Software Engineering: Planning for Change

David Alex Lamb
Department of Computing and Information Science
Queen's University
Kingston, Ontario, Canada

Prentice-Hall International, Inc.

Exercises

6–1 Arrange the following eleven tasks into a schedule.

Task	Length	Depends on	Task	Length	Depends on
A	2	K	B	3	H
C	4		D	2	F, G
E	1	D, I, J	F	3	B
G	3	C, K	H	3	
I	2	A, F	J	1	F
K	2	H			

- Identify the critical path.
- What is the minimum time to complete this schedule?
- What tasks are allowed to slip, and by how much?
- How many activities may proceed in parallel?

6–2 This question assumes you have been keeping a personal log like the one described in Section 15.4, and that you have been taking part in a group project. Summarize from your log how much time you spent on the group project, with subtotals for each phase of the project. Combine this with the information from your colleagues who worked on the same project. Count the number of lines of code in all modules of the program you produced, using a tool if one is available. Do not include lines of code from test drivers and stubs.

a. Determine your group's productivity in lines of code per hour. Extrapolate to lines of code per day and per year, assuming four productive hours per day and 1,000 productive hours per year.

b. Determine what percentage of your time you spent on each of the project's phases.

Chapter 17

Configuration Management

To make progress, developers need a careful mixture of stability and change. You must eventually fix bugs and introduce new features, but you must ensure you don't interfere with your co-workers. *Configuration management* is a discipline for controlling changes to a system to avoid confusion, misinterpretation, and interference. A *software configuration* is a collection of related documents, source files, and tools used in designing and implementing a software system.

Configuration management consists of three activities. *Configuration identification* decides what pieces of the system you need to keep track of. *Configuration control* ensures that changes to a configuration happen smoothly. *Configuration status and accounting* keeps track of what changed and why it changed.

17.1 The Need for Configuration Management

A large software project needs configuration management simply to keep track of the many objects that make up the system, and how to put them together. A project of any size, large or small, must also deal with multiple *versions* of the same software. A new version of an object is a *revision* if you intend it to completely replace the object from which it came; a module revision might fix a bug, or introduce new functionality for all possible versions of a system. A new version of an object is a *variant* or *variation* if you intend it to co-exist with the object from which it came; the two variants represent different members of the same program family (see Section 5.2.1). Formally, you can define a history relation, *is version of*,

between objects, and split it into two subrelations, *is revision of* and *is variant of*. In general, these relations form a directed acyclic graph, since you might create a version that merges several others.

Variants cause at least two kinds of problems. First, suppose you have several variants of the same module, and find a bug in one of them. If the bug is in code common to the variants, you need to fix it in them all, and thus must be able to find them all. Second, suppose two people each create variants of a base module to add new functionality; often, one eventually wants to merge both sets of changes back into the main line of revisions. If neither person knows about the other, each can try to update the main revision stream simultaneously, and one set of changes will be lost.

Even in a system where all versions are revisions can have problems. If a fellow programmer is debugging module M1, your changing M2 can change the symptoms of the bug. The hardest thing about some problems is to reproduce them consistently; changes in apparently unrelated code can make the bug behave differently, because the code is a little larger or smaller, or because the pattern of data in memory is different.

With modern programming languages and tools, the relationships between configuration elements become more complex. Certain kinds of changes to a module may force you to recompile some clients of the module. A source file may be input to some tool, whose output becomes input to a second tool, and so on through several steps before you get object code that is part of the running system. When a tool changes, you may need to regenerate its output files. You need to identify these relationships, and carefully manage rebuilding of the system when some parts change. To make progress, developers need a careful mixture of stability and change. You must eventually fix bugs and introduce new features, but you must ensure you don't interfere with your under co-workers.

17.2 Configuration Identification

Configuration identification is the process of deciding what things need to be placed under configuration control, and what the relationships between them will be.

