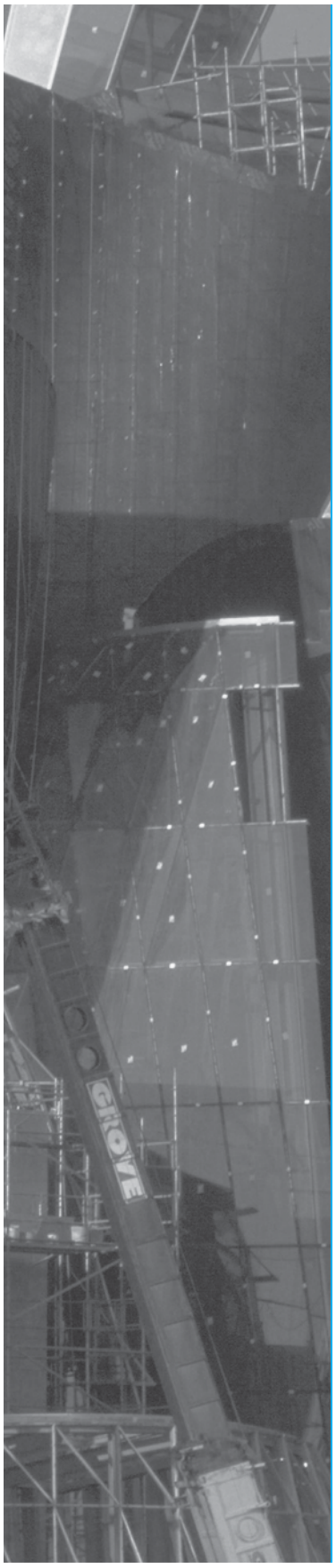

25

Configuration management

Objectives

The objective of this chapter is to introduce you to configuration management processes and tools. When you have read the chapter, you will:

- understand the processes and procedures involved in software change management;
- know the essential functionality that must be provided by a version management system, and the relationships between version management and system building;
- understand the differences between a system version and a system release, and know the stages in the release management process.

Contents

Software systems always change during development and use. Bugs are discovered and have to be fixed. System requirements change, and you have to implement these changes in a new version of the system. New versions of hardware and system platforms become available and you have to adapt your systems to work with them. Competitors introduce new features in their system that you have to match. As changes are made to the software, a new version of a system is created. Most systems, therefore, can be thought of as a set of versions, each of which has to be maintained and managed.

Configuration management (CM) is concerned with the policies, processes, and tools for managing changing software systems. You need to manage evolving systems because it is easy to lose track of what changes and component versions have been incorporated into each system version. Versions implement proposals for change, corrections of faults, and adaptations for different hardware and operating systems. There may be several versions under development and in use at the same time. If you don't have effective configuration management procedures in place, you may waste effort modifying the wrong version of a system, deliver the wrong version of a system to customers, or forget where the software source code for a particular version of the system or component is stored.

Configuration management is useful for individual projects as it is easy for one person to forget what changes have been made. It is essential for team projects where several developers are working at the same time on a software system. Sometimes these developers are all working in the same place but, increasingly, development teams are distributed with members in different locations across the world. The use of a configuration management system ensures that teams have access to information about a system that is under development and do not interfere with each other's work.

The configuration management of a software system product involves four closely related activities (Figure 25.1):

1. *Change management* This involves keeping track of requests for changes to the software from customers and developers, working out the costs and impact of making these changes, and deciding if and when the changes should be implemented.
2. *Version management* This involves keeping track of the multiple versions of system components and ensuring that changes made to components by different developers do not interfere with each other.
3. *System building* This is the process of assembling program components, data, and libraries, and then compiling and linking these to create an executable system.
4. *Release management* This involves preparing software for external release and keeping track of the system versions that have been released for customer use.

Configuration management involves dealing with a large volume of information and many configuration management tools have been developed to support CM

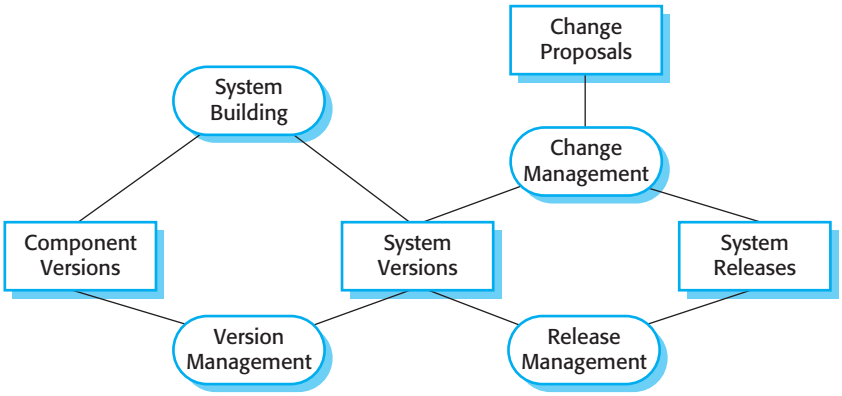

Figure 25.1 Configuration management activities

processes. These range from simple tools that support a single configuration management task, such as bug tracking, to complex and expensive integrated toolsets that support all configuration management activities.

Configuration management policies and processes define how to record and process proposed system changes, how to decide what system components to change, how to manage different versions of the system and its components, and how to distribute changes to customers. Configuration management tools are used to keep track of change proposals, store versions of system components, build systems from these components, and track the releases of system versions to customers.

Configuration management is sometimes considered to be part of software quality management (covered in Chapter 24), with the same manager having both quality management and configuration management responsibilities. When a new version of the software has been implemented, it is handed over by the development team to the quality assurance (QA) team. The QA team checks that the system quality is acceptable. If so, it then becomes a controlled system, which means that all changes to the system have to be agreed on and recorded before they are implemented.

The definition and use of configuration management standards is essential for quality certification in both the ISO 9000 and the CMM and CMMI standards (Ahern et al., 2001; Bamford and Deibler, 2003; Paulk et al., 1995; Peach, 1996). These CM standards can be based on generic CM standards that have been developed by bodies such as the IEEE. For example, standard IEEE 828-1998 is a standard for configuration management plans. These standards focus on CM processes and the documents produced during the CM process. Using the external standards as a starting point, companies then develop more detailed, company-specific standards that are tailored to their specific needs.

