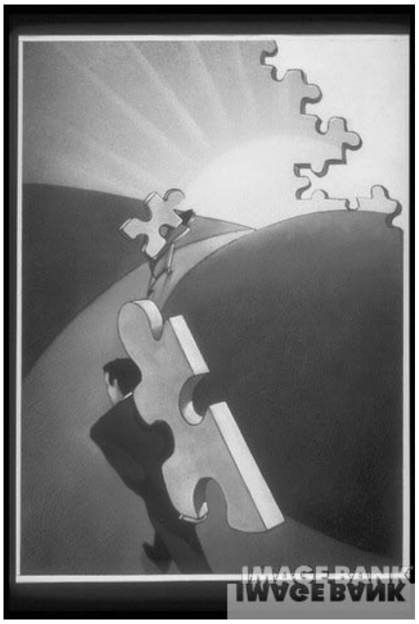
Software Configuration Management Patterns: Effective Teamwork, Practical Integration



By Steve Berczuk
with Brad Appleton

Software Configuration Management Patterns: Effective Teamwork, Practical Integration

By Steve Berczuk
with Brad Appleton

Table of Contents

Preface

Configuration management is not what I do. I am not a configuration management person. I am not an organizational-anthropology person. However, I discovered early on that understanding organizations, software architecture, and configuration management were essential to doing my job as a software developer. I also find this systems perspective on software engineering interesting. I build software systems, and configuration management is a very important, and often neglected, part of building software systems. In this book I hope that I can show you how to avoid some of the problems that I have encountered so that you can build systems more effectively with your team.

I should probably explain what I mean in making the distinction between SCM people and people who build software systems. The stereotype is that configuration management people are concerned with tools and control. They are conservative, and they prefer slow predictable progress. They are also "the few" as compared with" the many" developers in an organization. Software engineers (so the stereotype goes) are reckless. They want to build things fast, and they are confident that they can code their way out of any situation. These are extreme stereotypes, and in my experience, the good software engineers and the good release / quality assurance / configuration management people have a common goal: they are focused on delivering quality systems with the least amount of wasted effort.

Good configuration management practice is the not the silver bullet to building systems on time, just as patterns, extreme programming, the Unified Process, or anything else that you might hear about. It is however, a part of the toolkit that most people ignore because they fear "process," often because of bad experiences in the past. (Weigers 2002)

This book describes some common software configuration management practices. The book will be particularly interesting to software developers working in small teams who suspect that they are not using software configuration management as effectively as they can. The techniques that we describe are not tool specific. Like any set of patterns or best practices, the ease with which you can apply the patterns may depend on whether or not your tool provides explicit support for it.

Why I wrote this book

I started my software development career with a small R&D group that was based in the Boston area. Aside from the many interesting technical problems we had encountered as part of our jobs, we had the added twist of having joint development projects with a group in our parent company's home base in Rochester, New York. This experience helped me recognize early in my career that software development wasn't just about good design and good coding practices, but also about coordination among people in the same group, and even teams in different cities. Our group took the lead in setting up the mechanics of how we would share code and other artifacts of the development process. We did the usual things to make working together easier such as meetings, teleconferences and e-mail lists. The way that we set up our (and the remote team's) software configuration management system to share code played a very large part in making our collaboration easier.

The people who set up the SCM process for out Boston group used techniques that seemed to have been tried throughout their careers. As I move on to other organizations, I was amazed to find how may places were struggling with the same common problems — problems that I knew had good solutions. This was particularly true because I have been with a number of startups that were only one or two years old when I joined. One to two years is often the stage in a startup where you are hiring enough people that coordination and shared vision are difficult goals to attain.

A few years into my career, I discovered patterns. Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides were just finishing the book *Design Patterns* (Gamma et al. 1995), and the Hillside group was organizing the first Pattern Languages of Program (PLoP) conference. There is a lot of power in the idea of patterns since they are about using the right solution at the right time, and also because patterns are interdisciplinary; they are not just about domain or language specific coding techniques, but about how to build software from all perspectives, from the code to the team. I workshopped a number of papers at the various PLoP conferences that dealt with patterns at the intersection of design, coding, and configuration management (Steve Berczuk 1996b; Stephen P Berczuk 1996a, 1995; Appleton et al. 1998; Cabrera et al. 1999; Steve Berczuk and Appleton 2000).

At one Pattern Languages of Programming (PLoP) conference I met Brad Appleton, who is more of an SCM person than I am. We co-authored a paper about branching patterns (Appleton et al. 1998),just one aspect of SCM. After much encouragement from our peers, I started working on this book.

I hope that this book helps you avoid some common mistakes, either by making you aware of these approaches, or by providing you with documentation you can use to explain methods that you already know about to others in your organization.

Who should read this book

I hope that anyone who builds software and uses a configuration management system can learn from this book. The details of the configuration management problem change depending on the types of systems that you are building, the size of the teams, and the environment that you work in. Since it's probably impossible to write a book that will address everyone's needs and also keep everyone's interest, I had to limit what I was talking about. This book will be most valuable to someone who is building software, or managing a software project, in a small to medium size organization where there is not a lot of defined process. If you are in small company, astartup, or in a small project team in a larger organization, you will benefit most from the lessons in this book. Having said that, even if your organization has a very well defined, heavy, process, that seems to be impeding progress, you'll be able to use the patterns in this book to better focus on the key tasks of SCM.

How to read this book.

The introduction explains some basic concepts for software configuration management (SCM) and also the notation that the diagrams use. Chapter 1 introduces the software configuration management concepts that I use in this book. Chapter 2 talks about some of the forces that influence decisions that you make about what sort of SCM environment that you have. Chapter 3 introduces the concept of patterns and the patterns in this book and how they relate to each other. The rest of the book consists of patterns that illustrate problems and solutions to common SCM problems.

Chapters 1 and 2 define the general problems that this book addresses. To understand the how the patterns fit together, you should read chapter 3 to get an overview of the language. After you have read the first 3 chapters, you can browse the patterns in the rest of the book, starting with an interesting one, and following the ones that relate to your problem. Another approach is to read the patterns in order and form a mental picture of the connections between them.

The references to the other patterns in the book appear in the introductory paragraph for each section, and in the "Unresolved Issues" section at the end of each chapter, using a presentation like this: *ACTIVE DEVELOPMENT LINE (5)*. The number in parantheses is the chapter number that contains the patterns.

Since this is a large field to cover, some of the context and unresolved issues sections don't refer to other patterns, either in the book, or elsewhere, since they haven't been documented. In this case you will see a description about what a pattern might cover.

Origins of this Material

Much of the material in this book has its origins in papers that were written for various Pattern Languages of Programs conferences by myself, Brad Appleton, Ralph Cabrera, and Robert Orenstein. The patterns here have been greatly revised from the original material, but it's appropriate to mention these papers here to acknowledge the roles of others to this work: "Streamed Lines: Branching Patterns for Parallel Software Development" (Appleton et al. 1998) , "Software Reconstruction: Patterns for Reproducing the Build" (Cabrera et al. 1999), "Configuration Management Patterns" (Steve Berczuk 1996b).

About the Photos

The photos that start each chapter are from the the Library Of Congress. All of the photos are from the first half of the twentieth cetntury. With the exception of one photo (the photo for *ACTIVE DEVELOPMENT LINE (5)*), they are from the collection: *Depression Era to World War II ~ FSA/OWI ~ Photographs ~ 1935-1945: America from the Great Depression to World War II: Photographs from the FSA and OWI, ca. 1935-1945*. I chose these pictures because I wanted to provide a visual metaphor for the patterns. Software is an abstract concept, but many of the problems that we solve, particularly the ones about teams, are similar to real world problems. I also have always had an interest in photography and history.

-Steve Berczuk, Arlington Massachusetts, June, 2002.

Contributor’s Preface

Why I co-wrote this book with Steve

I began my software development career in 1987 as a part-time software tools developer to pay for my last year of college. Somehow it “stuck” because I’ve been doing some form of tool development ever since (particularly SCM tools), even when it wasn’t my primary job. I even worked (briefly) for a commercial SCM tool vendor, and part of my job was to stay “current” on the competition. So I amassed as much knowledge as I could about other SCM tools on the market. Even after I changed jobs, I continued my SCM pursuits, and frequented various tool user groups on the Internet.

At one time, I longed to advance the “state of the art” in SCM environments, and kept up with all the latest research. I soon became frustrated with the vast gap between the “state of the art” and the “state of the practice.” I concluded I could do more good by helping advance the state of the practice to better utilize available tools. Not long after that, I discovered software patterns and the patterns community. It was clear these guys were “onto” something important in their combination of analysis and storytelling for disseminating recurring best practices of software design.

At the time, there weren’t many people in the design patterns community that were trying to write-up SCM patterns. SCM is, after all, the “plumbing of software development” to a lot of programmers: everyone acknowledge that they need it, but no one wants to have to dive into it too deeply and get their hands entrenched in it. They just want it to work, and to not have to bother with it all that much.

It didn't take long for me to "hook up" with Steve Berczuk.We wrote several SCM patterns papers together (with Ralph Cabrera) as part of my ACME project at acme.bradapp.net, and later decided to work on this book.We hope this small but cohesive set of core patterns about integration and teamwork helps the unsuspecting developer-cum-project-lead to survive and thrive in successfully leading and coordinating their team's collaborative efforts and integrating the results into working software.

Thank you to my wife Maria for her unending love and support (and our daughter Kaeley), and to my parents for their encouragement. Thanks also to my former manager Arbela for her encouragement, support and friendship.

-- *Brad Appleton*

Acknowledgements

My Editor, Debbie Lafferty for her patience, negotiation skill, and enthusiasm. The production staff, ...(names?)

Everyone who provided feedback on the manuscript, including Hisham Alzanoon, Bruce Angstadt, Stanley Barton, David Bellagio, Phil Brooks, Kyle Brown, Frank Buschmann, Thomas Dave, Bernard Farrell, Linda Fernandez, Jeff Fischer, William Hasling, Kirk Knoernschild, Dmitri Lapyguine, McDonald Michael, James Noble, Damon Poole, Linda Rising, Alan Shalloway, Eric Shaver, Michael Sheetz, Dave Spring, Marianne Tromp, Ross Wetmore, Farrero XavierVerges.

And lastly, I must mention Gillian Kahn, my partner in all things, whose feedback, insight, and especially patience as I finished this project were invaluable to me.

Chapter 0
Introduction

This chapter describes some of the basic concepts, notation, and terminology that we use in this book. The vocabulary of software configuration management is used in various ways in different contexts, and the definitions here are not a comprehensive survey of way that these terms are used. Where we can we have tried to use terminology that is commonly used. This section also provides a basic introduction to the practices of version control, and some suggestions for further reading.

Key Concepts and Terminology

Software configuration management (SCM) comprises factors such as configuration identification, configuration control, status accounting, review, build management, process management, and team work (Dart 1992). SCM practices taken as a whole define how an organization builds and releases products, and identifies and tracks changes. This book concerns itself with the aspects of SCM that have a direct impact on the day-to-day work of the people writing code, and implementing features and changes to that code.

Some of the concepts that developers deal with implicitly, if not by name, are *workspaces*, *codelines,* and *integration*.

A *workspace* is a place where a developer keeps all of the artifacts that they need to accomplish a task. In concrete terms, a workspace can be a directory tree on disk in the developer's working area, or it can be a collection of files maintained in an abstract space by a tool. A workspace is normally associated with particular versions of these artifacts. A workspace also should have a mechanism for constructing executable artifacts from the its contents. For example, if you are developing in Java, your workspace would include:

- Source code (.java files) arranged in the appropriate package structure
- Source code for tests
- Java library files (.jar files)
- Library files for native interfaces, that you do not build (for example, dll files on windows)
- Scripts that define how you build java files into an executable

Sometimes a workspace is managed in the context of an integrated development environment (IDE). A workspace is also associated with one or more codelines.

