CHAPTER 1

What Is Software Configuration Management?

The title of this chapter asks such a simple question, the answer to which, one would think, ought to be known by anyone with any kind of record in software development. In reality, it seems that few are able to actually articulate what is meant by that term *software configuration management*.

To be fair, those who have a record in software development recognize that there is a need to control what is happening in the development process, and, once controlled, there is a sense that the process can be measured and directed. From that recognized need, then, comes a good working definition of software configuration management:

> Software Configuration Management is how you control the evolution of a software project.

Slightly more formally, *software configuration management* (SCM) is a software-engineering discipline comprising the tools and techniques (processes or methodology) that a company uses to manage change to its software assets. The introduction to the IEEE "Standard for Software Configuration Management Plans" [IEEE 828-1998] says this about SCM:

> SCM constitutes good engineering practice for all software projects, whether phased development, rapid prototyping, or ongoing maintenance. It enhances the reliability and quality of software by:

- Providing structure for identifying and controlling documentation, code, interfaces, and databases to support all life-cycle phases
- Supporting a chosen development/maintenance methodology that fits the requirements, standards, policies, organization, and management philosophy
- Producing management and product information concerning the status of baselines, change control, tests, releases, audits, etc.

Clearly, software is easy to change—too easy. And not only is it easy to change, but it is unconstrained by the physical laws that serve as the guardrails of what is possible with hardware systems. Software is bounded only by the limits of the human imagination. Uncontrolled and undirected, imagination can quickly give rise to nightmare.

Today most software project teams understand the need for SCM to manage change to their software systems. However, even with the best of intentions, software projects continue to fail because of problems that could have been avoided through the use of an SCM tool and appropriate processes. These failures are reflected in poor quality, late delivery, cost overruns, and the incapability to meet customer demands.

To understand software configuration management, you might find it easier to look first at configuration management in a hardware-development environment. Hardware systems have physical characteristics that make the problems caused by the lack of sound configuration management easier to see.

For example, consider a personal computer. A computer has a processor, a mainboard, some memory, a hard drive, a monitor, and a keyboard. Each of these hardware items has an interface that connects it to other hardware items. Your mouse has a plug, and your computer has a port into which you plug your mouse, and, voilá, everything works.

If the plug on the mouse was not compatible with the port on the computer, there would be no way to connect the two pieces of hardware into a working system. Throughout the computer, there are many other similar interfaces. The processor and memory plug into the mainboard, the hard drive plugs into the computer, and the printer, monitor, and keyboard all have interfaces.

When the hardware is manufactured, the interfaces that are essential for the operation of the final system are easily seen. Therefore, they are well known and are carefully examined whenever changes are made to the hardware design.

For a hardware system, configuration management has the following aspects. Each system is numbered or identified and also has a version number. Each version number identifies different designs of the same part. For example, the model year for a car is a version number of that car. It is a 2003 Honda CRV, a 2004 Honda CRV, and so on. When the design of a hardware system is changed, the system gets a new version number.

A hardware system can be made up of hundreds, thousands, or tens of thousands of parts. The next thing that must be recorded is which versions of these parts go together. In manufacturing terms, this is often called a bill of materials. The bill of materials lists all the parts and specifies which version of each part should be used to build the system.

Parts are assembled into bigger parts, which simplifies the manufacturing process for large systems. In the personal computer example, you can say what version of the mouse, hard drive, processor, and monitor go together to make a complete system. Each of these parts, such as a hard drive, is made of many, many subparts, all of which must go together to have a working unit.

Software configuration management deals with all of the same problems as hardware configuration management (and more because of the lack of the guardrails that the laws of physics provide). Each software part has an interface, and software "parts" are plugged together to form a software system. These software "parts" are referred to by different names, such as subsystems,

modules, or components. They must be identified and must have a version number. They also must have compatible interfaces, and different versions of parts can have different interfaces. Ultimately, you need a bill of materials to see which versions of which components make up the entire software system.

However, software configuration management is much harder to get right because software is much easier to change than hardware. A few keystrokes and a click of the Save button, and you've created a new version of the software. Unlike hardware, software manufacturing is very fast and can be performed hundreds of times a day by individuals on a software team. This is usually referred to as "performing a software build" or "building the software."