17.2.1 Objects

Any object is in one of three categories. *Controlled* objects are under configuration control; there are formal procedures you must follow to change them. *Precontrolled* objects are not yet under configuration control, but will be eventually. *Uncontrolled* objects are not and will not be subject to configuration control. *Controllable* objects include both *controlled* and *precontrolled*.

Typical controllable objects include

- Any of the design documents from Figure 2–1 on page 8. The configuration manager might decide to split some of these objects into individual chapters, controlled individually.

- Tools used to build the product, such as compilers, linkers, program component generators (such as the lexer and parser generators used in most compiler-building projects).
- The source code for each module.
- Input to tools other than source files, such as command line arguments, command files, and system libraries.
- Test cases. Changes to tests may be less likely, but during the coding and testing phase an ill-considered change might impede progress. For example, one developer might be using a particular test case to track down some complex problem.
- Problem reports (see Section 10.3.1). These are a primary source of requests for changes.

The Configuration Management Plan, written during the Project Design stage, lists what objects to control. The managers who develop the plan must strike a balance between controlling too much, and thus impeding development, and controlling too little, thus leading to confusion when something changes.

Some typical documents that might not be subject to controls include personal logs and minutes of meetings. The first belongs to the individual keeping the log; likely no one else would read it. The second would never change, and so needs no change controls. Minutes have an intermediate place between uncontrolled objects such as logs and controllable ones such as source code; someone must keep track of them, so that developers can refer to what happened.

17.2.2 Relationships

The conventional English meaning of "configuration" includes not only a collection of components, but also relationships among those components. In software, "configuration management" has come to mean primarily managing multiple versions of components. The more recent term *system modeling* covers the other half of the English meaning.

When you build a piece of software, you start with objects you build by hand, pass those objects through tools to derive new objects, and combine those derived objects into one (or at most, a few) objects you deliver to customers. For example, you write two Pascal modules, compile each of them into relocatable files, and link the relocatables into an executable program. The original objects are *source* objects, and the output from the tools are *derived* objects. The objects you deliver to customers are *exported* objects; normally they are derived objects such as executable programs, but some source objects might also be exports. The system model describes the process by which you generate the exported objects from the source objects.

Formally, a system model is a directed acyclic graph with two types of nodes: objects and processing steps. Objects and processing steps alternate in the graph; each processing step takes one or more objects as inputs and produces one or more objects as outputs. Processing steps are simply placeholders in the graph. Objects

include familiar things such as the source and derived files mentioned previously, but also include tools (such as compilers and linkers), command line arguments for invoking them, "hidden" files read automatically by the tools (such as standard module libraries), and even such data as the value of the time-of-day clock, if any outputs depend on them. With this view, derived objects are simply the outputs of processing steps, and source objects are inputs to some steps that are not outputs of any other steps.

The idea of a system model is that if you begin with the same source objects, and run through the same processing steps, you would get the same output. This is important if you ever need to regenerate an old system to track down a customer's problem. The idea of including tools, command line arguments, and hidden files is important. Experienced developers allude to the idea of "software rot": you try to recreate a program after some months or years, only to discover that it no longer works. What has usually happened is that a system library has changed, or someone has installed a new release of the compiler; what you thought of as unchanged source really has changed, because you did not preserve all inputs to the system building process.

To be serious about system modeling and reproducible system building, you must treat source objects as immutable. Instead of changing an object, you create a new object (a version of an old object). Instead of changing a file and rebuilding the system, you substitute new versions of some source objects, and re-derive all the appropriate derived objects. As of the mid-1980s, most file systems and database systems did not provide enough support for system modeling. Some tools, such as SCCS and RCS under UNIX, could keep track of versions of source files. Experimental systems with much better support for the full process showed promise of coming into production use within a few years.

17.3 Configuration Control

Configuration control is the process of managing changes; it is the piece of configuration management that most directly affects day-to-day operations of developers.

Each controllable object starts off as precontrolled; the people assigned to work on it may make whatever modifications they choose. When the object is reasonably complete, the developers (or, more likely, their team leader or manager) places it under configuration control, and it becomes a controlled object.