A *codeline* is the set of source files and other artifacts that comprise some software component as they change over time. Every time that you change a file or other artifact in the version control system, you create a *revision* of that artifact. A codeline contains every version of every artifact along one evolutionary path.

At any point in time, a snapshot of the codeline will contain various revisions of each component in the codeline. Figure 0-1 illustrates this; at one point you have version 1 of both file1.java and File2.java. The next time there is a change to the codeline, the next version of the codeline comprises revision 1 of File1.java and revision 2 of File2.java. Any snapshot of the codeline that contains a collection of certain revisions of every component in the codeline is a *version* of the codeline[1]. If you choose to identify or mark a version as special, you define a *label*. You might label the set of revisions that went into a release, for example.

In the simplest case, you might just have one codeline that includes all of your product code. Components of a codeline evolve at their own rate, and have revisions that we can identify. You can identify a version of the codeline by a label. The version of

1. In general, you can also "tag" different revisions of components to identify a version of the codeline. For example, Version 1 of File2.java and version 3 of File1.java, but there are other, more intuitive ways, of identifying c configuration like this.

the codeline is a snapshot that includes the revisions of the components up to the point of the label.

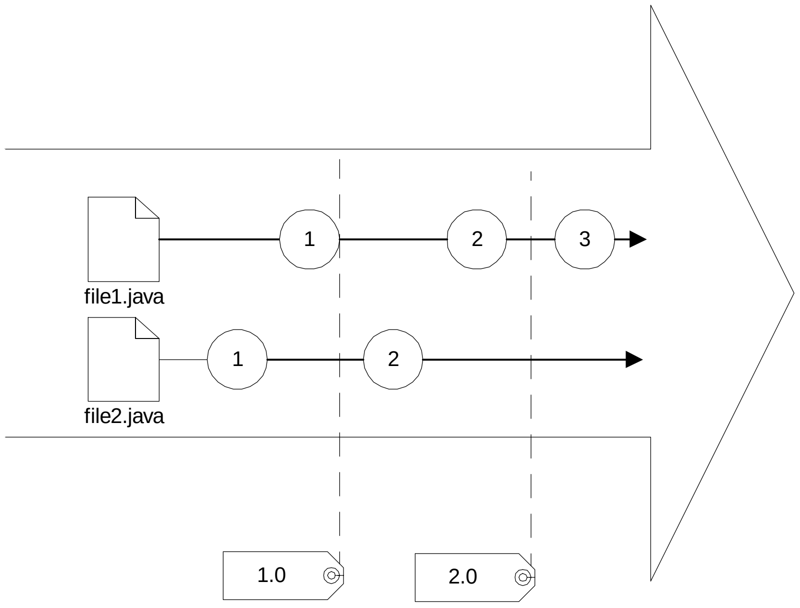


Figure 0-1 .A Code Line and its Components

You can have more than one codeline contribute to a product if each codeline consists of a coherent set of work. Each codeline can have a different purpose, and you can populate your workspace from an identifiable configuration of snapshots from various codelines. For example, you can have third-party code in one codeline, active development in another, and internal tools that are treated as internal products in a third. Figure 0-2 illustrates this. Each codeline will also have a *policy* associated with it. These policies define the purpose of the codeline, and rules for when and how you can make changes.

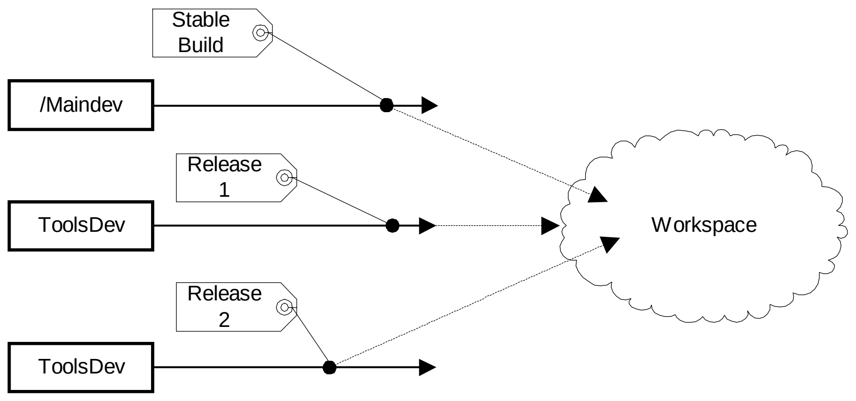


Figure 0-2 .Populating aWorkspace from different Code Lines

As codelines evolve, you may discover that some work is derivative from the intention of the codeline. In this case, you may want to branch the file so that it can evolve independently of the original development. A *branch* of a file is revision of the file that starts with the trunk version as a starting point, and evolves independently. Figure 0-3 illustrates this. After the second revision someone creates a branch and changes the file through revisions 2.1, 2.2, etc. A common notation is to indicate a branch by adding a minor version number (after a ".") to indicate that the branched revision is based on the major revision on the trunk. An example of a reason to create a branch would be that you want to start work on a new release of a product, yet still be able to fix problems with the released version. In this case you can create a branch to represent the released version, and do your ongoing work on the trunk. In this case, some of the changed you make on the branch may need to also make their way to the trunk, and you do by doing a *merge* to integrate the changes from the branch to the trunk. Figure 0-3 shows this with the dotted line from revision 2.2 to revision 3.

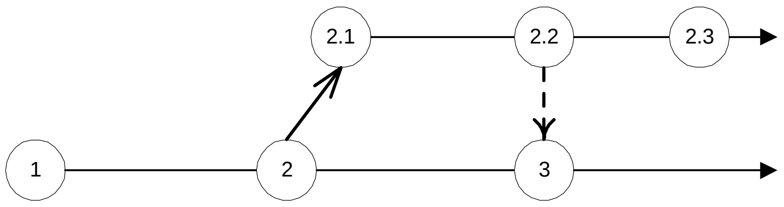


Figure 0-3 .Branching a Single File and Merging with the Trunk

Merging can be automated to some degree by tools that identify contextual text differences, but you often need to understand the intention of the change to merge correctly.

Often you will want to branch not just a single file, but an associated set of files in an entire codeline. In this case, the versions refer to versions of the entire codeline taken as a unit, where a version of the codeline includes all of the revisions of the codeline at the point in time.

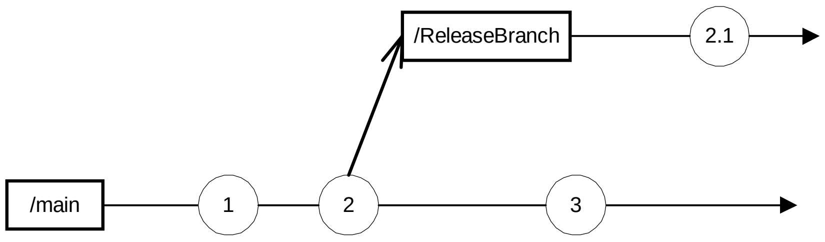


Figure 0-4 .Branching an Entire Codeline.

Every time you change any component of the codeline, you create a (conceptual) version of the codeline. In reality, most users of the code base don’t need to identify each change by a version number that indicates the total number of changed files in the codeline. Certain versions are significant, including points at which there is a product release, a branch, or a validated build. These versions on the codeline can be identified by *labels*.

Codeline and Branching Diagrams

The discussion up to this point in this chapter has illustrated the concepts of codelines, branches, etc. using the notation that this book uses for most of the examples. This section summarizes the notation, and defines the symbols a bit more strictly. The codeline diagram notation is based on the notation for UML activity diagrams, with the addition of symbols to indicate versions and revisions, and with the variation that the flow goes from left to right as time increases. The notation is based on the one used in the paper *Streamed Lines* (Appleton et al. 1998), and was further inspired by the diagrams in Michael Bays' book *Software Release Methodology* (Bays 1999). As with any notation, the purpose of this notation is to convey meaning clearly, so some of the diagrams in the book may use additional symbols, or vary slightly from the description here where it helps to explain the subject matter.

Figure 0-5 shows the notation that we use in the codeline diagrams in this book

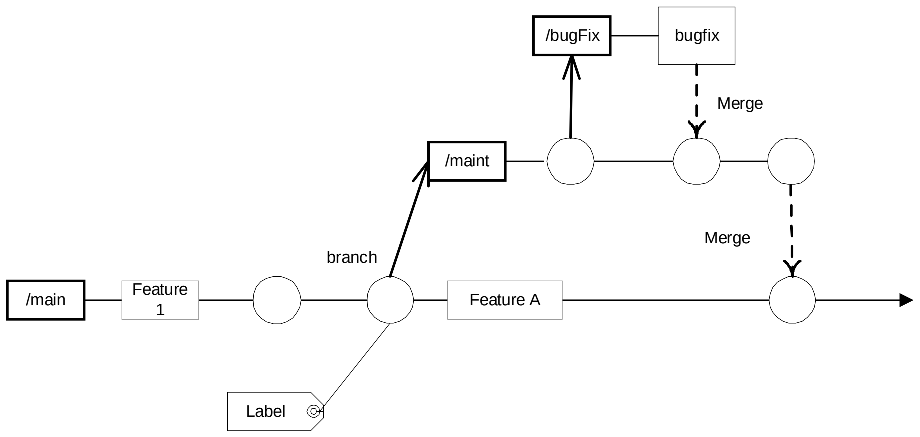
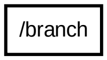
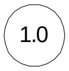
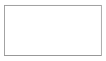
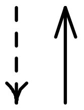
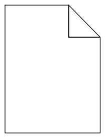
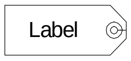


Figure 0-5 .Codeline Diagram Notation

Symbol	Description and Notes
	Ar rectangle with a bold border is the start of a codeline. It often has an indentifying name
	A circle is a version of the codeline, or a revision of a file. A branch or merge point is also considered a version. It sometimes has an identifier for the branch, such as a version number. This can be blank.
	A grey bordered rectangle within a codeline indicates a change task, which can be identified by a description inside the box.
	An arrow with a dotted line indicates a merge from the codeline at the start of the line to the codeline with the arrowhead. A solid arrow indicates a branch.
	A document symbol, when it is attached to a codeline start indicates the policy. You may also see this symbol used informally to represent a document.
	This symbol indicates a label, or an identified revision. There will be a line going from the tag to the part of the codeline that is indicated.

Further Reading

- Tichy's paper on RCS: "A System for Version Control" (Tichy 1985) is a classic paper on an early popular version control system.

- The paper “High Level Best Practices in Software Configuration Management” by Wingerd and Seiwald (Wingerd and Seiwald 1998) provides a good overview.
- Michael Bays’ book *Software Release Methodology* (Bays 1999) has excellent descriptions of the concepts of codelines and version control.
- Babich’s book, *Software Configuration Management: Coordination for Team Productivity,* is a classic (Babich 1986).
- *Open Source Development with CVS* (Fogel and Bar 2001) provides good advice on how to use a common open source version control tool, CVS, on open source projects, among other uses.
- *Antipatterns and Patterns in Software Configuration Management* (Brown et al. 1999) has a good collection of advice on what to do, and what not to do.
- The Appendix provides further sources on SCM and tools.

PART 1

Background

Chapter 1
Putting a System Together

For software configuration management (SCM) to help you work effectively as a team you must understand how all the parts of the development environment interact with each other and how SCM techniques fit in to the larger picture of a software development effort. You use version control on a regular, if not daily, basis and using it properly can make your development effort go faster and give you the flexibility you need to work effectively. If you use it incorrectly it will slow you down. This chapter describes the role that good software configuration management practice can play in a productive development environment.

Balancing Stability and Progress

Any complex piece of software is the product of a team of people who need to work together. They must coordinate ideas and code so that each team member can make progress without interfering with the work of other people on the team. For example, you might make a code change that implements a feature that you are responsible for, but if you don't communicate or coordinate with the rest of the team you may break some other team members' code unexpectedly.