In this dynamic, changing environment, the discipline of SCM is brought to bear to ensure that when a final version of the entire software system is produced, all of the system's component parts can be brought together at the same time, in the same place, and can then be plugged together to work as required. Although most software project teams understand that they need SCM, many fail to get it right—not only because SCM is complex, but also because there isn't a clear understanding of specifically what a good SCM system should do. To begin to create that understanding for you, the rest of this chapter discusses key best practices of SCM in detail and introduces the concepts of the SCM tools and processes that are used to implement those best practices.

1.1 SCM Best Practices

When implementing SCM tools and processes, you must define what practices and policies to employ to avoid common configuration problems and maximize team productivity. Many years of practical experience have shown that the following best practices are essential to successful software development:

- Identify and store artifacts in a secure repository.
- Control and audit changes to artifacts.
- Organize versioned artifacts into versioned components.
- Organize versioned components and subsystems into versioned subsystems.
- Create baselines at project milestones.
- Record and track requests for change.
- Organize and integrate consistent sets of versions using activities.
- Maintain stable and consistent workspaces.
- Support concurrent changes to artifacts and components.
- Integrate early and often.
- Ensure reproducibility of software builds.

The rest of this section explains each of these best practices.

1.1.1 Identify and Store Artifacts in a Secure Repository

To do configuration management, you must identify which artifacts should be placed under version control. These artifacts should include both those used to manage and design a system (such as project plans and design models) and those that instantiate the system design itself (such as source files, libraries, and executables and the mechanisms used to build them). IEEE calls this *configuration identification:* "an element of configuration management, consisting of selecting the configuration items for a system and recording their functional and physical characteristics in technical documentation" [IEEE Glossary, 1990].

In terms of an SCM tool, identification means being able to find and identify any project or system artifact quickly and easily. Anyone who has managed a development project with no SCM or poor SCM can attest to the difficulty of finding the "right" version of the "right" file when copies are floating around all over the place. Ultimately, losing or misidentifying artifact versions can lead to the failure of a project, either by hindering delivery of the system because of missing parts or by lowering the quality of the system because of incorrect parts.

Organizing artifacts and being able to locate them are not enough. You also need fault-tolerant, scalable, distributable, and replicable repositories for these critical assets. The repository is a potential central point of failure for all your assets; therefore, it must be fault-tolerant and reliable. As your organization grows, you will add data and repositories, so scalability and distributability are required to maintain high system performance.

Another means of growth in today's software market is through acquisition, which affects many companies by resulting in the geographical distribution of development groups. The SCM tool, therefore, must be capable of supporting teams that collaborate across these geographically distributed sites.

Finally, the repositories should be backed up with appropriate backup and disaster-recovery procedures. Sadly, many companies overlook this last step, which can lead to severe problems.

1.1.2 Control and Audit Changes to Artifacts

After the artifacts have been identified and stored in a repository, you must be able to control who is allowed to modify them, as well as keep a record of what the modifications were, who made them, when they were made, and why they were made. We refer to this as the audit information. This best practice is related to the IEEE configuration management topics *configuration control* and *configuration status accounting,* defined, respectively, as "the evaluation, coordination, approval or disapproval, and implementation of changes to configuration items" and "the recording and reporting of information needed to manage a configuration effectively" [IEEE Glossary, 1990].

Using both control and audit best practices, an organization can determine how strictly to enforce change-control policies. Without control, anyone can change the system. Without audit, you never really know what went into the system. With audit information, even if you don't restrict changes, you can see at any time what was changed, by whom, and why. The audit information also enables you to more easily make corrections if errors are introduced. Using control and audit

in balance enables you to tune your change control approach to best fit your organization. Ideally, you want to optimize for development productivity while eliminating known security risks.

1.1.3 Organize Versioned Artifacts into Versioned Components

When there are more than a few hundred files and directories in a system, it becomes necessary to group these files and directories into objects representing a larger granularity, to ease management and organization problems. These single objects, made up of sets of files and directories, have a number of different names in the software industry, including packages, modules, and development components. For the purposes of this book, an *SCM component* is a set of related files and directories that are versioned, shared, built, and baselined as a single unit.

To implement a component-based approach to SCM, you organize the files and directories into a single SCM component that physically implements a logical system design component. The Rational Unified Process (RUP) refers to the SCM component as a *component subsystem* [RUP 5.5, 1999] (RUP is a software-engineering process developed and marketed by Rational Software).