Details of when to make something controlled vary between projects; the Configuration Management Plan should define criteria for making such a decision. When someone other than the creator needs to refer to it is a typical time. For example, a design document might go under configuration control when its authors circulate a first draft to reviewers. Source code might go under configuration control just before testers need it during integration.

17.3.1 Baselines and Updates

The whole reason for configuration control is that people need a stable environment to make progress. If you are trying to integrate module A with modules B and C, you cannot make progress if the developer of module C keeps changing it out from under you; this is especially frustrating if a change to C forces you to recompile A. Thus before anyone who depends on a particular object need to use it, the configuration manager *freezes* it; no one may change the frozen object. This establishes a *baseline* for others to use and depend on. The term comes from surveying; surveyors carefully measure a baseline, then make all other measurements relative to that line.

Freezing a configuration may involve archiving everything needed to rebuild it. Archiving means copying to a safe place such as magnetic tape. An archival tool examines a system model, writes a representation of the model to tape, then copies all source objects to tape. Some of these "source objects" may actually be programs used to generate derived files. A partial archive saves the executable programs; a full archive saves enough to rebuild them. This process recurs with programs used to build the programs, and so forth; it stops by saving the executable form of programs considered immutable (such as a compiler from a particular old release tape from a manufacturer).

It usually does become necessary to change a baseline at some point. For example, those who review a document need a baseline so that they know they are all referring to the same document. However, their review usually leads to changes in the document. What happens here is that developers copy the baseline document, then change the copy. When a reviewer says "change page 4, paragraph 7" a developer may need to translate this to page 6, paragraph 2 of the new document, but both people share a standard reference point. This process of changing a copy is the basis of immutability.

At any given time a programmer may pay attention to three versions of a configuration:

1. the current baseline configuration
2. an updated configuration
3. the programmer's private version

A large enough project may have several distinct updated configurations, one per subproject plus a master updated configuration. There may be one private configuration per programmer; a developer may make whatever changes she chooses to her own private configuration. Moving objects from a private version to the updated configuration requires approval from a manager responsible for change control. Objects never move into a baseline configuration; instead, every so often the configuration manager freezes the updated configuration and declares it to be the new baseline. A developer who needs stability uses the current baseline configuration.

One who needs the most recent version uses the updated configuration, but must live with the consequences of frequent changes.

17.3.2 Change Control

Once an object goes under configuration control, any changes require management approval. Approval usually certifies several things about the change:

1. The change is well-motivated.
2. The developer has considered and documented the effects of the change.
3. The change interacts well with changes made by other developers.
4. Appropriate people have validated the change (for example, someone has tested a code change, or has verified that a requirements change is consistent with other requirements).

The amount of work involved here depends on the size of the change and the importance of the changed object. For source code, you might be able to describe both motivation and effects by a short statement like "Fixes problem report 17, callers need to be recompiled." A manager might disallow a change made from motivations such as "changed spelling of all original author's identifiers," or with effects such as "requires extensive editing of all modules containing calls on procedure X." If a manager approves a change with such large effects, the configuration control procedures should notify all the appropriate people of the need to make such a change. A configuration manager needs to be extremely hard-nosed; it is hard to convince someone who wants a bug fixed today to wait a week, accepting a short-term loss of personal productivity in return for a gain in group productivity.

17.3.3 System Rebuilding

When you update a source object in a configuration, you must regenerate the derived objects that depend on it. With older systems this meant compiling the source file and placing the resulting object file in a library. Modern systems are complex enough to require tools to manage the system rebuilding process.

A system rebuilding tool takes as input a system model and a list of files to substitute for some of the objects (normally, source files) of the model. It reruns those processing steps that depend on the changed objects, then reruns those steps that depend on changed derived objects, and continues until it has regenerated all derived objects that depend directly or indirectly on anything that has changed. The most widely known system rebuilding tool is the UNIX *make* utility. Each object is a file; it deduces that an object has changed by the "date of last change" maintained by the file system, and rebuilds a derived object when its date is less than that of anything on which it depends. More recent experimental tools compare the new object to the old. For example, adding a comment to a source file might not require any rebuilding. For another example, if a tool produces several outputs, a change to its source file may affect only one output. Eliminating unnecessary regeneration steps can greatly reduce the time to rebuild the system, especially if the comparisons are fast or the output of a tool becomes input to many other tools.