Some development organizations take one of these extreme positions:

- Speed is essential, so we worry about quality and versioning later. Besides we're small enough that everyone knows what everyone else is doing.
- Quality is essential. We will work slowly, following processes to the letter, regardless of how it frustrates people on the project, or reduces productivity. We work on one release at a time.

Of course, neither of these sounds entirely correct, but we do them anyway, because each sounds like a really good approach, and if we have been burned by another approach in the past, following one of these philosophies may lead us to believe that we are avoiding problems that we've seen in the past. Consider the number of times that you have experienced one of the following situations in a software organization:

- "We're in code freeze. No one may check in any code until the product ships." This can be a period of 2 days or even a week. While this increases stability for the to-be-shipped product in a simple way, it hurts work on subsequent versions, and may even hurt the viability of a product, since releases take longer to start and finish.
- "Just copy the files somewhere. I'll use your version." This is faster, but it increases the risk of inconsistencies between developers, and can cause confusing problems later.
- "It works for me! Do you have the correct version of the code?" While one developer may have an environment that works, this is a symptom of undisciplined and inconsistent use of version control.
- "We use this tool in development, but builds are done out of another version control tool. Be sure to keep them in synch!" This "solves" the hard problem of about having a consistent tool set, but using manual steps to keep both code streams synchronized can cause errors and unpredictable builds.

Glacial Development

One of the most common issues that I see between release engineering and the development organization has to do with finishing up a release.Typically this is done with a “code freeze,” which is a period of time when access to the current codeline in the version control system is restricted so that only critical changes are made to the code. Since there is a risk that anyone can add a destabilizing change at any time, this makes a certain amount of sense from the point of view of those who want to ensure that they know what they are releasing as a product. The conflict comes when the developers have work to do for later releases at the same time.

If the codeline is frozen for too long (days or weeks) development either stops, or developers resort to “unsafe” practices, like trading files to share code (instead of using the version control system), or making lots of changes between check ins, making it harder to back out a change later. These things hurt quality and stability too, but it’s not as visible of a problem. One can avoid many of these problems with a good understanding of the issues and how to use the tools.

So, if long code freezes are a problem, why do they persist? They persist because people are not using their tools well.You can avoid code freeze by judicious branching. This assumes, of course, that the code is ready for freeze. A too aggressive pre-checking validation requirement can lead people to start copying files around instead of using the version control system.These are situations where the version control discipline has broken down, and version control doesn’t fit in with the way that people work and people waste time. This frustrates the individual developer, and will eventually frustrate management when release dates slip (because of slow progress) or morale declines (because of extra time that people put in to compensate for the slow progress). This is often because version control processes and tools are put in place before understanding how people work and how the product is structured. There are some procedures that make sense for every environment. But for most of the other cases, the techniques need to meet the needs of the team.

Because developers are creative people, they will often find ways to work around processes that do not work. Here is another statement that some may find controversial: Version control is a supporting discipline. Developers need to work with version control and release management tools and techniques because it is essential to track what they are developing. But version control should support the developer's work style. Since developers are smart they will use techniques that help them get their work done.

The Role of SCM in Agile Software Development

Agile software development approaches acknowledge the reality of change in software development, and suggest that you adapt your development methods to acknowledge that some projects have high levels of uncertainty and risk. In agile approaches "control focuses on boundaries and simple rules rather than prescriptive, detailed procedures and processes." (Highsmith 2002) Often people think of configuration management and version control as process-heavy things that might get in the way of the "real work" of coding. For many projects, SCM does get in the way, and some organizations overcompensate and don't use the tools to help them because of a fear that a process is inherently limiting. Other organizations want control and have so much process around version control that they hurt themselves. The right amount of version control is appropriate in agile projects. The approach to configuration management and version control that this book describes is most suited for agile teams, where the development pace is rapid that you don't want processes to get in the way, but you don't want to be stepping over each other either.

Often conflicts about software configuration management are because of the difficulty of determining how much structure you need. Too little and chaos reigns, too much and the environment becomes stagnant. Highsmith describes how this debate is really about balancing adaptation with anticipation (Highsmith 2002). Highsmith also observes that "one of the reasons for the divide between process and practice is often the perception that onerous process reduces the incentive to use *any* process." This reflects the reality that what matters is not your process as much as what people actually do.

A common area where the disconnect is obvious is in how a company handles branching. It is often the case that a company's branching model does not match its business model. A company that wants frequent product releases may have complex branching structures, and a need for time intensive merges. Another company may have many customers using independent releases, but have few branches, trying to manages the differences between the customer versions in some other way. This often happens because they misunderstand their tools, techniques, or because they are striving for some sort of ideal model that is inappropriate for their situation.

Much of the heavyweight application of SCM techniques comes out of a desire for perfection. Seeking an ideal isn't always the best approach to a successful project. As Gerald Weinberg says, "Reasonableness saves enormous amounts of time." (Weinberg 2002)

SCM in Context

There are many good software development practices that current developers just don't follow, even though they have been around for a while, as Steve McConnell comments (McConnell 2002). Incremental Integration and Source Code Control have been around since 1980. These are two approaches that you can add to your process and get a major gain in productivity with a minimum of effort. This book will show you how.

Software Configuration Management (SCM) processes and tools support at least two classes of tasks in the development process: management and day-to-day software development. (Conradi and Westfechtel 1998) The management related tasks that Software Configuration Management supports include identification of product components and their versions, change control procedures, status accounting, and audit and review. For day-to-day activities SCM helps you, as a developer, with version control functions that allow them to accurately record the composition of versioned software products as they are revised, maintain consistency between interdependent components, and to building compiled code, executables, and other derived objects from their sources.

This separation between management and development activities doesn't really make a lot of sense. The things that developers do are necessary for the management support tasks to be meaningful; you can't identify product components if there is no product to identify. There are times when the SCM process — particularly the management-support aspects — seems to impede development work as opposed to enhancing it. One reason for this is that the SCM processes are often defined with the goals and needs of management first, and ignore the daily needs of the developers. Another is that the processes don't use the appropriate techniques, out of ignorance, or a (misguided) attempt to avoid potential risks.

There are many reasons why organizations make the same mistakes in applying SCM practices, and, as a result, frustrate developers, and reduce productivity and quality. One reason is that some organizations lose sight of what the real goal of their work is. The goal of a software development organization is to develop software that solves a customer's problem and deliver quality software. The definition of quality is important here, so we use the definition: "value to some person." (Weinberg 1991b)

Some organizations put a lot of energy into doing things that don't help with the process of making useful software. Some of these places talk a lot about "the customer" and "quality" and "productivity" but their actions don't always support those spoken goals. Gerald Weinberg describes this sort of situation as a lack of *congruent behavior*. (Weinberg 1986) Non-congruent behavior confuses people and hurts quality.

Some organizations make decisions that have more to do with avoiding change and expense than with writing quality (and income-producing) software. Often these decisions are well intentioned; they end up being counter-productive because they are made without understanding all of the aspects of how developers work, and without consideration of how their decisions affect other parts of the development process. A team may not create a branch when a branch would appear to be helpful because the manager of the team had earlier experienced branching as being problematic.These survival rules are useful for reducing risk, but when over applied can increase risk. Gerald Weinberg says:

> "Survival rules are not stupid; they are simply over generalizations of rule we once needed for survival. We don't want to simply throw them away. Survival rules can be transformed into less powerful forms, so that we can still use their wisdom without becoming incongruent." (Weinberg 1993)

While members of a team will spend large amounts of time and energy thinking through how to design a software component, teams rarely put the same effort into thinking about how they work. In particular many source control practices often don't fit the needs of all of their users. Often the SCM practices are established at one point in time of the company, and are then continued somewhat blindly. One reason that SCM practices don't adapt to a team's needs is that some of these practices are organization-wide, and it is hard to justify, not to mention implement, global change.

A reason that practices may not meet the needs of the users in the first place is that the easy target for changing version control process is the tool, and discussions of how to work with version control often get lost in deciding what tool to get. Sometimes organizations decide that what they really need is a high-end (and powerful) tool like ClearCase, and when the know that they can't afford that, they decide that they shouldn't get something else, and they continue to use a tool that does not meet their needs, for example, Microsoft Visual Source Safe.

In the quest for a good solution we often lose sight of the fact that there are many small scale changes that can have a large impact on how we work. While a new tool might make a big difference, using your current tool effectively, and working within its limitations also helps greatly. Another reason that source control is ignored is that you can resolve most source control process issues through manual processes. While this *is* work and has a cost, it is one that most managers don't see. Similarly to planning for a product, it is important to consider the developers as a class of users for the SCM Process. "People who are left out of planning invariably turn up late in a change project to haunt you." (Weinberg 1991b) In this case, the "haunting" will be visible in terms of frustration, and decreased productivity.

This book will help you understand some of the techniques and processes of SCM, and how to apply them in a way that allows you to work more effectively. Many of the techniques can be applied incrementally and locally, so that you don't need to change the entire organization to improve the way that you work. It may also provide you with a means to explain how some day-to-day practices can support both your needs as a developer, and the larger needs of a someone managing a product release.

Keeping it Simple

Sometimes processes really do help get things done in a stable and repeatable way. Everyone involved in the work needs to have the same goal and vision for a process to work. When the work styles of the release management team and the development team don't mesh, and the processes involve many manual steps, there can be glitches that slow things down.

Consider the story of the company where the developers drive the process of what is part of the product. When they want to start using a new third party component, they release it into the repository, and everyone starts using it on the next update. The release management team however, doesn't just build using what is in the repository, but only from a well defined list, and they don't pay attention to development announcements about updates. Invariably a number of nightly builds will fail until the release team does them with the new version. There are a number of responses to this problem. The release manager can scold the developers, the release team can use the version control tool's reporting facilities to check for updates, or the release team can monitor the development team's announcements for updates.

Of these approaches, scolding is the least effective, and often, the most frequently chosen approach. Having a common approach to the version control process, or using tools to remedy differences can make things run much more smoothly.

SCM as aTeam Support Discipline

Encompassing the management and development support aspects of SCM, configuration management, and in particular version control, plays a role in supporting the work of teams. Version control can certainly benefit a one or two person work group, but when you have more people than can comfortably manage to communicate everything that they are doing on a project to each other, you need an infrastructure to support communication. A version control system is the way that teams can get the answers to questions about who made recent changes, when something broke, what code customers are using, and what components are related. This is what we mean when we say that software configuration management serves as a mechanism for communication, change management, and reproducibillity.

To develop software you need to do obvious things like define requirements and develop designs, writing code for the product and for tests and writing documentation. The hardest, and one of the most important things that has to happen is effective communication. Communication is not just sharing status and general information, but also sharing enough detailed information about what people are doing so that teams can work together and be more productive. While engineers often spend a lot of effort on design and implementation decisions, engineers and their managers often leave teamwork issues to “just happen.”

You can realize important improvement in team productivity and software quality by using the appropriate version control practices. Unfortunately, many version control practices are often established without a good consideration of the things that influence how code is written. Some of these things include:

- The *structure of the organization*. A three person team in one room has different needs than a large team spread across the globe.
- The *product architecture*. Some points in the architecture allow for more decoupling than other points.
- The *tools* available. Some tools support some techniques more effectively than other tools. If you have a tools that does not handle branching well, you may want to come up with a different release model, or get a better tool.

For each of the these influences you could change factors themselves. Put your whole team in one city (or in one room); fix the product architecture to reduce coupling; buy a “better” SCM tool. These changes can be expensive. It can also be expensive (though in not as obvious of a way) to *not* fix the other problems. We often find ourselves in a position where the costs for changing a tool, for example, are very visible, but the costs (in productivity, morale, etc.) in using the wrong tool badly, are hidden. It may be easier to adapt the way that you use Version Control to the way that people work than it is to change the environment. Chapter 2 discusses the role of organizations and teams in more detail.