A component-based approach to SCM offers many benefits, as follows:

- *Components reduce complexity.*

  Use of a higher level of abstraction reduces complexity and makes any problem more manageable. Using components, you can discuss the 6 that make up a system instead of the 5,000 files subsumed under them. When producing a system, you need to select only 6 baselines, one from each component, instead of having to select the right 5,000 versions of 5,000 files. It is easier to assemble consistent systems from consistent component baselines than individual file versions. Inconsistencies result in unnecessary rebuilding and errors discovered late in the development cycle.

- *It is easier to identify the quality level of a particular component baseline than that of numerous individual file*s.

  A component baseline identifies only one version of each file and directory that makes up that component. Because a component baseline contains a consistent set of versions, these can be integration-tested as a unit. It is then possible to mark the level of testing that has been performed on each component baseline.

  This method improves communication and reduces errors when two or more project teams share components. For example, a project team produces a database component, and another team uses it as part of an end-user application. If the application project team can easily determine the newest database component baseline that has passed integration testing, it will be less likely to use a defective set of files.

- *Instantiating a physical component object in a tool helps institutionalize component sharing and reuse.*

  After component baselines have been created and the quality level has been identified, project teams can look at the various component baselines and choose one that can be

referenced or reused from one project to the next. Component sharing and reuse is practically impossible if you cannot determine which versions of which files make up a component. Component sharing between projects is not practical if you cannot determine the level of quality and stability for any given component baseline.

- *Mapping logical design components to physical SCM components helps preserve the integrity of software architectures.*

  In an iterative development process, pieces of the software architecture are built and tested early in the software life cycle to drive out risk. By mapping the logical architectural components to the physical SCM components, you gain the ability to build and test individual pieces of the architecture. This mapping between architecture and the implementation of the architecture leads to higher- quality code and cleaner interfaces between the components by preserving the integrity of the original architecture in the SCM tool.

1.1.4 Organize Versioned Components and Subsystems into New Versioned Subsystems

To allow management of highly complex software systems, you must be able to step beyond the management of individual components and group those components into subsystems. Beyond that, you must also be able to include other subsystems in the definition of a subsystem. This recursive definition of a subsystem as a collection of compatible components and subsystems allows incredibly complex systems to be hierarchically defined, controlled, and managed. Going back to the personal computer example, recall that the PC is composed of a processor, a mainboard, some memory, a hard drive, a monitor, and a keyboard. Each of these entities can be described as a subsystem—that is, a collection of components and other subsystems. Using this recursive method of describing the system, the structure of the entire PC can be specified and managed hierarchically, all the way down to the individual parts. Note that if you wanted, you could use this recursive method to specify the PC all the way down to its atoms. Furthermore, the subsystems defined in that hierarchy, such as the hard drive, can be reused in other PC designs. Figure 1-1 illustrates the hierarchical nature of subsystems.

The ability to use an SCM system to recursively define and manage the subsystems that make up a software system enables you to define and control very complex development efforts and to designate subsystems as projects that are independently controlled, managed, and released and that can become candidates for reuse in other projects.

1.1.5 Create Baselines at Project Milestones

At key milestones in a project, all the artifacts should be baselined together. In other words, you should record the versions of all the artifacts and components that make up a system or subsystem at specific times in the project. At a minimum, artifacts should be baselined at each major project milestone. In an iterative development process as prescribed by the Rational Unified Process [RUP 5.5], at a minimum, baselines should be created at the end of each project iteration.

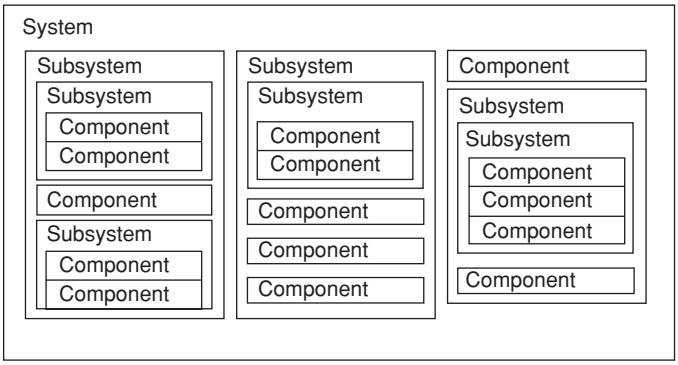

Figure 1-1 Decomposing larger systems into hierarchical subsystems makes it easier for complex entities such as this one to be defined and managed.