One of the problems of configuration management is that different companies talk about the same concepts using different terms. There are historical

Term	Explanation
Configuration item or software configuration item (SCI)	Anything associated with a software project (design, code, test data, document, etc.) that has been placed under configuration control. There are often different versions of a configuration item. Configuration items have a unique name.
Configuration control	The process of ensuring that versions of systems and components are recorded and maintained so that changes are managed and all versions of components are identified and stored for the lifetime of the system.
Version	An instance of a configuration item that differs, in some way, from other instances of that item. Versions always have a unique identifier, which is often composed of the configuration item name plus a version number.
Baseline	A baseline is a collection of component versions that make up a system. Baselines are controlled, which means that the versions of the components making up the system cannot be changed. This means that it should always be possible to re-create a baseline from its constituent components.
Codeline	A codeline is a set of versions of a software component and other configuration items on which that component depends.
Mainline	A sequence of baselines representing different versions of a system.
Release	A version of a system that has been released to customers (or other users in an organization) for use.
Workspace	A private work area where software can be modified without affecting other developers who may be using or modifying that software.
Branching	The creation of a new codeline from a version in an existing codeline. The new codeline and the existing codeline may then develop independently.
Merging	The creation of a new version of a software component by merging separate versions in different codelines. These codelines may have been created by a previous branch of one of the codelines involved.
System building	The creation of an executable system version by compiling and linking the appropriate versions of the components and libraries making up the system.

Figure 25.2 CM terminology

reasons for this. Military software systems were probably the first systems in which configuration management was used and the terminology for these systems reflected the processes and procedures already in place for hardware configuration management. Commercial systems developers were not familiar with the military procedures or terminology and so often invented their own terms. Agile methods also have devised new terminology, sometimes introduced deliberately to distinguish the agile approach from traditional CM methods. Figure 25.2 defines the configuration management terminology that I use in this chapter.

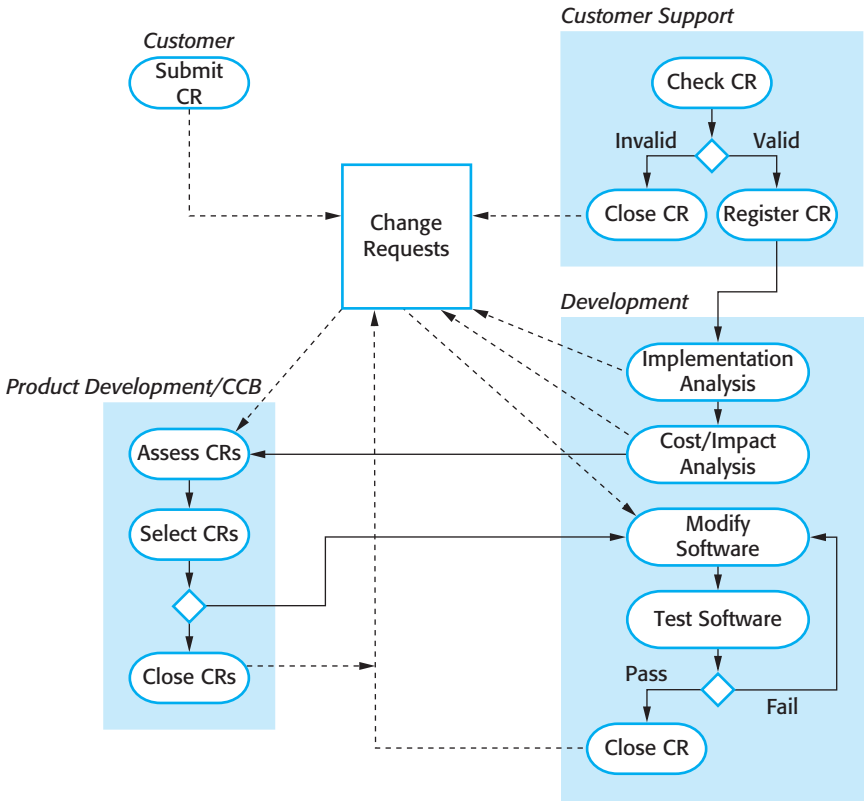

Figure 25.3 The change management process

25.1 Change management

Change is a fact of life for large software systems. Organizational needs and requirements change during the lifetime of a system, bugs have to be repaired, and systems have to adapt to changes in their environment. To ensure that the changes are applied to the system in a controlled way, you need a set of tool-supported, change management processes. Change management is intended to ensure that the evolution of the system is a managed process and that priority is given to the most urgent and cost-effective changes.

The change management process is concerned with analyzing the costs and benefits of proposed changes, approving those changes that are worthwhile, and tracking which components in the system have been changed. Figure 25.3 is a model of a change management process that shows the principal change management activities. There are many variants of this process in use but, to be effective, change management processes should always have a means of checking, costing, and approving changes. This process should come into effect when the software is handed over for release to customers or for deployment within an organization.

The change management process is initiated when a 'customer' completes and submits a change request describing the change required to the system. This could

Change Request Form

Project: SICSA/AppProcessing **Number:** 23/02
Change requester: I. Sommerville **Date:** 20/01/09
Requested change: The status of applicants (rejected, accepted, etc.) should be shown visually in the displayed list of applicants.

Change analyzer: R. Looek **Analysis date:** 25/01/09
Components affected: ApplicantListDisplay, StatusUpdater

Associated components: StudentDatabase

Change assessment: Relatively simple to implement by changing the display color according to status. A table must be added to relate status to colors. No changes to associated components are required.

Change priority: Medium
Change implementation:
Estimated effort: 2 hours
Date to SGA app. team: 28/01/09 **CCB decision date:** 30/01/09
Decision: Accept change. Change to be implemented in Release 1.2
Change implementor: **Date of change:**
Date submitted to QA: **QA decision:**
Date submitted to CM:
Comments:

Figure 25.4 A partially completed change request form

be a bug report, where the symptoms of the bug are described, or a request for additional functionality to be added to the system. Some companies handle bug reports and new requirements separately but, in principle, these are both simply change requests. Change requests may be submitted using a change request form (CRF). I use the term 'customer' here to include any stakeholder who is not part of the development team, so changes may be suggested, for example, by the marketing department in a company.