Our goal in this book is to point out the solutions that are effective given the environment you must work in. You may be surprised in the improvements you can find in small changes in process. You will also identify aspects of your environment that you really should change. In the end we want to help you build better software faster, not to use a particular process.

Some of the techniques are things that affect how the team works, and may need consensus or management buy in. Some are practices that you and one or two colleagues can do on your own.

There are many aspects to team communication, including organizations or management. These things do need to be considered, since they have an impact on how people work. This books is about the tools and techniques that you can use as a person developing software to work cooperatively with members of your team, and members of other teams. The way that teams communicate their work products to each other is through their Software Configuration Management and Version Control practices.

What Software Configuration Management Is

Like many things in our discipline of software development, software configuration management means many things to many people. This section discusses some of the dimensions of SCM and highlights the aspects that we are concerned with.

Software Configuration Management (SCM) serves at least two distinct purposes: management support and development support.

A standard definition of software configuration management includes the following aspects (Dart 1992):

- *Configuration identification*, which includes determining which body of source code you are working with. This makes it possible to know, among other things, that you are fixing a bug in the source code which is in the correct release.
- *Configuration control*, controlling the release of a product and changes to it throughout the lifecycle to ensure consistently creation of a baseline software product. This can include not only changes to source files, but also which compiler and other tools were used so issues such as differences between compiler support for language features can be taken into account.
- *Status accounting* audit, recording and reporting the status of components and change requests, and gathering vital statistics about components in the product. One question we may want to answer is: "How many files were affected by fixing this one bug?"
- *Review*, validating the completeness of a product and maintaining consistency among the components by ensuring that components are in an appropriate state throughout the entire project life cycle and that the product is a well-defined collection of components.
- *Build management*, i.e., managing what processes and tools developers use to create a release, so it can be repeated.
- *Process management*, ensuring that the organization's development processes are followed by those developing and releasing the software.
- *Teamwork*, controlling the interactions of all the developers working together on a product so that people's changes get inserted into the system in a timely fashion.

Ideally a configuration management process should serve both broad organizational interests as well as to make the work of a developer easier. A good SCM process makes it possible for developers to work together on a project effectively, both as individuals and as members of a team. While there are various tools that can make the process simpler, tools alone are not enough. Successful development organizations will also use certain patterns for software configuration management

With respect to team interactions, a successful configuration management process allows:

- Developers to work together on a project, sharing common code. For example a developer of a derived class needs to stay in synch with whoever is developing a base class, and a client of a class needs to be able to work with the current version of that class.

- Developers to share development effort on a module, such a class or simply a single source file. This can be by design or to allow someone to fix a bug in another person's module if the other person in unavailable.
- Developers to have access to the current stable (tested) version of a system, so you can check if your code will work when someone else tries to integrate it into the current code set.
- The ability to back up to a previous stable version of a system. This is important to allow a developer to test their code against the prior consistent versions of the system to track down problems.
- The ability of a developer to checkpoint changes to a module and to back off to a previous version of that module. This facility makes it safer to experiment with a major change to a module that is basically working.

Attaining all of these goals involves compromises. A cynic could paraphrase Otto von Bismark's remark "To retain respect for sausages and laws, one must not watch them in the making" to apply to software systems and processes. We need to watch how our processes evolve, attend to what works, and what does not work, and by leveraging the experiences of others we can improve.

Version Control is an important part of making team software development work effectively. Version control practices help people work on the same components in parallel without interfering with each other's work. Software Configuration Management and Version Control practices allow you to do things like:

- Develop the next version of a piece of software while fixing problems with the current one.
- Share code with other team members in a controlled way, allowing you to develop code in parallel with others and join up with the current state of the codeline.
- Identify what versions of code went into a particular component.
- Analyze where a change happened in the history of a component's development.

The next section identifies some of the tensions in a development project that interfere with establishing good SCM practice.

The Role of Tools

The first question that people ask when they talk about version control is “what tool are you using?” This is a very practical question, the brings out the important impact that tools have on the way that we work.

While the tool influences how you work, it should not be the main concern. Of course, tools with a feature set that is matched to your needs make things work better. But the most important thing is to balance the capabilities of the tool with the needs of the organization and the developers. It is critical to make the processes easy so that people will follow them. Another aspect that this book shares with advocates of agile development is that it is the people on a team and what they do that is important, or as the agile manifesto says: “Individuals and interactions are more important than processes and tools.”

When you find that an everyday practice needs a large number of (hard to remember) manual steps, you may want to question the capabilities of the tool, or the value of the practice.

The appendix describes some common tools, and how to use them to implement the patterns using the concepts of the tools. The resources section provides even more information.

The Larger Whole

Tools, Product Architecture, and Organization are all important aspects of the software development environment that we need to take into account when building software systems, but when we let them drive the process at the expense of delivering quality software in a reliable manner we get into trouble. There is a lot to do besides coding; documentation and testing are part of the process, and quality and reproducibility are things that help get a product out. The next chapter discusses in more detail what the other pieces of the picture are, and how SCM fits into this picture

This Book’s Approach

This book approaches the problem of using software configuration management and version control by looking at the overall environment within which you use version control, and by demonstrating how to solve problems after considering your specific environment.

This book places a number of best practices that are well documented in the context of a team's work style and the constraints of your organization. This book does not present a set of rules that you should follow, but rather a set of practices that work together (with variations).

The practices are cast in terms of patterns. We discuss patterns in general inChapter 3 You don't need to understand patterns to use the ideas in this book, though you may get an added benefit if you do.

Unresolved Issues

How do we improve the way that we use version control? The next chapter describes how a pattern oriented approach is helpful.

Further Reading

This book is not about software architecture or process, per se, so we cannot go into as much detail about these issues that we would like. The following references provide more detail.

- Garlan and Shaw discuss give a good overview of architectural styles(Shaw and Garlan 1996).
- The *Pattern Oriented Software Architecture* series describes a number of architectural patterns (Schmidt et al. 2000; Buschmann et al. 1996).
- The *UML User Guide* discusses the various architectural views suggested by the Unified Process (Booch et al. 1999).
- Jim Highsmith provides a good overview of the current set of agile techniques in his book *Agile Software Development Ecosystems* (Highsmith 2002).

An important aspect of improving processes is detecting the problems in a organization and influencing people to change.

- *Seeing Systems. Unlocking the Mysteries of Organizational Life* by Barry Oshry (Oshry 1996) is an useful and entertaining book about how to detect cycles of behavior that need to change, and how to change them.

- *Getting to Yes* (Fisher et al. 1991) is a classic book on negotiation, and informal negotiation is something that you may find yourself doing when you try to make changes in the way people work. *Getting Past No* (Ury 1993) is also worth a read.
- *Becoming a Technical Leader* by Gerry Weinberg (Weinberg 1986) has great advice on leading from any role.

Chapter 2
The Software Environment

To use software configuration management (SCM) properly, you need to understand how its techniques fit in to the larger picture. This chapter discusses some of the other aspects of the development environment and the processes that we use to create and maintain them so that we can understand how SCM fits in.

General Principles

By Software Configuration Management, we mean the set of processes that one uses to create and maintain a consistent set of software components to release. Version Control is the part of the SCM process that a developer sees most often. It manifests itself in terms of your version control tool and how you use it. Often you will associate the use of a tool with a set of policies or rules for how to use it to follow the lifecycle that the organization needs (checkins, builds, releases, etc.). Sometimes the policies are enforced automatically, other times developers must follow a manual procedure.

We can't give you one set of rules to say how to use version control, or which tool to use. How to use it (or any other tool or technique) depends strongly on your situation. Just as you might use different tools for building a dog house as opposed to a summer vacation cottage, different software project teams need different approaches. If you use big complicated tools for a small project where everyone communicates well, you may slow things down, and increase costs. Likewise, trying to blindly apply techniques that worked well in an initial product release project in a 5 person startup may cause trouble in the same startup 6 months later with 20 people working on release 3 (and maintaining release 2, and release 1).

Working in a team adds a need for communication and also changes the way that you would execute these principles. For a 1 person project you can try to build your software system and accompanying development environment, based solely on first principles of "good" coding practice. Some basic version control system will help you with your memory, but you should not (usually!) have any communication issues. Once more than one person is involved in building the system communication and interaction between developers comes into play. Each software system is built in a context that shapes the code and the patterns of work. The same principles that you use as an individual developer that also apply when you work as part of a team. Each team, and every project is different, so no one process will work. Version control is a core part of the communication mechanism.You need to vary the details of how you use version control based on your situation. To do that you need to take a step back and look at the places where software is built.

There are a few general principles that we can apply to using configuration management on any software project. The details of how to apply the principles will vary, depending on the size and nature of the team. Some of these principles are:

- *Use version control*. Version control is the backplane on which a software organization communicates work products among themselves. This sounds obvious, but some organizations do not really track versions. Even if they have a version control tool, they do not use it.They copy files between developers without checking them in. They have no way of identifying what went into a release and how to reproduce the system at a point in time. Some places need more lightweight tools and processes than others, and some development techniques allow for more flexibility in the process, but every team needs to have a way to do version control, and needs to use it to communicate code changes among the members.

- Do *periodic builds*, and integrate frequently. Have some sort of periodic build process that builds what looks like the current version of the software so that people can see how well things fit together. The longer you put off integration, the harder integration problems will be. How often you need to do this depends on a a number of details about your organization. Extreme Programming (Beck 2000), and other agile approaches say to integrate continuously, but for simple systems that may cause too much overhead. Erring on the side of integrating too often is generally better, but integration takes time, and you want to find the point where integrating more often adds speeds up the overall process.
- Allow for *autonomous work*. Every team, even those with one person, may need to work on various points in time of the codeline. Each team member should be able to control what versions of what components they are working on. Sharing "common" components may work well most days, but when you need to diverge from working on the latest code, it will probably be for an emergency.
- Use *tools*. If you have too many manual processes people may make mistakes or simply skip a step out of frustration or perceived need. Be lazy and write tools.

How you apply these general rules depends on the environment that you are building your software in. And, of course, there are other software engineering practices that you also need to apply around testing, coding, and related issues. See the References section for pointers. The next sections describe some of the elements of the software environment.

What Software is About

In its most concrete form, a software system is the sum of all of its code. You write code and you end up with an application that does something useful. There are also other artifacts that make the system run; data, documentation, and anything else you need for your system. The code, however, defines the shape of the other artifacts. Without the code, or these artifacts, there is nothing to deliver to a customer or user.

To understand how you build a software system you need to think about more than just the code and other physical artifacts. You need to understand how the people on the team work, how the product is structured, and what the goals and structure of the organization are. We often ignore social issues such as organizational structure and politics, thinking them secondary to what we perceive to be the main goal: building a software system. Why this sounds appealing (focusing in on the core goals and all that), these other factors are important to consider. If you don't consider them, they will get in the way when you try to build and deliver systems quickly.

It's easy to get lost in all of the other factors. Corporate politics alone can cause you to spin endlessly. One way to think about the software environment where you are using configuration management is to model it as consisting of the following structures:

- Where the developer codes, the *Workspace*. This is the day to day code environment of the developers.
- Where the coding takes place, the *Organization*. The developer works in the context of an organizational structure. There are teams developing software, testing, marketing, doing customer support, etc.
- Where the code fits in: The *Product Architecture*. Along with the product architecture is the Release structure, which specifies how many releases are being developed together
- Where the code (and code history) is kept: The *Configuration Management Environment*. This includes tools, processes, and policies.

These structures all have policies and processes associated with them that sustain them. While it’s not always obvious, these structures influence each other, and if you don’t take the influences into account, your work may be harder than it needs to be.

