the dual character of software. A software solution must be read and understood by anyone intending to maintain it. As such, it can be seen as a technical document describing the design and implementation of the solution. We can legitimately apply evaluation criteria appropriate to any document, such as readability, ease of accessing pertinent information, and so on. Though at the moment we are necessarily concerned with the details of how to express our problem solutions in a programming language, we should keep in mind that what we write must be clear and comprehensible to a human reader.

Furthermore, the correctness of the solution described by the software should be verifiable in a formal, mathematical sense. That is, the correctness of our solution should follow from the program in much the same sense that the correctness of a mathematical theorem follows from its proof. This is not an easy idea to grasp, and a complete exposition is beyond the scope of this text. But as we describe the software development process in subsequent chapters, we will illustrate how to provide checkpoints in the code to insure that it performs as specified.

To illustrate how these views help explain the quality of a program, consider the activity of testing a program to detect a possible bug. The quality of the solution is directly related to the ease with which the bug can be isolated and corrected. Ease in identifying and correcting the problem depends on the readability of the code (software as an English document), localization of the affected code (software as structured on data abstractions), and tractability of the code to modification (software as a temporary solution).

Before closing, we again mention that you should not feel uncomfortable if these ideas seem a bit vague and ill-defined. We will spend most of the rest of the book developing methodologies intended to satisfy the goals and principles outlined here.

z.6 Summary

In this chapter, we enumerated some of the important attributes we want our systems to have. Of particular importance are the attributes of correctness – the system conforms to its specifications – and maintainability – the system can be modified with minimum difficulty to meet changing requirements. We noted that it was essential for our systems to be modular constructions, and that the composing modules be

- highly cohesive: that is, completely encapsulating a single notion; and
- weakly coupled, interacting with other modules only through a well-defined minimal interface.

We briefly touched on the importance of constructing reusable – that is, tailorable – modules.

We concluded with three design principles:

- *information hiding*: a module has no access to anther module except as explicitly required by the interface;
- *continuity*: a small change in specifications necessitates only a small change in the system; and
- *open/closed*: a module that is available for use by other modules can still be modified;

and three views of software:

- software is a temporary solution to a changing problem;
- software is data design followed by algorithm design;
- software is a technical English document and a formal mathematical document.

Exercises

z.1 The discussion of correctness in Section y.1 states that "we can test the system, but we cannot conceivably test all situations." Suggest some general guidelines for "adequately" testing a system. Can you think of any other ways of insuring a system's correctness besides testing?

z.2 Give examples to illustrate the three type of maintenance modifications in each of the following systems:

a. a university registration system;

b. the maze game;

c. a company payroll system;

d. a university course scheduling system;

e. a hospital patient management system.

z.3 If system maintenance is as high as 70% of the budget dedicated to system development, why aren't inadequate systems simply abandoned and new systems constructed?

z.4 Each of the statements below refers to a component or functionality of a university registration system. Indicate what property was violated in its development.

a. The user interface component handles input/output operations to get and display data, and keeps track of the number of times the system is used.

b. The properties used to model a student include name, social security number, address, courses the student is enrolled in, and the instructors of these courses.

c. Students who take only occasional courses and are not pursuing a degree need to be accommodated in the registration system. In evaluating the system for this extension, it is determined that about 80% of the system's code needs to be modified.

d. The component that computes a student's g.p.a. uses the date the g.p.a. was last computed, and whether the student has completed any new courses since then.

z.5 Discuss how various system properties affect the ability of a software system to evolve.

z.6 *Correctness* is a software property that overrides any other. Suppose one system is very efficient in producing results, but sometimes the results are not within the required degree of accuracy. Another system is not as fast, but all its results are correct and within the required accuracy. Which system would you prefer and why? How would you respond to someone who argued "I just need ball park figures from the system, but I need them quickly"? How would you respond to someone who said "I'll take a system that's fast, even if I have to put up with it crashing once in a while"?

z.7 Assume a set of requirements is implemented via two systems, where one system's source code is twice as large as the other's. What can you infer about the systems with respect to:

a. time taken to develop the system;

b. correctness of the system;

c. maintainability of the system;

d. efficiency of the system.

z.8 When we implement a method, what do we do that reflects the fact that we are preparing a "mathematical document"? What do we do that reflects the fact that we are preparing an "english document"?

GLOSSARY

cohesion: a measure of the degree to which a module completely encapsulates a single notion.

compatibility: a property exhibited by software systems that can be used in conjunction with each other.

continuity principle: a system should be designed so that a small change in requirements can be accomplished with a correspondingly small change in the system.

correctness: a property exhibited by a system that performs according to its specifications.

coupling: a measure of the degree to which a module interacts with and depends on other modules.

efficiency: a property exhibited by a system that requires a minimal amount of time to accomplish a task.

extendibility: a property exhibited by a system that can be easily modified to meet additional requirements.

functional strength: a property exhibited by a module that performs only one function, and performs that function completely.

information hiding principle: a module should be designed so that only a well-defined, essential subset of the module's features are available to client modules.

maintainability: a property exhibited by a system that can be easily modified to adapt to changing requirements, and in which run-time errors can be easily isolated and corrected.

modularity: a property exhibited by a system that is composed of coherent, self-contained, syntactic components.

open-closed principle: modules should be available for use by other modules (closed) and still available for extension (open).

portability: a property exhibited by a system that can be easily transferred to different platforms.

real-time system: a system whose correctness depends on responding to a stimulus within a well-defined elapsed time.

robustness: a property exhibited by a system that behaves reasonably when encountering unexpected circumstances or incorrect input.

weak coupling: a property exhibited by a module that has minimal interaction with other modules.

12 Addendum zSoftware quality