Electronic change request forms record information that is shared between all groups involved in change management. As the change request is processed, information is added to the CRF to record decisions made at each stage of the process. At any time, it therefore represents a snapshot of the state of the change request. As well as recording the change required, the CRF records the recommendations regarding the change; the estimated costs of the change; and the dates when the change was requested, approved, implemented, and validated. The CRF may also include a section where a developer outlines how the change may be implemented.

An example of a partially completed change request form is shown in Figure 25.4. This is an example of a type of CRF that might be used in a large complex systems engineering project. For smaller projects, I recommend that change requests should be formally recorded and the CRF should focus on describing the change required, with less emphasis on implementation issues. As a system developer, you decide how to implement that change and estimate the time required for this.

After a change request has been submitted, it is checked to ensure that it is valid. The checker may be from a customer or application support team or, for internal requests, may be a member of the development team. Checking is necessary because not all change requests require action. If the change request is a bug report, the bug may have already been reported. Sometimes, what people believe to be problems are actually misunderstandings of what the system is expected to do. On occasions, people request features that have already been implemented but which they don't know about. If any of these are true, the change request is closed and the form is updated with the reason for closure. If it is a valid change request, it is then logged as an outstanding request for subsequent analysis.

For valid change requests, the next stage of the process is change assessment and costing. This is usually the responsibility of the development or maintenance team as they can work out what is involved in implementing the change. The impact of the change on the rest of the system must be checked. To do this, you have to identify all of the components affected by the change. If making the change means that further changes elsewhere in the system are needed, this will obviously increase the cost of change implementation. Next, the required changes to the system modules are assessed. Finally, the cost of making the change is estimated, taking into account the costs of changing related components.

Following this analysis, a separate group should then decide if it is cost-effective from a business perspective to make the change to the software. For military and government systems, this group is often called the change control board (CCB). In industry, it may be called something like a 'product development group', who is responsible for making decisions about how a software system should evolve. This group should review and approve all change requests, unless the changes simply involve correcting minor errors on screen displays, webpages, or documents. These small requests should be passed to the development team without detailed analysis, as such an analysis could cost more than implementing the change.

The CCB or product development group considers the impact of the change from a strategic and organizational rather than a technical point of view. It decides whether the change in question is economically justified and prioritizes accepted changes for implementation. Accepted changes are passed back to the development group; rejected change requests are closed and no further action is taken. Significant factors that should be taken into account in deciding whether or not a change should be approved are:

1. *The consequences of not making the change* When assessing a change request, you have to consider what will happen if the change is not implemented. If the change is associated with a reported system failure, the seriousness of that failure has to be taken into account. If the system failure causes the system to crash, this is very serious and failure to make the change may disrupt the operational use of the system. On the other hand if the failure has a minor effect, such as incorrect colors on a display, then it is not important to fix the problem quickly, so the change should have a low priority.

Customers and changes

Agile methods emphasize the importance of involving customers in the change prioritization process. The customer representative helps the team decide on the changes that should be implemented in the next development iteration. Although this can be effective for systems that are in development for a single customer, it can be a problem in product development where there is no real customer working with the team. In those cases, the team has to make their own decisions on change prioritization.

http://www.SoftwareEngineering-9.com/Web/CM/agilechanges.html

2. *The benefits of the change* Is the change something that will benefit many users of the system or is it simply a proposal that will primarily be of benefit to the change proposer?
3. *The number of users affected by the change* If only a few users are affected, then the change may be assigned a low priority. In fact, making the change may be inadvisable if it could have adverse effects on the majority of system users.
4. *The costs of making the change* If making the change affects many system components (hence increasing the chances of introducing new bugs) and/or takes a lot of time to implement, then the change may be rejected, given the elevated costs involved.
5. *The product release cycle* If a new version of the software has just been released to customers, it may make sense to delay the implementation of the change until the next planned release (see Section 25.3).

Change management for software products (e.g., a CAD system product) rather than systems that are specifically developed for a certain customer, has to be handled in a slightly different way. In software products, the customer is not directly involved in decisions about system evolution, so the relevance of the change to the customer's business is not an issue. Change requests for these products come from the customer support team, the company marketing team and the developers themselves. These requests may reflect suggestions and feedback from customers or analyses of what is offered by competing products.

The customer support team may submit change requests associated with bugs that have been discovered and reported by customers after the software has been released. Customers may use a webpage or e-mail to report bugs. A bug management team then checks that the bug reports are valid and translates them into formal system change requests. Marketing staff meet with customers and investigate competitive products. They may suggest changes that should be included to make it easier to sell a new version of a system to new and existing customers. The system developers themselves may have some good ideas about new features that can be added to the system.

The change request process shown in Figure 25.3 is used after a system has been released to customers. During development, when new versions of the system are

```
// SICSA project (XEP 6087)
//
// APP-SYSTEM/AUTH/RBAC/USER_ROLE
//
// Object: currentRole
// Author: R. Looek
// Creation date: 13/11/2009
//
// © St Andrews University 2009
//
// Modification history
// Version   Modifier   Date          Change       Reason
// 1.0       J. Jones   11/11/2009    Add header   Submitted to CM
// 1.1       R. Looek   13/11/2009    New field    Change req. R07/02
```

Figure 25.5
Derivation history

created through daily (or more frequent) system builds, a simpler change management process is normally used. Problems and changes must still be recorded, but changes that only affect individual components and modules need not be independently assessed. They are passed directly to the system developer. The system developer either accepts them or makes a case for why they are not required. However, an independent authority, such as the system architect, should assess and prioritize changes that affect those system modules that have been produced by different development teams.

In some agile methods, such as extreme programming, customers are directly involved in deciding whether a change should be implemented. When they propose a change to the system requirements, they work with the team to assess the impact of that change and then decide whether the change should take priority over the features planned for the next increment of the system. However, changes that involve software improvement are left to the discretion of the programmers working on the system. Refactoring, where the software is continually improved, is not seen as an overhead but rather as a necessary part of the development process.