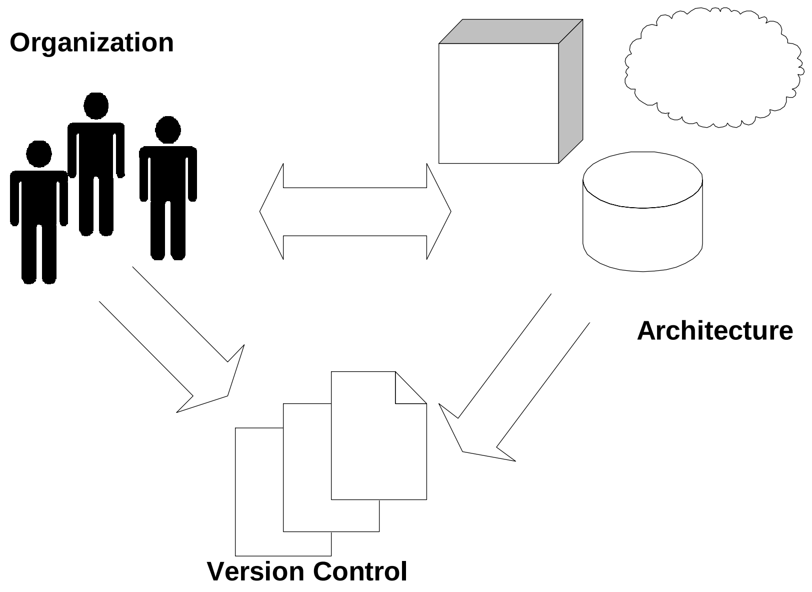


Figure 2-1 .The Interactions Between Elements of the Environment

Some examples of these influences are:

- The organization influences the structure of the code and the architecture (Conway’s Law)
- The organization and the architecture both influence version control policies. A less tightly coupled architecture can be developed in parallel more easily than a tightly coupled one. A small team in one room can communicate more effectively than a large team spread across the globe. In each case version control policies can compensate.
- The version control environment and the organization influence the structure and use of workspaces. You can use some policies to alter the way people work; this isn’t always the best way to do things since it is working backwards in many cases, but it can have an influence.

Often tools and mechanisms for work style and version control fail because they are applied in the absence of an understanding of the reality that the mechanisms really need to work well with the existing environment.

Policies influencing work

One company that I worked at had a very rigorous pre-check in testing regime that developers had to follow. You had to run a very long suite of tests -- approximately 30 minutes -- before checking in any code.

In the lifecycle of this company, 30 minutes was a long time, so people worked to avoid the long tests by either checking in less often, which meant that a check in would add more than one feature or fix more than one bug, or skipping the tests, which hurt the reliability of the code base. There are more extreme examples of policies affecting work. The developers eventually lobbied for a pre-check in smoke test that took a couple of minutes to run, and the processes got back in sync with good work practices.

The next sections discuss each of the development structures, and their interactions, in more detail.

The Development Workspace

Software development happens in the workspaces of the developers in the team. A workspace simply is the set of components that a developer is currently working with, including source code that the developer is editing and is using for reference, build components necessary to run the software, and third party components. The *PRIVATE WORKSPACE (6)* pattern discusses the components of a workspace in more detail.

Each developer will have one or more workspaces depending on the number of projects that he is working on. The build team may also have an integration workspace to do periodic builds in.

Architecture

"Software architecture" is a widely used, and widely misunderstood phrase. The phase can mean many things depending on the person using the term and the audience. Sometimes different understandings of what "architecture" means can lead to legal disputes. Steve McConnell recounts a story about being an expert witness in a case where a company sued a VP over non performance of work duties, in part because they had a different understanding of what an ‘architecture’ was (McConnell 2000).

The one thing that most definitions have in common is that the architecture defines some aspects of the structure of the system. IEEE Std. 610.12 defines architecture as “The organizational structure of a system of component.” The architecture places constraints on where and how parts fit together and how easy it is to change or add functionality. Even if there is no explicit architecture document, or architectural vision, there is an implicit “architecture” that the existing body of code defines. The code establishes a structure that you need to work within, perhaps changing it as you go. Even in a “green fields” development project, there are still “architecture” decisions that will constrain software decisions, including, but not only, method of communication, language, choice of database.

For the purpose of discussing team software development we define architecture as “A description of how the parts of the software fit together that provides guidance about how the system should be modified.” This includes both the logical and physical structure of the system, including relationships between components, deployment units, and even the way that the code is structured in the source tree.

The role of architecture in developing a version control strategy is to establish the structure of the version control modules, and therefore how we make concurrent work easier. Printer writes that “software architectures represent another mechanism of interaction supporting collaborative work at a higher level of systems abstraction” (Grinter 1995).

The architecture influences this because the architecture defines:

- What comprises a deliverable unit
- The communication paths among the units
- and, indirectly, The directory structure and other structural aspects of the source code respository.

For example, one way to allow concurrent work is to design a system using a Pipes and Filters architecture (for example, (Buschmann et al. 1996)). At a finer level of detail, you can decouple systems using patterns such as Parser-Builder (Stephen P Berczuk 1996a), or Visitor ((Gamma et al. 1995)). These patterns each allow teams to treat the code that they own as fairly separate from other teams’ code. The Pipes and Filters approach can be a way to design a system that is being built by teams a large distance from each other. It is entirely possible to structure your source tree in a way that conflicts with this goal, and if you ignore integration process issues, the architecture alone will not save you.

An architecture comprises a number of views, each of which exposes the part of the picture that you care about in a specific context. The UML (Booch et al. 1999) takes this approach by defining the following 4 views as well as a *Use Case* view that describes what the software does (Krutchen 1995):

- *Implementation*, which defines units of work, at various degrees of granularity
- *Deployment*, which specifies where physical components are put. The deployment structure has a strong impact on how software in built.
- *Design*, which is the lower level details. Except for the way that the class design effects the module structure and the dependencies between components, this has the least effect on how the version control should be structured.
- *Process*, which affects performance, scalability and throughput. This has the least relevance to our discussion, though it is, of course, important.

Architecture is heavily influenced by organizational structure. There are occasions where an architecture can influence the formation of an organization, but most often, the product architecture is developed in an existing organization.

Architecture can be organized with organization in mind. The architecture may match the structure of the organization, based on location, the skill levels of developers of components.

Modularity leads to decoupling, which adds concurrency to the development process:

> Modularity is about separation: When we worry about a small set of related things, we locate them in the same place. This is how thousands of programmers can work on the same source code and make progress. We get in trouble when we use that small set of related things in lots of places without preparing or repairing them.(Gabriel and Goldman 2000)

The architecture and organization have an effect on the best way to partition source code into directories:

- The Architecture / Module Structure. The products module structure is the strongest influence on the source code structure. A typical partitioning is to have the source code for one module in one directory, so that you can manage the files more easily. if you are using a language that has header files, like C++, or interface definition files, like COM, or CORBA IDL files, you may want to put these files in an common location, apart from the bulk of

the source code; this makes it easier to ship interface definitions to customers if you provide an API.
- The Team structure. The number of people working on a project, the number of people working on a module. Modules may be grouped logically based on the people working on them. This may not always be optimal, but it happens.
- The particulars of your development environment. Do you have symbolic links? What is your programming language?

The Organization

This section discusses some of the aspects of an organization that affect the way that software systems come into being. This is a large area, which has research papers (Allen 1997b, 1997a; Allen et al. 1998) and entire books (Weinberg 1991a; Fred Brooks 1975) devoted to it, so we will only mention some of the more relevant factors here.

Software development is, in many respects, a social discipline.The organization can have a significant influence on how a product is designed and how teams are structured. Things can get difficult when your development approach conflicts with the structures that the organization imposes. Changing the organizational structure is often the best, if most difficult, fix. Even if changing the organization is impossible, the real danger is in not acknowledging these influences and focusing only on the immediate programming tasks.

Organization has an influence from the perspective of workspace structure and version control because the structure of the organization constrains communication. The organization affects communication by the *distance* between people and teams. Distance is a measure of how hard it is for teams and team members to interact. It can be about physical distance, but not always. Organizational structure that divides responsibilities, which can also make it communication harder if the responsibilities are not divided in a manner consistent with the needs of the application.

The organization can be more subtle in it’s influence on the architecture and the way that you work. The nature of the organization and its culture constrain the team dynamic, the architecture, the goals of the development process, and how problems are handled.

Distance is about:

- Physical location.Teams that are not physically close can have a lower bandwidth of communication.
- Culture and team dynamics. An appropriate culture can result in teams that are physically far apart having very good communication, and can also result in people in the same room having very poor communication.
- Organization structures that dictate communication paths. This is an aspect of culture, since communication need not follow corporate structures, but if this is an ethic in your organization, then you might do well to have the architectural communication paths follow the organizational boundaries

You need to be aware of the effects of organizational distance, and we encourage you to work to be a change agent to improve the long term picture, but simply building your work structures so that they are robust in the face of these organizational issues will generate a big win.

Some other organizational influences include:

- Skill sets of people.
- Distance, which encompasses many things, including physical distance.
- Values and culture
- Location of personnel and other resources
- Team stability
- Type of company: consulting company vs. product company vs. product company with customer specific changes.

The structure of an organization can have a large impact on how the software is built, and hence coordination needs. Organization forces can have a very strong impact on the product architecture, as well as having controlling process aspects such as when and how releases get created, tested, etc. It is beyond the scope of this book to tell you know to match the product process, architecture to adapt and leverage the structure of the organization. We will briefly describe some of the issues so that you can better understand them.

While many say that an ideal development environment has developers who are near each other, and who communicate effectively, many real organizations have resources that are geographically distributed, and you need to help them to work well together. One answer is to distribute responsibilities so that remote groups can very well defined interfaces between them, and few dependencies.You will also need to structure the version control system so that people at all locations can see all the code easily.

Some of the influences of organization on architecture that are particularly relevant include:

- Module structure ((Bass et al. 1998)). Modules should be developed by people who can work well together, and aspects such as geography or technology experience may dictate that certain components should be developed by certain groups. There is an interplay with the product architecture here; you may have to make the choice of assigning a component to a group with the most appropriate technical experience, for example, or a group with lesser technical skills, but that is in a better position to interact (by whatever means are appropriate -- this is not an argument for geographic proximity) with the groups that interface with its work product.
- How often all of a systems components are integrated integrate is often a function of organizational policy.
- Workspace management: How you set up your local development environment and how your workspace related to others.
- Version control and identification: how you use source control tools and other means to coordinate changes with others, publish your changes, and reproduce environments, such as when you need to fix a bug in an earlier release. This includes issues such as branching and labeling, which are often faced with much consternation.
- Coordination: How you work together with other teams and developers.
- Identification: How you know what you built.

The Big Picture

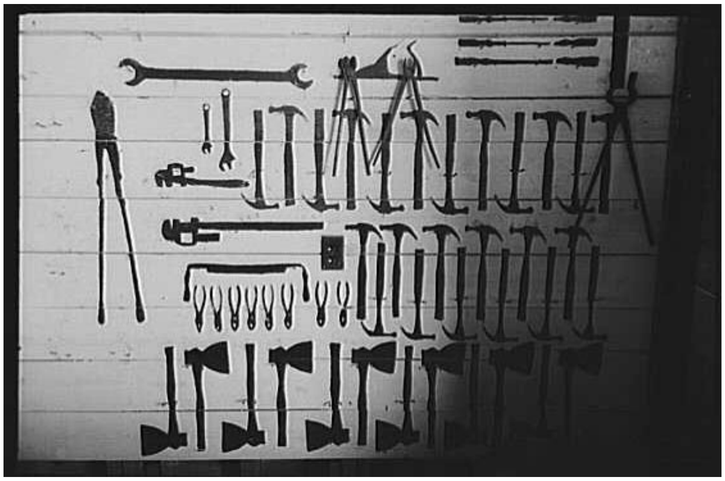
Given all of the things that have a part in building a software system, it is easy to lose track of what your primary goal is, which is to build a software system. It is easy to get overwhelmed by the technologies and techniques that you use as part of the process. Version control, configuration management, build, testing, pair-programming, branching and other process things, are all aspects of the software engineering environment. These techniques support the process. A goal is not to branch, for example. Branching is one way to accomplish concurrent development, or isolating a line of work. This may seem like an obvious statement, but I have seem a lot of energy expended on figuring how to accomplish tasks for which there were other, simpler, ways to reach the end goal. An excellent bit of advice to keep in mind as you read this book is “Don’t mistake a solution method for a problem definition, especially if it is your own solution method” (Gause and Weinberg 1990).