17.4 Configuration Status and Accounting

Status and accounting ensures that developers, managers, and users know the history and current state of controllable objects. A particular developer or customer may need to know when a particular problem is fixed; managers may need to know how often changes are happening. The two major types of object for which people need to know status are individual modules and *problem reports* (see Section 10.3.1).

Several distinct types of information contribute to the status of a module. The configuration identification activity may separate these into distinct objects, but for status purposes it is more useful to group them together. A common practice is to create a *module development folder* for each module. In its simplest form this is a physical file folder. The folder contains

- the requirements the module implements (if it is a behavior-hiding module)
- the current specification, design, and code listing
- a log identifying all changes to the specification, design, or code, including date of change and who made the change
- the unit test plan for the module
- records of test case results
- copies of problem reports that affect the module
- notes about the module by the specifier, designer, and coder of the module
- a cover page with summary information, such as planned and real dates for completion of the specification, design, and code, with a place for signatures for the person responsible for verifying completion.

Status reporting involves analyzing and summarizing information about changes. The simplest report is the status of some collection of problem reports: who is responsible for handling them, what their current disposition is, what progress is being made on changes needed to fix the problems, and so on.

Managers also need summary information. For example, frequent change requests may suggest adding more resources to handle them, or may suggest improving quality control procedures. If you classify many problem reports as misunderstandings, you may need to improve user documentation. Finally, analyzing what changes people request may lead to better planning for change in future systems.

17.5 Configuration Hierarchy

So far this chapter has presented a system as one large configuration, with a single manager responsible for configuration control. This may be true of a medium-sized project, but is probably not true of a large one.

A large project may consist of several small projects, each with its own development schedule and configuration control. For example, a project may have several subprojects for developing individual pieces of the system, plus subprojects for developing or maintaining in-house tools such as compilers, simulators, or test-case generators. The manager responsible for each small project may have configuration control responsibility for his own project.

Typically, changes to widely visible portions of a system may require approval from a *configuration control board*. Such a board would consist of representatives of both developers and customers. For example, changes to requirements require negotiation and joint approval because they involve changes to a contract. In a more complex contractual arrangement, there may be operating and sponsoring agencies distinct from customers; they should have representation too. Changes to a major in-house tool such as a compiler might require approval from the compiler developers as well as the principal users of the compiler within the development organization. All representatives should be fully capable of committing their organizations to accepting whatever decisions the board makes.

Further Reading

Babich (1986) gives a highly readable explanation of configuration management based on modern industrial experience. Glass' books include many tales of computing disasters, some of them attributable to poor configuration management [Glass 1977, Glass 1978, Glass 1979, Glass 1981].

Chapter 18

Quality Assurance

Quality assurance is the process of raising a developer's (and possibly a customer's) confidence that a system is of high quality. It is a planned, systematic activity. The two major thrusts of quality assurance are building in quality and measuring quality. Most of Part II discusses building in quality; this chapter concentrates on measurement and analysis. A related topic, *verification*, demonstrates that specifications are consistent and complete, and that implementations meet their specifications. Part III, on specification techniques, discusses verification issues. Chapter 8 discusses testing.

18.1 Measures of Quality

Software quality has several aspects. Some of these we can measure readily; for others, we can measure something that seems related to the quality we're looking for; for others, we currently can judge only subjectively. The following sections discuss several aspects of software quality. During requirements analysis, you should consider this list of qualities, define requirements for the measurable ones, and define goals for trading off the immeasurable ones.

There are two criteria with which to judge how well you can guarantee that a system is good according to a particular measure of quality. The first is the degree to which you can measure the quality; the second is the degree to which you can build your systems to embody the quality.