As the development team changes software components, they should maintain a record of the changes made to each component. This is sometimes called the derivation history of a component. A good way to keep the derivation history is in a standardized comment at the beginning of the component source code (Figure 25.5). This comment should reference the change request that triggered the software change. You can then write simple scripts that scan all components and process the derivation histories to produce component change reports. For documents, records of changes incorporated in each version are usually maintained in a separate page at the front of the document. I discuss this in the web chapter on documentation.

Change management is usually supported by specialized software tools. These may be relatively simple web-based tools such as Bugzilla, which is used to report problems with many open source systems. Alternatively, more complex tools may be used to automate the entire process of handling change requests from initial customer proposal to change approval.

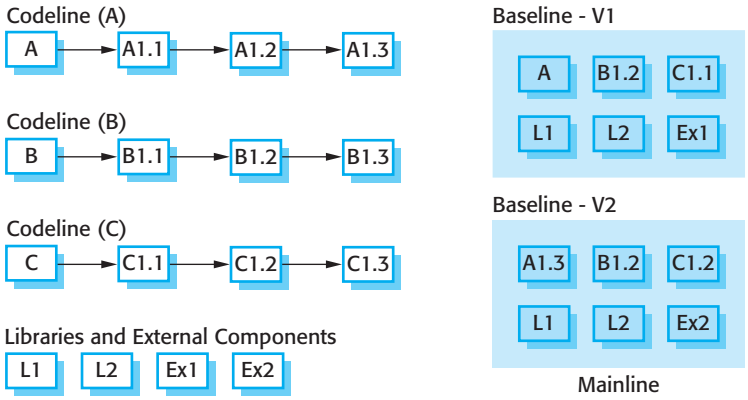

Figure 25.6 Codelines and baselines

25.2 Version management

Version management (VM) is the process of keeping track of different versions of software components or configuration items and the systems in which these components are used. It also involves ensuring that changes made by different developers to these versions do not interfere with each other. You can, therefore, think of version management as the process of managing codelines and baselines.

Figure 25.6 illustrates the differences between codelines and baselines. Essentially, a codeline is a sequence of versions of source code with later versions in the sequence derived from earlier versions. Codelines normally apply to components of systems so that there are different versions of each component. A baseline is a definition of a specific system. The baseline therefore specifies the component versions that are included in the system plus a specification of the libraries used, configuration files, etc. In Figure 25.6, you can see that different baselines use different versions of the components from each codeline. In the diagram, I have shaded the boxes representing components in the baseline definition to indicate that these are actually references to components in a codeline. The mainline is a sequence of system versions developed from an original baseline.

Baselines may be specified using a configuration language, which allows you to define what components are included in a version of a particular system. It is possible to explicitly specify a specific component version (X.1.2, say) or simply to specify the component identifier (X). If you use the identifier, this means that the most recent version of the component should be used in the baseline.

Baselines are important because you often have to re-create a specific version of a complete system. For example, a product line may be instantiated so that there are individual system versions for different customers. You may have to re-create the version delivered to a specific customer if, for example, that customer reports bugs in their system that have to be repaired.

To support version management, you should always use version management tools (sometimes called version control systems or source code control systems).

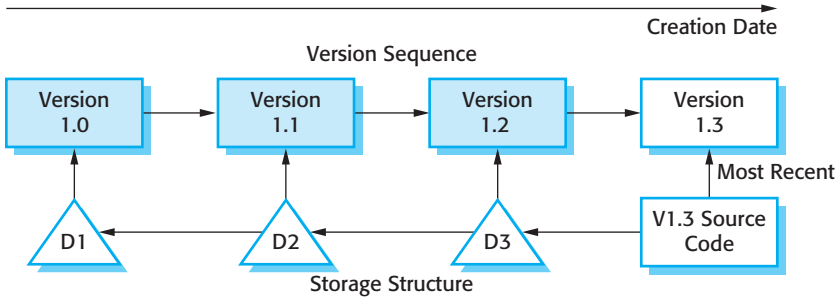

Figure 25.7 Storage management using deltas

These tools identify, store, and control access to the different versions of components. There are many different version management systems available, including widely used open source systems such as CVS and Subversion (Pilato et al., 2004; Vesperman, 2003).

Version management systems normally provide a range of features:

1. *Version and release identification* Managed versions are assigned identifiers when they are submitted to the system. These identifiers are usually based on the name of the configuration item (e.g., ButtonManager), followed by one or more numbers. So ButtonManager 1.3 means the third version in codeline 1 of the ButtonManager component. Some CM systems also allow the association of attributes with versions (e.g., mobile, smallscreen), which can also be used for version identification. A consistent identification system is important because it simplifies the problem of defining configurations. It makes it simpler to use shorthand references (e.g., *.V2 meaning version 2 of all components).

2. *Storage management* To reduce the storage space required by multiple versions of components that differ only slightly, version management systems usually provide storage management facilities. Instead of keeping a complete copy of each version, the system stores a list of differences (deltas) between one version and another. By applying these to a source version (usually the most recent version), a target version can be re-created. This is illustrated in Figure 25.7.

3. *Change history recording* All of the changes made to the code of a system or component are recorded and listed. In some systems, these changes may be used to select a particular system version. This involves tagging components with keywords describing the changes made. You then use these tags to select the components to be included in a baseline.

4. *Independent development* Different developers may be working on the same component at the same time. The version management system keeps track of components that have been checked out for editing and ensures that changes made to a component by different developers do not interfere.

5. *Project support* A version management system may support the development of several projects, which share components. In project support systems, such as

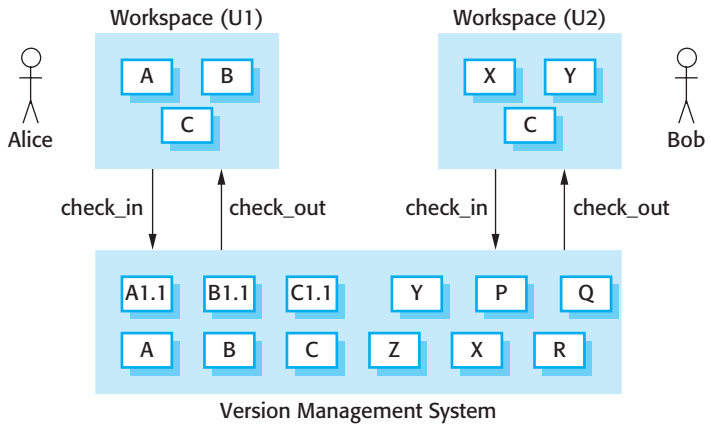

Figure 25.8 Check-in and check-out from a version repository

CVS (Vesperman, 2003), it is possible to check in and check out all of the files associated with a project rather than having to work with one file or directory at a time.