Putting together all the influences of architecture, organization, and configuration management on each other, we see that the big picture view of an SCM environment encompasses the tools and process for identifying, organizing, controlling, and tracking both the decomposition and recomposition of a software system's structure, functionality, evolution, and team work. An effective SCM environment is the glue between software artifacts, features, changes, and team members.

Further Reading

- Steve McConnell has a few books that cover best practices for coding and teamwork, in particular, *Rapid Development* (McConnell 1996), and *Code Complete* (McConnell 1993) have some excellent guidelines.
- *The Pragmatic Programmer* (Andrew Hunt and Thomas 2000) talks about the value of using tools to automate processes and procedures.
- *The Practice of Programming* by Kernighan and Pike (Kernighan and Pike 1999) says quite a bit about the value of automation and tools.
- Linda Rising and Mary Lynn Manns provide some guidance on introducing new ideas in their paper *Introducing Patterns into Organizations* (Manns and Rising 2002).
- Alistair Cockburn gives some good advice about writing use cases in (Cockburn 2000). *Patterns for Effective Use Cases* (Adolph et al. 2003) provides a patterns-based approach to writing use cases.
- David Kane and David Dikel's book *Software Architecture: Organizational Principles and Pattern*s (Dikel et al. 2001) is an excellent source for information about what Architecture is and how to use it.

Chapter 3
Patterns



For configuration management to help you work effectively as a team you must understand how all the parts of the development environment interact with each other. One way to model this is to think about the development process in terms of the relationships among the patterns in the development environment.

You do not need to master the concepts of patterns and pattern languages to find value in this book, but the pattern approach is an easy way to think about how the elements of a system work together as they compose a system. This chapter explains what patterns are, and how a pattern language can help you understand and improve your team process, and it gives an overview of the patterns in the book.

About Patterns and Pattern Languages

A solution only makes sense if you apply it at the right time. A pattern language is a way to place solutions in the context of the things that you've already done.

There are many books and papers that talk about patterns for software, architecture, organizations and teams, and technology development, and we won't try to cover all of that in this chapter. This section will give a brief overview of what patterns are about, and provide references if you want more detail.

A simple definition of a pattern is a "solution to a problem in a context." Each pattern in a pattern language completes the other patterns in the pattern language. In that way, the context of a pattern is the the patterns that came before it. This means that a pattern fits within other patterns to form a pattern language.

The idea of patterns and pattern languages is originally from work that the architect Christopher Alexander did in building architecture to describe qualities for good architectural designs. In the seventies he started using pattern languages to describe the events and forms that appeared in cities, towns, and buildings in the world at large.

Alexander talks about a pattern as something that "describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."(Alexander et al. 1977). Alexander defines a pattern as "a rule which describes what you have to do to generate the entity which it defines." (Alexander 1979) A pattern describes a solution to a problem in an environment "in such a way that you can use this solution a million times over, without ever doing it the same way twice." (Alexander et al. 1977) Patterns according to Alexander, are more than just solutions. They are *good* solutions.

> And there is an imperative aspect to the pattern. The pattern solves a problem. In is not merely "a" pattern, which one might or might not use on a hillside. It is a *desirable* pattern; for a person who wants to farm a hillside, and prevent it from erosion, he *must* create this pattern in order to maintain a stable and healthy world. In this sense, the pattern not only tells him how to create the pattern of terracing, if he wants to; it also tells him that it is essential for him to do so, in certain particular contexts, and that he must create this pattern there (Alexander 1979).

Alexander's patterns set out to be more than just cookbook solutions.

> But when properly expressed, a pattern defines an invariant field which captures all the possible solutions to the problem given, in the stated range of contexts.(Alexander 1979)

A pattern is a rule for resolving forces, but the important thing is that it fits in with other patterns:

> We see, in summary, that every pattern we define must be formulated in the form of a rule which establishes a relationship between a context, a system of forces which arise in that context, and a configuration which allows these forces to resolve themselves in that context (Alexander 1979).

Alexander's pattern language is “a system which allows its users to create an infinite variety of those... combinations of patterns which we call buildings, gardens, and towns.” (Alexander 1979) Alexander documents patterns that exist in our towns and buildings. For example, one of Alexander’s patterns is *HALF PRIVATE OFFICE*, which describes how to achieve the right balance between privacy and connection to office work.

Alexander’s pattern languages are very ambitious, and the pattern language that he has authored can give you much insight into architecture and urban planning. It was also the inspiration for the patterns in software.

Patterns in Software

While the initial work about patterns was about building things on a human scale, we can apply the basic ideas of patterns to software development. Using and writing patterns is about the quest for objective quality. Software development involves people working together to build things. These things are often similar to other things that were built in the past. In some software development organizations the process works well because the teams apply techniques that work, and cast aside techniques that do not. In other organizations they don’t, so there must be something that we can learn from the successes and when to apply the techniques that worked.

The first major work in software patterns was the book *Design Patterns* (Gamma et al. 1995). This book catalogs some key techniques in object oriented design with very good descriptions about when and how to implement them. *Design Patterns* does not capture the full power of patterns because each pattern stands by itself, and you still need a good understanding about software systems to put them together to build something larger. To help people apply design patterns books have been written to show how to implement these patterns in various situations (Vlissides 1998) and in other languages (Alpert et al. 1998).

Other books and writings on software patterns address architecture issues (Buschmann et al. 1996), (Schmidt et al. 2000), and organization issues (Coplien 1995), but the patterns are still not connected in languages. By not being connected, finding the correct pattern and applying it in the correct situation requires a deep understanding of the patterns and the situation. While you do need to understand something about the patterns and what you are doing to benefit from a collection of patterns, this need for finding your way through the patterns reduces a major benefit of patterns, namely as a way to navigate through a set of complex trade-offs and implement a good solution.

Pattern languages are useful for documenting software architecture principles, particularly object-oriented systems because of the emphasis on structure in OO systems. They can and have been used to describe architecture systems in other paradigms. Some of the existing pattern languages and collections about software include pattern languages that describe how to build social structures such as software development organizations(Coplien 1995) and (Olson and Stimmel 2002), patterns on architectural principles (Buschmann et al. 1996), (Schmidt et al. 2000). The "Further Reading" section at the end of this chapter lists some more examples.

This book describes a pattern language that works at the intersection of the team and the architecture: the developer's workspace and its interface to the version control system. We talk about patterns that describe how people build software on a human scale. In particular, we describe patterns that people use when applying Software Configuration Management techniques.

Configuration Management Patterns

Patterns are particularly useful way to think about software configuration management for the following reasons:

- Software Configuration Management (SCM) involves how people work in addition to the mechanics of how the code is built.
- SCM involves processes for doing things, and the artifacts that result. Patterns are particularly good at describing the process and thing aspects at the same time.
- While there are many "best practices" for SCM, to use them effectively, you must understand how they relate to other practices that you use, and on the environment in that you use them.
- Small local changes in SCM practices can lend a large improvement to the process; small changes, and organic growth can effect change; you don't need high level management buy-in, though it can help.

When you think about the way that the members of a team work together some of the first words that come to mind are process and behavior. Processes and behaviors are dynamic things, and dynamic things can be hard to understand. People are better at modeling static situations and later extending the static models to include behavior. We know how to describe what a building looks like, and given a set of diagrams we can build one that matches the specification quite nicely. We have a harder time modeling systems that have people in them. But static structures do not appear of their own accord, and they don't do anything, so you need processes to create and sustain them; this is where processes come into play. The software architecture, the way a developer's workspace is configured, and the way that the SCM system is structured are all sustained by the processes that you use each day.

For a team to build software in a consistent manner you need people to implement processes consistently. There a a few ways to get that consistency. Goodwill might work, but is better if behaviors are enforced by the tools and the environment. Sometimes this is hard to do, and you need to allow for situations that your didn't expect, otherwise people may view the processes are arbitrary and bureaucratic rather than useful.

When there are manual procedures, you need to motivate people to follow them. People tend to follow processes more closely when they understand the rationale behind the processes. Understanding the rationale allows people to make good judgement calls when the process does not precisely cover the situation at hand. It helps to be able to explain the process in terms of the structure it supports. But passing on a complete understanding of the system takes time.

There are some static things that underlie all of the processes that go on in a software engineering environment. Things like source code modules, executables, version control systems are concrete things; there is a process that you use to change things, but the process is closely tied to the other parts of the environment.

We want to take the "simple" things that we understand, and then compose them to describe how to help developers work together in teams.

Using patterns is different than other ways of looking at problems because how the patterns relate to each other is important as the problem and solution that they solve.(Stephen Berczuk 1994)

Some of the patterns here may seem obvious, but the details of how to get there are not always so obvious. This is part of their value. Alexander says that the process of applying patterns is valuable" not too much because it shows us things which we don't know, but instead, because it shows us what we know already, only daren't admit because it seems so childish, and so primitive." (Alexander 1979)

Structure of Patterns in this Book

The patterns in this book have the following parts:

- A *title* that describes what the pattern builds.
- A *picture* that can serve as a metaphor, and perhaps a mnemonic, for the pattern. The pictures in this book are from things in the real world of the past, rather than of software or technology. We also want to emphasize that the solutions here do have analogs in the "real world," that is, outside of software. We also hope that this interjects an element of humor, allowing you to better remember the pattern.
- A paragraph describing the *context* of the pattern; that is, when you should consider reading the pattern. In general, this will contain reference to other patterns, some of which are in this book, others that others have published elsewhere. Since there are aspects of the team development problem that we did not cover, there may also be a prose description.
- A concise statement of the *problem* that the pattern solves in bold.
- A detailed *problem description* illustrating trade-offs, some dead end solutions, and the issues that need to be resolved.
- A short summary of the *solution*
- A description of the *solution in detail*
- A discussion of *unresolved issues*, and how to address them. This will lead you to other patterns that can address these problems.

The pattern chapters may also have a section that suggests ways to learn more about the topic of the pattern.

The Pattern Language

This section will briefly describe how the patterns in this book are organized, and then discuss how to use the book.

As you work read through the patterns you may find that some match your current practice, while others do not. Teamwork is a complicated thing, and a there is no simple cookbook-like approach to using the patterns in this book that will work for everyone. The best way to approach the patterns is to read through them all until you have a high-level understanding of them, and then work through the patterns that are particularly relevant to you. After you have done this (or if you want to start attacking your problems now, here is one approach that we suggest starting out with:

- Identify the global problem that you are trying to solve. If you are reading this book, you are probably considering how to get team members to work together more efficiently.
- Look through the pattern context and problem description sections to identify the patterns that you already use in your organization, as well as any patterns that seem to solve pressing problems. The context sections mostly consist of other patterns in the language, but since no pattern language can be totally complete, the context of some patterns many describe a situation, rather than reference patterns.
- Once you identify which patterns you already have in place, start with that pattern. If you are trying to establish “mainline” development, start at the first pattern in the language.
- Apply the patterns as the language directs by looking at the context and unresolved issues sections.
- Repeat this process until you have worked through the language.

Each pattern can also stand on its own to some degree.

This book shows one path through the patterns, that for doing Mainline development. Parts of the language may be relevant for other situations. For example, the Private Workspace will be useful to you regardless of the sort of environment that you use.

These patterns are independent of the tools that you use. Some tools support some of these concepts explicitly, some less directly. But the concepts have proven themselves useful in may development environments. For example, some tools don’t support branching and merging as well as other tools. Clearly these are not practices that you want to do routinely without tool support, but there are times when the even rudimentary support for branching will make your life easier, as opposed to fearing avoiding branching. Where appropriate we discuss tool support in general terms so that you can see what features you need if you are looking for tools.We will discuss examples using common tools, but we will try to keep them at a level that will allow the book to stay current as tool interfaces change.