Typically, new baselines are created more frequently (sometimes daily) near the end of an iteration or release cycle. It can be useful to create baselines before each nightly build. This enables you to reproduce any project build, query what has changed between builds, and indicate the stability of a build using baseline quality attributes.

There are three main reasons to baseline: reproducibility, traceability, and reporting. Reproducibility is the ability to go back in time and reproduce a given release of a software system or a development environment that existed earlier. Traceability ties together the requirements, project plans, test cases, and software artifacts. To implement it, you should baseline not only the system artifacts, but also the project-management artifacts. Reporting enables you to query the content of any baseline and to compare baselines. Baseline comparison can assist in debugging errors and generating release notes.

Good traceability, reproducibility, and reporting are necessary for solving process problems. They enable you to fix defects in released products, facilitate ISO-9000 and SEI audits, and ultimately ensure that the design fulfills the requirements, the code implements the design, and the correct code is used to build the executables.

1.1.6 Record and Track Requests for Change

Change request management involves tracking requests for changes to a software system. These requests can result from defects found by a testing organization, defects reported by customers, enhancement requests from the field or customers, or new ideas produced internally.

Recording and tracking change requests supports configuration and change control as defined by IEEE (see the "Control and Audit Changes to Artifacts" section, earlier in this chapter). The critical points are that change requests are recorded and that the progress, whether through implementation or a decision never to implement, is tracked. Beyond simply tracking change requests, a good change-management process enables project management to prioritize and establish target dates for the inclusion of change requests in product releases. Chapter 12, "Change Request Management and ClearQuest," covers this best practice in more detail.

1.1.7 Organize and Integrate Consistent Sets of Versions Using Activities

Although all SCM systems provide version control at the file level, it is left to the software developer to keep track of which versions of which files go together to implement a logical, consistent change and to ensure that this change is integrated as a unit. This is a tedious, manual, and error-prone process. Mistakes are easy to make, especially if a developer is working on more than one change at a time. This can lead to build errors and lost development time. Mistakes also show up as runtime defects that can't be reproduced by the developer in his or her working environment.

Some SCM tools provide a way for developers to record which change request or defect they are working on. This information is used to track which file and directory changes make up a single logical change. Often this information is not used by the SCM tool, but is instead maintained only for reporting and auditing purposes. The key advantage of collecting this change information is streamlining the integration process and ensuring consistency of the configuration in any given working or build environment.

The grouping of file and directory versions is called a *change set*, or change package. (Some in the SCM industry distinguish between these two terms. The difference is subtle and has to do with the implementation. A change set is defined as the actual delta that comprises the change, even if it spans files. A change package is a grouping together of a set of file versions. In this book, I use the ClearCase term *change set* to refer to the grouping and manipulation of a change.) This grouping is mostly useful when the *change set* contains a single logical change. The change set approach has been around for a long time. In 1991, Peter Feiler wrote an excellent paper, "Configuration Management Models in Commercial Environments" [Feiler 1991], that describes the change set model.

Change sets are only the glue. There must be a link between the versions that are changed and the activity that is the "why" for the change. An *activity* represents a unit of work being performed. Activities can be of different types. For example, a defect, enhancement request, and issue are all activities. This unit of work ties directly into the change request management system and processes. An activity might also be a child of another activity that appears in the project-management system. The capability of the change set to tie together the disciplines of configuration management, change request management, and project management is where the power of the activity-based approach really becomes visible.

The key idea of activity-based SCM is to increase the level of abstraction from dealing with files and versions to dealing with activities. This is done first by collecting versions created during development into a change set and associating that change set with an activity. Activities are then presented throughout the user interface and used by SCM operations as the way to operate on a consistent set of versions.

The benefits of activity-based SCM are as follows:

- *Consistent changes cause fewer build and integration problems.*

  Integrating a consistent change set (single logical change) reduces build and integration errors caused by developers forgetting about files when delivering their changes. It also

ensures that the testing was performed against the same versions being integrated, making integration errors less frequent.