When version management systems were first developed, storage management was one of their most important functions. The storage management features in a version control system reduce the disk space required to maintain all system versions. When a new version is created, the system simply stores a delta (a list of differences) between the new version and the older version used to create that new version (shown in the bottom part of Figure 25.7). In Figure 25.7, the shaded boxes represent earlier versions of a component that are automatically re-created from the most recent component version. Deltas are usually stored as lists of changed lines and, by applying these automatically, one version of a component can be created from another. As it is most likely that the most recent version of a component will be used, most systems store that version in full. The deltas then define how to re-create earlier system versions.

Most software development is a team activity, so situations often arise where different team members work on the same component at the same time. For example, let's say Alice is making some changes to a system, which involves changing components A, B, and C. At the same time, Bob is working on changes and these require making changes to components X, Y, and C. Both Alice and Bob are therefore changing C. It's important to avoid these changes interfering with each other—Bob's changes to C overwriting Alice's or vice versa.

To support independent development without interference, version management systems use the concept of a public repository and a private workspace. Developers check out components from the public repository into their private workspace and may change these as they wish in their private workspace. When their changes are complete, they check in the components to the repository. This is illustrated in Figure 25.8. If two or more people are working on a component at the same time, each must check out the component from the repository. If a component has been checked out, the version management system will normally warn other users wanting

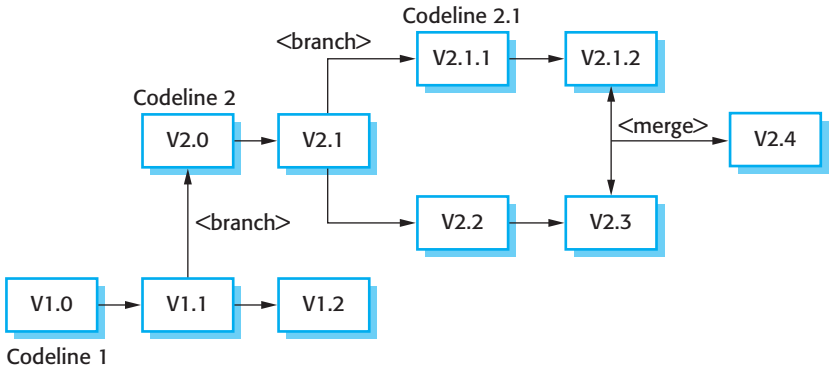

Figure 25.9 Branching and merging

to check out that component that it has been checked out by someone else. The system will also ensure that when the modified components are checked in, the different versions are assigned different version identifiers and are separately stored.

A consequence of the independent development of the same component is that codelines may branch. Rather than a linear sequence of versions that reflect changes to the component over time, there may be several independent sequences, as shown in Figure 25.9. This is normal in system development, where different developers work independently on different versions of the source code and change it in different ways.

At some stage, it may be necessary to merge codeline branches to create a new version of a component that includes all changes that have been made. This is also shown in Figure 25.9 where versions 2.1.2 and 2.3 of the component are merged to create version 2.4. If the changes made involve completely different parts of the code, the component versions may be merged automatically by the version management system by combining the deltas that apply to the code. More often, there are overlaps between the changes made and they interfere with each other. A developer has to check for clashes and modify the changes so that they are compatible.

25.3 System building

System building is the process of creating a complete, executable system by compiling and linking the system components, external libraries, configuration files, etc. System building tools and version management tools must communicate as the build process involves checking out component versions from the repository managed by the version management system. The configuration description used to identify a baseline is also used by the system building tool.

Building is a complex process, which is potentially error-prone, as there may be three different system platforms involved (Figure 25.10):

1. The development system, which includes development tools such as compilers, source code editors, etc. Developers check out code from the version management system into a private workspace before making changes to the system.

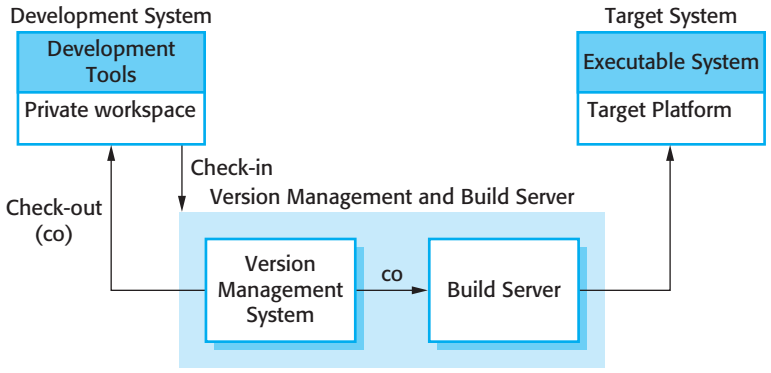

Figure 25.10 Development, build, and target platforms

They may wish to build a version of a system for testing in their development environment before committing changes that they have made to the version management system. This involves using local build tools that use checked-out versions of components in the private workspace.

2. The build server, which is used to build definitive, executable versions of the system. This interacts closely with the version management system. Developers check in code to the version management system before it is built. The system build may rely on external libraries that are not included in the version management system.

3. The target environment, which is the platform on which the system executes. This may be the same type of computer that is used for the development and build systems. However, for real-time and embedded systems, the target environment is often smaller and simpler than the development environment (e.g., a cell phone). For large systems, the target environment may include databases and other COTS systems that cannot be installed on development machines. In both of these cases, it is not possible to build and test the system on the development computer or on the build server.

The development system and the build server may both interact with the version management system. The VM system may either be hosted on the build server or on a dedicated server. For embedded systems, a simulation environment may be installed in the development environment for testing, rather than using the actual embedded system platform. These simulators may provide better debugging support than is available on an embedded system. However, it is very difficult to simulate the behavior of an embedded system in every respect. You therefore have to run system tests on the actual platform where the system will execute, as well as the system simulator.