The pattern language in this book is focused on a team of developers working off of one (or a small number) of codelines for any given release. These developers work in their own private workspaces.

There are some issues around codeline organization for *large* projects that we do not address in full, but the book is more about how things work in the local development environment. For a wide range of systems the principles hold equally well.

Overview of the Language

Figure 3-1 shows the patterns in this language. The arrows show the relationships between the patterns that the context and unresolved issues section of each pattern describes. An arrow from one pattern (A) to another (B) means that pattern A is in the context of pattern B. This means pattern B is most useful once you have already thought about using pattern A.The arrow from A to B also means that pattern A needs pattern B to be complete.

The patterns in this pattern language can guide you towards establishing an active development environment that balances speed with thoroughness, so that you can produce a good product quickly. After first two patterns, the patterns fall into two groups, patterns that describe the structure of your workspace, and patterns that describe the structure of your codelines.

There are other approaches to development, than the Mainline approach that we discuss here. The Mainline approach works well in many circumstance, and other approaches may share many of the same patterns. It is important to remember as you read through these patterns that, while we hope that this advice is useful, only you understand what your set of circumstances is. Apply the patterns with care

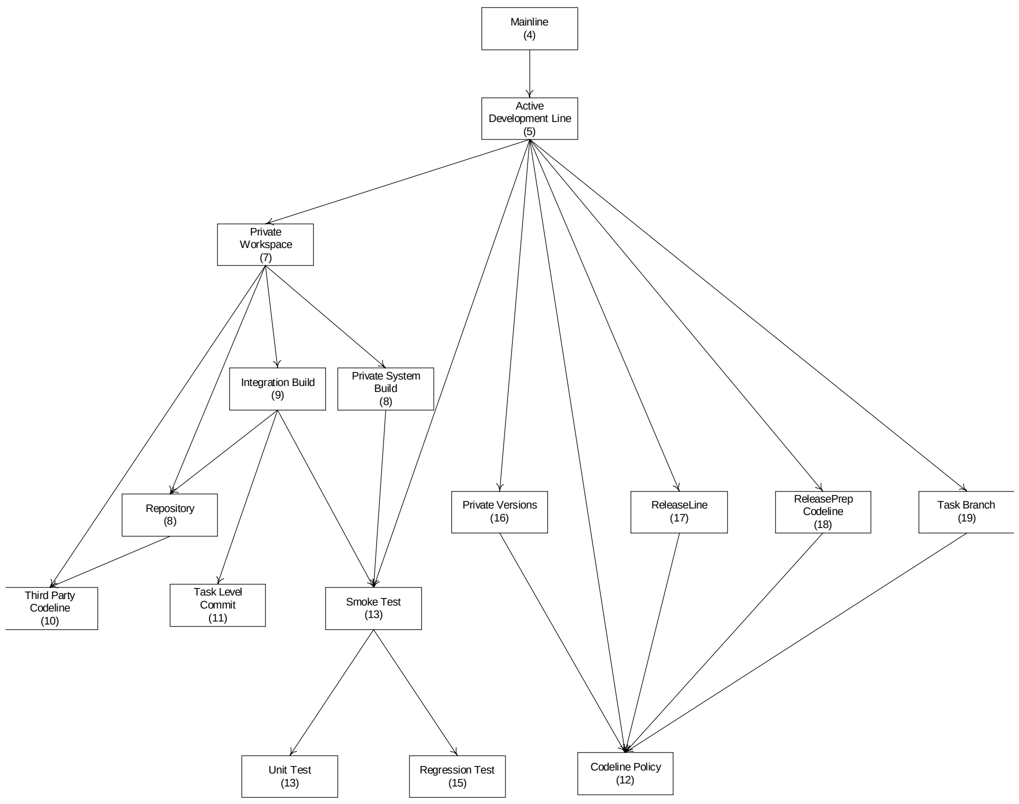


Figure 3-1 .The SCM Pattern Language

The patterns in the language can be grouped into two sets: codeline related patterns and workspace related patterns

The codeline related patterns help you to organize your source code and other artifacts in an appropriate fashion, both in terms of structure and time

The patterns relating to codelines are[1]:

- *MAINLINE (4)*
- *ACTIVE DEVELOPMENT LINE (5)*
- *THIRD PARTY CODELINE (10)*
- *CODELINE POLICY (12)*

1. References to patterns will show the name of the pattern, followed by the chapter number in parentheses.

- *PRIVATE VERSIONS (16)*
- *RELEASE LINE (17)*
- *RELEASE-PREP CODE LINE (18)*
- *TASK BRANCH (19)*

Figure 3-2 shows these patterns.

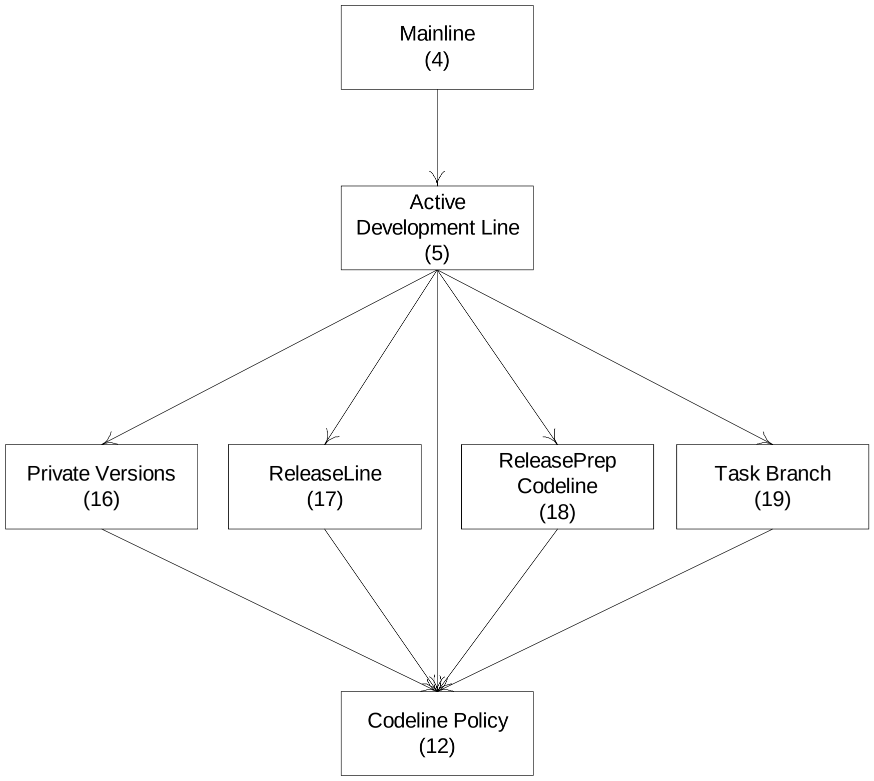


Figure 3-2 .Codeline related patterns

The patterns related to workspaces are:

- *PRIVATE WORKSPACE (6)*
- *REPOSITORY (7)*
- *PRIVATE SYSTEM BUILD (8)*
- *INTEGRATION BUILD (9)*
- *TASK LEVEL COMMIT (11)*
- *SMOKE TEST (13)*
- *UNIT TEST (14)*

- *REGRESSION TEST (15)*

Figure 3-3 shows these patterns.

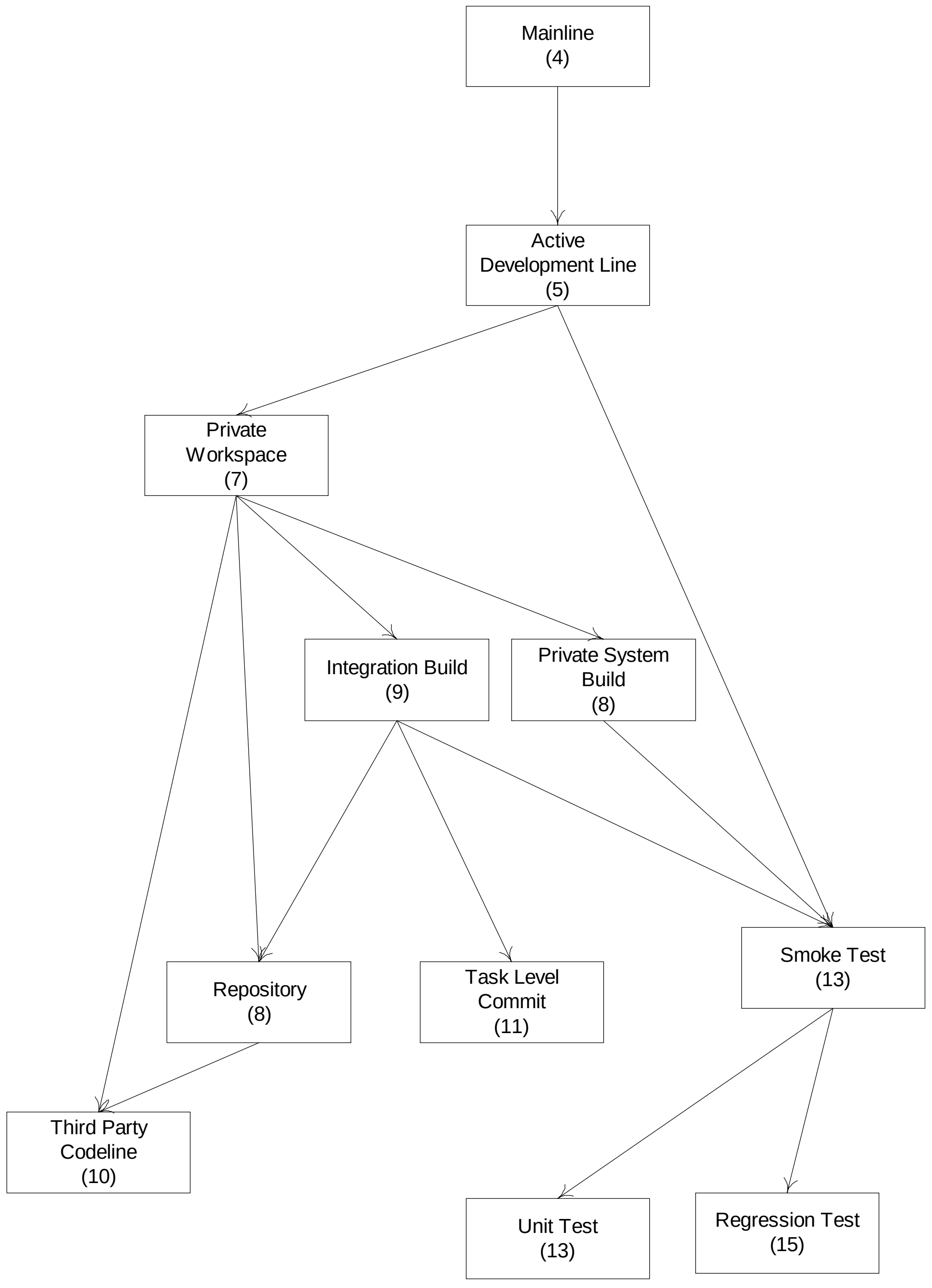


Figure 3-3 .Workspace Related Patterns

Unresolved Issues

This chapter gave you an overview of pattern languages in general, and the pattern language in this book. The rest of the book defines the patterns.