- *Activities are logically how people group what they do.*

  Generally, developers think about what feature, request for enhancement, or defect they are working on, which are all types of activities. By conforming to this activity-centric approach in the SCM tool and by using automation, developers are not required to know many details of the SCM implementation.

  Activities are a level of abstraction that all project members can use in common, enabling project leaders, testers, developers, and customer support personnel to communicate more effectively.

- *Activities provide a natural link to change request management.*

  Change request management (a subset of which is defect tracking) is an essential part of most software-development organizations, and tracking change requests is one of the key best practices of SCM. Instead of being a collection of versions in meaningless bundles, the change set should be tied to the change request stored in the change request management system. This combining of the change request and the change set allows accurate reporting of what defects were fixed and what files were changed between project baselines.

- *Activities provide a natural link to project management.*

  Project managers are interested not only in what is being changed but also in the status of the change, who is assigned to it, and how much effort is estimated to be needed to implement the change. The change set links the project-management data for an activity to the files and versions that are changed. This link supports better automation, bringing advantages to the project leader without requiring extra effort on the part of the developer. For example, when a developer completes a change in the SCM tool, a change of status in the activity could be made automatically and would show up on a project report.

- *Activities facilitate reporting.*

  Activity-based SCM allows all reports and tools to display information in terms of the changes made rather than the files and versions that went into making the change. This is more natural for everyone involved with a project.

- *Activities streamline code reviews.*

  Traditionally, when performing code reviews, a reviewer receives a list of files and which versions of these files to review. The trick is determining which version to compare against when doing the review. Should you compare the latest version against the immediate predecessor, the last baseline, or some other version? It is unclear.

With the change set information, it is possible for the SCM tool to provide the developer with the predecessor version of the change set automatically. This makes performing code reviews easier and less error-prone.

- *Activities streamline testing efforts.*

  Testing organizations often work with software builds "thrown over the wall" by development organizations. They must determine what went into the build or what was different from the previous build to decide what needs to be tested and what level of testing is required.

  Most testing organizations do not have the resources to run the full test suite on every software build that development delivers. So, automatic reporting between two baselines that provides a list of the activities included in the new baseline is far easier to work with than a list of the hundreds of file versions that were modified from one baseline to the next.

1.1.8 Maintain Stable and Consistent Workspaces

The developer requires tools and automation to create and maintain a stable working environment. This maintenance involves periodically synchronizing changes with other team members in a way that results in a consistent set of shared changes that are of a known level of stability.

Consistency and stability in the developer's working environment maximize the developer's productivity. Without stable and consistent workspaces—private file areas where developers can implement and test code in relative isolation—developers spend significant time investigating erroneous build problems and are sometimes unable to build the software in their own workspaces. These problems can quietly sap time and available effort from any project.

A stable model allows developers to isolate themselves easily from disruptive changes going on in other parts of the project and for the developer to decide when it is appropriate to introduce change into the workspace. A stable model also allows a project to be isolated from disruptive changes occurring in a developer's workspace.

A consistent model means that when developers update their workspace, these updates will consist of a known, buildable, and tested set of versions.

1.1.9 Support Concurrent Changes to Artifacts and Components

Ideally, only one person would be making changes to any single file at a time, or only one team would be working on any single component at a time. Unfortunately, this is not always efficient or practical. The most obvious case is when maintaining a release in the field while developing the next release.

Early SCM tools forced users to serialize changes to files. This was inefficient, in that some developers were blocked waiting for other developers to complete their changes. It was also a problem from a quality standpoint. Blocked developers often worked around the system by getting a copy of the file without checking it out and modifying it outside SCM control. After the original developer checked in his or her changes, other developers would check out the file, copy their changes in place, and check the file back in, unknowingly removing the previous developer's

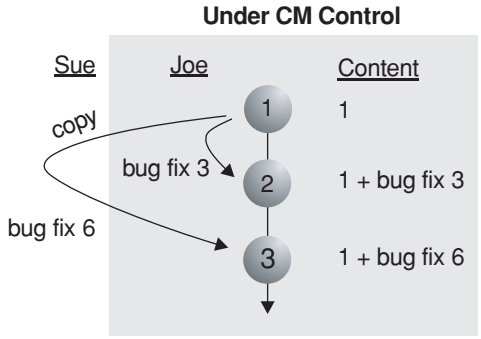

Figure 1-2 Serial development problems.

changes from the latest version of the file (see Figure 1-2). This problem is usually exposed when a bug reappears in the latest build of a system. Because the assumption is made that the bug has been fixed and already verified, full regression tests might not be run. If so, the reintroduced bug could make its way into released software.