System building involves assembling a large amount of information about the software and its operating environment. Therefore, for anything apart from very small systems, it always makes sense to use an automated build tool to create a system build (Figure 25.11). Notice that you don't just need the source code files that are involved in the build. You may have to link these with externally provided

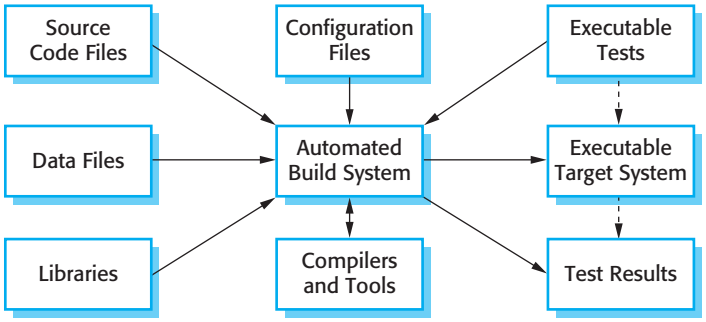

Figure 25.11 System building

libraries, data files (such as a file of error messages), and configuration files that define the target installation. You may have to specify the versions of the compiler and other software tools that are to be used in the build. Ideally, you should be able to build a complete system with a single command or mouse click.

There are many build tools available and a build system may provide some or all of the following features:

1. *Build script generation* If necessary, the build system should analyze the program that is being built, identify dependent components, and automatically generate a build script (sometimes called a configuration file). The system should also support the manual creation and editing of build scripts.
2. *Version management system integration* The build system should check out the required versions of components from the version management system.
3. *Minimal recompilation* The build system should work out what source code needs to be recompiled and set up compilations if required.
4. *Executable system creation* The build system should link the compiled object code files with each other and with other required files, such as libraries and configuration files, to create an executable system.
5. *Test automation* Some build systems can automatically run automated tests using test automation tools such as JUnit. These check that the build has not been 'broken' by changes.
6. *Reporting* The build system should provide reports about the success or failure of the build and the tests that have been run.
7. *Documentation generation* The build system may be able to generate release notes about the build and system help pages.

The build script is a definition of the system to be built. It includes information about components and their dependencies, and the versions of tools used to compile and link the system. The build script includes the configuration specification so the scripting language used is often the same as the configuration description language.

The configuration language includes constructs to describe the system components to be included in the build and their dependencies.

As compilation is a computationally intensive process, tools to support system building are usually designed to minimize the amount of compilation that is required. They do this by checking if a compiled version of a component is available. If so, there is no need to recompile that component. Therefore, there has to be a way of unambiguously linking the source code of a component with its equivalent object code.

The way that this is done is to associate a unique signature with each file where a source code component is stored. The corresponding object code, which has been compiled from the source code, has a related signature. The signature identifies each source code version and is changed when the source code is edited. By comparing the signatures on the source and object code files, it is possible to decide if the source code component was used to generate the object code component.

There are two types of signatures that may be used:

1. *Modification timestamps* The signature on the source code file is the time and date when that file was modified. If the source code file of a component has been modified after the related object code file, then the system assumes that recompilation to create a new object code file is necessary.

   For example, say components Comp.java and Comp.class have modification signatures of 17:03:05:02:14:2009 and 16:34:25:02:12:2009, respectively. This means that the Java code was modified at 3 minutes and 5 seconds past 5 on the 14th of February 2009 and the compiled version was modified at 34 minutes and 25 seconds past 4 on the 12th of February 2009. In this case, the system would automatically recompile Comp.java because the compiled version does not include changes made to the source code since 12th of February.

2. *Source code checksums* The signature on the source code file is a checksum calculated from data in the file. A checksum function calculates a unique number using the source text as input. If you change the source code (even by one character), this will generate a different checksum. You can therefore be confident that source code files with different checksums are actually different. The checksum is assigned to the source code just before compilation and uniquely identifies the source file. The build system then tags the generated object code file with the checksum signature. If there is no object code file with the same signature as the source code file to be included in a system, then recompilation of the source code is necessary.

As object code files are not normally versioned, the first approach means that only the most recently compiled object code file is maintained in the system. This is normally related to the source code file by name (i.e., it has the same name as the source code file but with a different suffix). Therefore, the source file Comp.Java may generate the object file Comp.class. Because source and object files are linked by name rather than an explicit source file signature, it is not usually possible to build different versions of a source code component into the same directory at the same time, as these would generate object files with the same name.

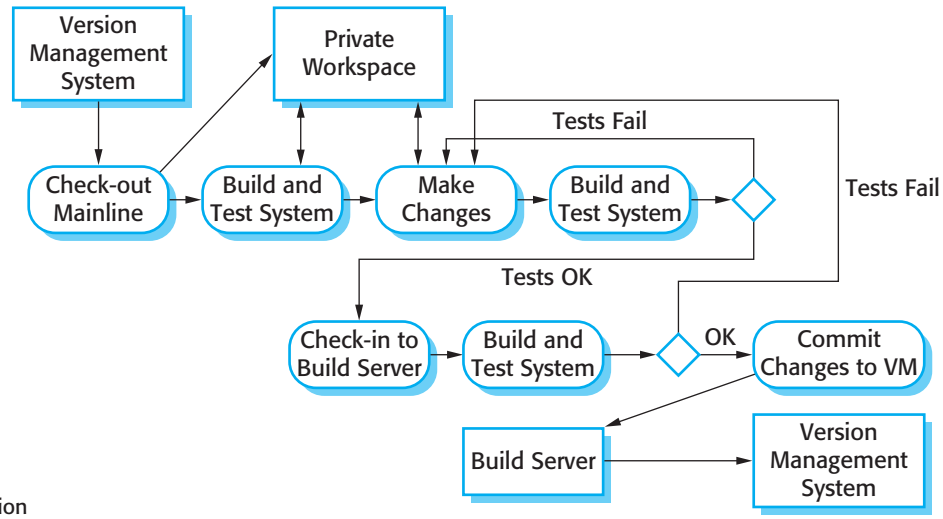

Figure 25.12
Continuous integration

The checksum approach has the advantage of allowing many different versions of the object code of a component to be maintained at the same time. The signature rather than the file name is the link between source and object code. The source code and object code files have the same signature. Therefore, when you recompile a component, it does not overwrite the object code, as would normally be the case when the timestamp is used. Rather, it generates a new object code file and tags it with the source code signature. Parallel compilation is possible and different versions of a component may be compiled at the same time.

Agile methods recommend that very frequent system builds should be carried out with automated testing (sometimes called smoke tests) to discover software problems. Frequent builds may be part of a process of continuous integration, as shown in Figure 25.12. In keeping with the agile methods notion of making many small changes, continuous integration involves rebuilding the mainline frequently, after small source code changes have been made. The steps in continuous integration are:

1. Check out the mainline system from the version management system into the developer's private workspace.
2. Build the system and run automated tests to ensure that the built system passes all tests. If not, the build is broken and you should inform whoever checked in the last baseline system. They are responsible for repairing the problem.
3. Make the changes to the system components.
4. Build the system in the private workspace and rerun system tests. If the tests fail, continue editing.
5. Once the system has passed its tests, check it into the build system but do not commit it as a new system baseline.

6. Build the system on the build server and run the tests. You need to do this in case others have modified components since you checked out the system. If this is the case, check out the components that have failed and edit these so that tests pass on your private workspace.
7. If the system passes its tests on the build system, then commit the changes you have made as a new baseline in the system mainline.

The argument for continuous integration is that it allows problems caused by the interactions between different developers to be discovered and repaired as soon as possible. The most recent system in the mainline is the definitive working system. However, although continuous integration is a good idea, it is not always possible to implement this approach to system building. The reasons for this are:

1. If the system is very large, it may take a long time to build and test. It is therefore impractical to build that system several times per day.
2. If the development platform is different from the target platform, it may not be possible to run system tests in the developer's private workspace. There may be differences in hardware, operating system, or installed software. Therefore more time is required for testing the system.

For large systems or for systems where the execution platform is not the same as the development platform, continuous integration may be impractical. In those circumstances, a daily build system may be used. Features of this are as follows:

1. The development organization sets a delivery time (say 2 P.M.) for system components. If developers have new versions of the components that they are writing, they must deliver them by that time. Components may be incomplete but should provide some basic functionality that can be tested.
2. A new version of the system is built from these components by compiling and linking them to form a complete system.
3. This system is then delivered to the testing team, which carries out a set of predefined system tests. At the same time, the developers are still working on their components, adding to the functionality and repairing faults discovered in previous tests.
4. Faults that are discovered during system testing are documented and returned to the system developers. They repair these faults in a subsequent version of the component.

The advantages of using frequent builds of software are that the chances of finding problems stemming from component interactions early in the process are increased. Frequent building encourages thorough unit testing of components. Psychologically, developers are put under pressure not to 'break the build'; that is,

they try to avoid checking in versions of components that cause the whole system to fail. They are therefore reluctant to deliver new component versions that have not been properly tested. Consequently, less time is spent during system testing discovering and coping with software faults that could have been found by the developer.

25.4 Release management

A system release is a version of a software system that is distributed to customers. For mass- market software, it is usually possible to identify two types of release namely major releases, which deliver significant new functionality, and minor releases, which repair bugs and fix customer problems that have been reported. For example, this book is being written on an Apple Mac computer where the operating system is OS 10.5.8. This means minor release 8 of major release 5 of OS 10. Major releases are very important economically to the software vendor as customers have to pay for these. Minor releases are usually distributed free of charge.

For custom software or software product lines, managing system releases is a complex process. Special releases of the system may have to be produced for each customer and individual customers may be running several different releases of the system at the same time. This means that a software company selling a specialized software product may have to manage tens or even hundreds of different releases of that product. Their configuration management systems and processes have to be designed to provide information about which customers have which releases of the system and the relationship between releases and system versions. In the event of a problem, it may be necessary to reproduce exactly the software that has been delivered to a particular customer.

Therefore when a system release is produced, it must be documented to ensure that it can be re-created exactly in the future. This is particularly important for customized, long-lifetime embedded systems, such as those that control complex machines. Customers may use a single release of these systems for many years and may require specific changes to a particular software system long after its original release date.

To document a release, you have to record the specific versions of the source code components that were used to create the executable code. You must keep copies of the source code files, corresponding executables, and all data and configuration files. You should also record the versions of the operating system, libraries, compilers, and other tools used to build the software. These may be required to build exactly the same system at some later date. This may mean that you have to store copies of the platform software and the tools used to create the system in the version management system along with the source code of the target system.

Preparing and distributing a system release is an expensive process, particularly for mass-market software products. As well as the technical work involved in creating a release distribution, advertising and publicity material have to be prepared and marketing strategies put in place to convince customers to buy the new release of the

Factor	Description
Technical quality of the system	If serious system faults are reported which affect the way in which many customers use the system, it may be necessary to issue a fault repair release. Minor system faults may be repaired by issuing patches (usually distributed over the Internet) that can be applied to the current release of the system.
Platform changes	You may have to create a new release of a software application when a new version of the operating system platform is released.
Lehman's fifth law (see Chapter 9)	This 'law' suggests that if you add a lot of new functionality to a system; you will also introduce bugs that will limit the amount of functionality that may be included in the next release. Therefore, a system release with significant new functionality may have to be followed by a release that focuses on repairing problems and improving performance.
Competition	For mass-market software, a new system release may be necessary because a competing product has introduced new features and market share may be lost if these are not provided to existing customers.
Marketing requirements	The marketing department of an organization may have made a commitment for releases to be available at a particular date.
Customer change proposals	For custom systems, customers may have made and paid for a specific set of system change proposals, and they expect a system release as soon as these have been implemented.

Figure 25.13 Factors influencing system release planning

system. Careful thought must be given to release timing. If releases are too frequent or require hardware upgrades, customers may not move to the new release, especially if they have to pay for it. If system releases are too infrequent, market share may be lost as customers move to alternative systems.

The various technical and organizational factors that you should take into account when deciding on when to release a new version of a system are shown in Figure 25.13.

A system release is not just the executable code of the system. The release may also include:

- configuration files defining how the release should be configured for particular installations;
- data files, such as files of error messages, that are needed for successful system operation;
- an installation program that is used to help install the system on target hardware;
- electronic and paper documentation describing the system;
- packaging and associated publicity that have been designed for that release.