One of the key things an SCM tool must support is the capability to modify the same files concurrently and to integrate or merge the changes made in parallel at the appropriate time. We see this requirement manifested in parallel development activities such as those just mentioned, as well as in a need to allow a single developer to work on two or more activities simultaneously, or allow multiple developers to work in isolation on a single feature before that feature is incorporated into the project. This might mean integrating at check-in time for two developers working on the same project, or it might mean scheduling an integration action when merging changes into the latest development work. By providing this support, developers aren't forced to work around the system or be blocked.

1.1.10 Integrate Early and Often

During integration, you will discover problems with interfaces and misunderstandings in the design. Both of these problems can have a major impact on the development schedule, so early discovery is essential. Plan integration points early in the development life cycle, and continue integrating often over the course of a software-development project.

If you do too good of a job at developer isolation (see the "Maintain Stable and Consistent Workspaces" section, earlier in this chapter), you might establish a model in which it is easy to remain too isolated. This problem can occur regardless of the tool you use because it is more often related to how you use a tool rather than the tool itself.

Development workspaces should be under the developer's control; however, project managers and integrators must have an automated and enforceable way to ensure that developers keep current with ongoing project changes. This is the classic dilemma of developer isolation versus project integration. The balance to be maintained here is to integrate as early and as often

as possible without negatively impacting productivity (see the section "Isolation and Integration with ClearCase" in Chapter 9, "Integration").

1.1.11 Ensure Reproducibility of Software Builds

It is often necessary to find out how a software build was constructed and what went into its construction, to debug a problem or reproduce the same build. You must establish the proper procedures and provide sufficient automation to record who did the build, what went into each executable or library that was built, which machine it was built on, what OS version was running on that machine, and what command-line options were specified in the build step. When you have this build audit information, it is useful to have tools that enable you to do reporting. In particular, being able to compare two builds can often help debug problems. Sometimes bugs can be introduced simply by changing the optimization switches to the compiler.

Without being able to reproduce a software build, you will be unable to perform system maintenance and, therefore, to support a system distributed to your customers.

1.2 SCM Tools and SCM Process

SCM best practices are achieved by applying both processes and tools to a software-development project. This section briefly introduces both.

1.2.1 SCM Tools

SCM tools are software tools that automate and facilitate the application of the SCM best practices. As with a compiler, debugger, and editor, an SCM tool is an essential part of every software engineer's tool kit today.

It is unrealistic to try to maintain effective SCM without an SCM tool. In early SCM project environments, one or more individuals acted as the CM librarians. These librarians handed out pieces of software for people to work on, diligently recorded who had which pieces, and logged in new versions as people turned in changes. This approach is not competitive today because it is too slow, is too prone to error, and does not scale.

The goal of successful SCM is to allow as much change as possible while still maintaining control of the software. SCM tools help automate tedious, manual, and error-prone pieces of the SCM process, and can ensure that your project can support all of the SCM best practices.

1.2.2 SCM Process

A process defines the steps by which you perform a specific task or set of tasks. An SCM process is the way SCM is performed on your project—specifically, how an SCM tool is applied to accomplish a set of tasks.

A key mistake most people make is to assume that an SCM tool will, in and of itself, solve their SCM problems or support their SCM requirements. This is wrong! The picture will not hang itself if you buy a hammer and nails. It is not the tool itself that solves a problem, but rather the

application of that tool. How you apply the SCM tool to your development environment is called the usage model, or SCM process. It is this model or process that will in part determine how successfully you address your SCM issues.

1.3 Summary

This chapter helped define software configuration management in simple terms as the mechanisms used to control the evolution of a software project. An understanding of what we mean by software configuration management is crucial because if we don't know what we want to do, we have no hope of converging on a good software-development environment. To enable this understanding, we specifically discussed software-development best practices and how they are enabled by a good SCM system. We also introduced the concepts of the SCM tools and processes that are used to implement those best practices. In Chapter 2, "Growing into Your SCM Solution," we begin to explain how to use those concepts and processes efficiently and effectively.