Release creation is the process of creating the collection of files and documentation that includes all of the components of the system release. The executable code of

the programs and all associated data files must be identified in the version management system and tagged with the release identifier. Configuration descriptions may have to be written for different hardware and operating systems and instructions prepared for customers who need to configure their own systems. If machine-readable manuals are distributed, electronic copies must be stored with the software. Scripts for the installation program may have to be written. Finally, when all information is available, an executable master image of the software must be prepared and handed over for distribution to customers or sales outlets.

When planning the installation of new system releases, you cannot assume that customers will always install new system releases. Some system users may be happy with an existing system. They may consider that it is not worth the cost of changing to a new release. New releases of the system cannot, therefore, rely on the installation of previous releases. To illustrate this problem, consider the following scenario:

1. Release 1 of a system is distributed and put into use.
2. Release 2 requires the installation of new data files, but some customers do not need the facilities of release 2 so remain with release 1.
3. Release 3 requires the data files installed in release 2 and has no new data files of its own.

The software distributor cannot assume that the files required for release 3 have already been installed in all sites. Some sites may go directly from release 1 to release 3, skipping release 2. Some sites may have modified the data files associated with release 2 to reflect local circumstances. Therefore, the data files must be distributed and installed with release 3 of the system.

The marketing and packaging costs associated with new releases of software products are high so product vendors usually only create new releases for new platforms or to add significant new functionality. They then charge users for this new software. When problems are discovered in an existing release, the software vendors make patches to repair the existing software available on a website to be downloaded by customers.

The problem with using downloadable patches is that many customers may never discover the existence of these problem repairs and may not understand why they should be installed. They may instead continue using their existing, faulty system with the consequent risks to their business. In some situations, where the patch is designed to repair security loopholes, the risks of failing to install the patch can mean that the business is susceptible to external attacks. To avoid these problems, mass-market software vendors, such as Adobe, Apple, and Microsoft, usually implement automatic updating where systems are updated whenever a new minor release becomes available. However, this does not usually work for custom systems because these systems do not exist in a standard version for all customers.

KEY POINTS

- Configuration management is the management of an evolving software system. When maintaining a system, a CM team is put in place to ensure that changes are incorporated into the system in a controlled way and that records are maintained with details of the changes that have been implemented.
- The main configuration management processes are concerned with change management, version management, system building, and release management. Software tools are available to support all of these processes.
- Change management involves assessing proposals for changes from system customers and other stakeholders and deciding if it is cost-effective to implement these in a new version of a system.
- Version management involves keeping track of the different versions of software components that are created as changes are made to them.
- System building is the process of assembling system components into an executable program to run on a target computer system.
- Software should be frequently rebuilt and tested immediately after a new version has been built. This makes it easier to detect bugs and problems that have been introduced since the last build.
- System releases include executable code, data files, configuration files, and documentation. Release management involves making decisions on system release dates, preparing all information for distribution, and documenting each system release.

FURTHER READING

Configuration Management Principles and Practice. This very comprehensive book covers standards and traditional approaches to CM as well as CM approaches that are more appropriate to modern processes, such as agile software development. (Anne Mette Jonassen Hass, Addison-Wesley, 2002.)

Software Configuration Management Patterns: Effective Teamwork, Practical Integration. A relatively short, easy-to-read book that gives good practical advice on configuration management practice, especially for agile methods of development. (S. P. Berczuk with B. Appleton, Addison-Wesley, 2003.)

'High-level Best Practices in Software Configuration Management'. This web article, written by staff at a CM tool supplier, is an excellent introduction to good practice in software configuration management (L. Wingerd and C. Seiwald, 2006.) http://www.perforce.com/perforce/papers/bestpractices.html.

'Agile Configuration Management for Large Organizations'. This web article describes configuration management practices that can be used in agile development processes, with a particular emphasis on how these can scale to large projects and companies. (P. Schuh, 2007.) http://www.ibm.com/developerworks/rational/library/mar07/schuh/index.html.

EXERCISES

25.1. Suggest five possible problems that could arise if a company does not develop effective configuration management policies and processes.

25.2. What are the benefits of using a change request form as the central document in the change management process?

25.3. Describe six essential features that should be included in a tool to support change management processes.

25.4. Explain why it is essential that every version of a component should be uniquely identified. Comment on the problems of using a version identification scheme that is simply based on version numbers.

25.5. Imagine a situation where two developers are simultaneously modifying three different software components. What difficulties might arise when they try to merge the changes that they have made?

25.6. Software is increasingly being developed by teams where the team members are working at different locations. Suggest features in a version management system that may be required to support this distributed software development.

25.7. Describe the difficulties that may arise when building a system from its components. What particular problems might occur when a system is built on a host computer for some target machine?

25.8. With reference to system building, explain why you may sometimes have to maintain obsolete computers on which large software systems were developed.

25.9. A common problem with system building occurs when physical file names are incorporated in system code and the file structure implied in these names differs from that of the target machine. Write a set of programmer's guidelines that helps avoid this and any other system-building problems that you can think of.

25.10. Describe five factors that should be taken into account by engineers during the process of building a release of a large software system.

REFERENCES

Ahern, D. M., Clouse, A. and Turner, R. (2001). *CMMI Distilled*. Reading, Mass.: Addison-Wesley.

Bamford, R. and Deibler, W. J. (2003). ' ISO 9001:2000 for Software and Systems Providers: An Engineering Approach'. Boca Raton, FL: CRC Press.

Paulk, M. C., Weber, C. V., Curtis, B. and Chrissis, M. B. (1995). *The Capability Maturity Model: Guidelines for Improving the Software Process*. Reading, Mass.: Addison-Wesley.

Peach, R. W. (1996). *The ISO 9000 Handbook, 3rd edition*. New York: Irwin Professional Pub.

Pilato, C. M., Collins-Sussman, B. and Fitzpatrick, B. W. (2004). *Version Control with Subversion*. Sebastopol, Calif.: O'Reilly Media Inc.

Vesperman, J. (2003). *Essential CVS*. Sebastopol, Calif.: O'Reilly and Associates.