Further Reading

- The series of books by Alexander, et al describe the core principles of patterns and pattern languages. *A Pattern Language* (Alexander et al. 1977) gives a pattern language for building towns and buildings. *The Timeless Way of Building* (Alexander 1979) describes the principles behind patterns and pattern languages. *The Oregon Experiment* (Alexander et al. 1975) gives a concrete example of using a pattern language in a real situation. Some of these books are a bit long, and Alexander's prose is a bit tough to wade through at times (with some examples of marginally correct grammar), but the ideas the book expresses are excellent.
- Many of the patterns for software systems were workshopped at the Pattern Languages of Programming conference. A good collection of the patterns from these conferences appears in the *Patterns Languages of Program Design* series (Coplien and Schmidt 1995; Vlissides et al. 1996; Martin et al. 1998; Harrison et al. 2000).
- The Hillside group's patterns page is a good starting point for all things relating to software patterns, including information on the various patterns conferences. The URL is http://www.hillside.net/patterns.
- For examples of patterns applied to software systems, *Design Patterns* (Gamma et al. 1995) is the one book "everyone" has read. The Pattern-Oriented Software Architecture series books are another example. These books do not approach the richness of the Alexander books, but they provide a concrete example.
- Brandon Goldfedder's *The Joy Of Patterns* (Goldfedder 2002) is an excellent introduction to using the design patterns.
- *The Manager Pool: Patterns for Radical Leadership* (Olson and Stimmel 2002), is a collection of patterns on managing software organizations.

- *The Patterns Almanac* (Rising 2000) is one of the most comprehensive indexes of the state of software patterns in publication.

PART 1

The Patterns

Chapter 4
Mainline

When you are developing a software application as part of a team effort you often have to reconcile parallel development efforts.Your version control tool provides branching and merging facilities. You can use branches to isolate parallel efforts, but this can have a cost. This pattern shows you how to manage your codeline to minimize the integration effort that branching and merging requires.

◈ ◈ ◈

How do you keep the number of currently active codelines to a manageable set, and avoid growing the project's version tree too wide and too dense? How do you minimize the overhead of merging?

Generically, a branch is a means to organize file versions and show their history. (White 2000) More specifically, a branch is a configuration of a system that is derived from, and developing independently of, the base configuration. For example, a branch can consist of a snapshot of the system at release time, and all patches that you apply to the release. A branch can also be used to keep a subset of files that, when merged with the main system, produce a unique variant, such as a platform specific version, or a customer variant.

Branching is a powerful mechanism for isolating yourself from change.You can branch to isolate changes for a release, for a platform, for subsystem, for a developer's work; just about any time that you have work that goes off in a different direction. Whenever branches need to be integrated together, you need to merge the changes. For example, when you must integrate a bug fix for the last release into the current release. This isolation from change can have a cost. Even with good tools, merging can be difficult, since it is entirely possible to make two changes that conflict with each other (because of intent of the change if for no other reason), and you have no way of resolving the conflict without knowing the intention of the authors. You may have to make the change in both codelines manually. Any work that you thought that you would save by branching can be more than compensated for in the effort of a messy merge. Figure 4-1 illustrates this.

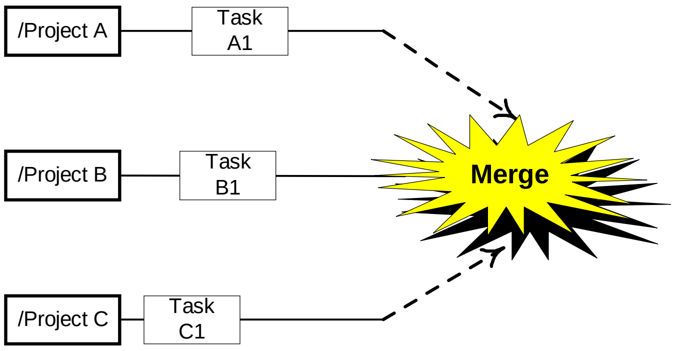


Figure 4-1 A Merge Can be Messy

Separate codelines seem like a natural way to organize work in some cases. As your product evolves, you may want to create more codelines to isolate and organize changes. This is helpful, because it is easy to allow the codelines to evolve in their own way. More codelines may mean more merging though, and more merging means more synchronization effort, and the practical costs of the merge may outweigh the improved apparent organization.

Some codelines are naturally derivative. It may be natural to think of each release using the prior release as a starting point, following a promotion model. This will give you a staircase codeline structure that can make it hard to determine where code originated, and making an urgent fix to a release without interrupting new development can be difficult with this structure. Figure 4-2 show this case. But with this structure the policy of the codeline will change from being active to development, and it requires developers to relocate work in progress to another codeline (Wingerd and Seiwald 1998)

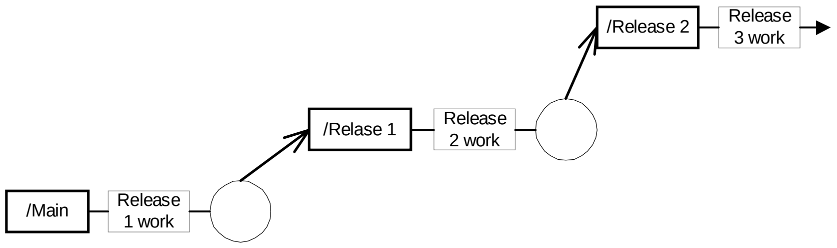


Figure 4-2 .Staircase Branching (or a Cascade)

Another use of a branch is to allow a subset of the team to work on a change with far reaching consequences. If they work on the branch, then they do not have to worry about breaking existing code, or about problems other people's changes can cause. The branch isolates each group from changes that another makes. This works well when you integrate back with the main body of code as quickly as possible, but if you branch simply to defer integration the merge will be difficult, and you are just putting off the inevitable.

Some argue for resisting the temptation to ever delay integration. Extreme Programming advocates continuous integration because the effort of integration is exponentially proportional to the amount of time between integrations (Fowler and Foemmel 2001), but sometimes parts of the codebase really are evolving in different directions, and independent lines of evolution make sense.

If you want to branch, creating a branch can be a simple matter with the right tools, but a branch is a fairly heavyweight thing. A branch of a project can be all of the components for a release, and some of their dependents. So creating that branch has serious consequences if you need to integrate any changes from the original codeline into it.

If you don’t branch, you lose the isolation that a branch gives you and your code all needs to work together for anyone to get anything done. But, in most cases, your code needs to work together in the end anyway.You need to balance the transient freedom that branching gives you with the costs that you will encounter when you need to re-synchronize.

You want to maximize the concurrency on your codelines while minimizing problems that deferred integration can cause.

Fear of Branching

People seem to be both fascinated by and fearful of branching. This ambiguity is often caused by people not understanding their tools, and not understanding their motivations for branching.

Branching (like most everything else) when done without good sound reasoning can hinder more than help. It is admittedly very easy to go branch-happy or branch-crazy and use them too much. Branching should definitely be done in moderation and with careful consideration. Going great lengths to avoid it however can often cause you more work in the end. So just as one should not branch “on whim” without good reason, also take care not to go to the other extreme of taking a lot of measures to avoid it based more on fear and the lack of information.

A great deal can also depend on the version control tool you use. Branching in a tool like VSS is very limited in capability and somewhat unwieldy, and not often recommended. Branching in CVS is much easier, but still not as good as with tools that have much better logical and visual support for branching or built-in intelligence to know which “paths” have already been merged

I have worked at places where someone discovers the branching facilities of a tool and then starts using them without thinking out a strategy. The branches create isolation but they sometimes isolate too much. The product reached a point where they have a ‘staircase’ branching structure: each branch has other branches off of it, and each of these branches is fairly long-lived and active. The when the time comes for a major change that affects many of the “branches”, the only good way to integrate the change is to manually add it to all of the branches. When people discuss situations like this with me, they quickly talk about how bad their tool is, since it won’t support their structure. They rarely discuss about whether the structure actually made sense.

The reality often is that structure does not fit the business model of the company. A simpler branching model would have worked better in practice. Also, the extent to which the company is willing to invest in tool support to allow complicated merges also indicates how complicated of a branching structure that they need. I've found that if you can't make a case for a tool that supports a complicated branch and merge structure, then management doesn't really understand why there should be such a structure.

At the other end of the spectrum are the people who decide to never branch. Often they had been in a situation where the branching and merging was hellish, and they avoid branches at all costs. This is, of course, short sighted, since there are many valid reasons to branch a codeline. Like any tool, you should understand how to use it, and how it can hurt you if you use it badly.

Simplify your Branching Model

When you are developing a single product release, develop off of a mainline. A mainline is a "home codeline" that you do all of your development on, except in special circumstances. When you do branch, consider your overall strategy before you create the branch. When in doubt, go for a simpler model.

The reason for a mainline is as "a central codeline to act as a basis for subbranches and their resultant merges" (Vance 1998).The mainline for a project will generally start with the codebase for the previous release or version. Of course, if you are doing new development, you start off with only one codeline, which is your mainline by definition.

Doing mainline development does not mean "do not branch." It means that all ongoing development activities end up on a single codeline at some time.

Don't start a branch unless you have a clear reason for it and the effort of a later merge is greatly outweighed by the independence of the branch. Favor branches that won't have to be merged often, for example release lines. Branching can be a powerful tool, but like any tool, it should be treated with respect and understanding. Have the majority of your development work off of a mainline. Use good build and development practice to ensure that what gets checked into the mainline is usable, but realize that the tip of the mainline is a work in progress and will not always be release quality.

Do the majority of your work on one codeline. The mainline need not be the root of the version control system. It can be a branch that you are starting a new effort on. The key idea is that all the work done on a release will be integrated quickly into one codeline.

You need to ensure that the code in the mainline always works in a reasonable fashion. It is to your advantage to maintain a continually integrated system anyway, because "big-bang" integration costs almost always exceed expectations.

When the time comes to create a codeline for a new major release, instead of branching the new release-line off of the previous release-line, merge the previous release-line back to the mainline branch and branch off the new release-line from there. Have a shallow branching model such as in Figure 4-3.

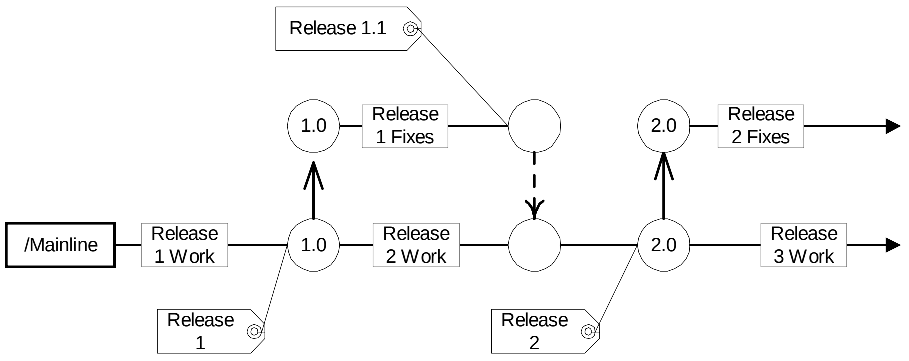


Figure 4-3 .Mainline Development

Mainline development offers the following advantages:

- Having a mainline reduces merging and synchronization effort by requiring fewer transitive change propagations.
- A mainline provides closure by bringing changes back to the overall work-stream instead of leaving them splintered and fragmented

There are still reasons to branch. You will want to branch towards the end of a release cycle to isolate a stable version of the codebase at that revision level. This will allow you to do bug fixes on the that branch without introducing new features and other work in progress to the already released codeline.

Limit branching to special situations, including:

- Customer releases. This is to allow bug fix development on the release code without exposing the customer to new feature work in progress on the mainline. You may want to migrate bug fixes between the release branches and the mainline, but depending on the nature of the work on each line, this too may be limited.
- Long lived parallel efforts that multiple people will be working on. If this work will make the codeline more unstable that usual, create a *TASK BRANCH (19)*. This works best when the mainline will only have small changes done on it.
- Integration. When you create customer release lines instead of doing a code freeze, create an integration branch on which your developers will do all their work. This will allow progress to continue on the mainline. Bug fixes in the integration line need to be integrated into the mainline, but this should not be too difficult as the release should be close to what is currently active.

Some situations will still require a branch. But you will generally want to think hard before branching; ask yourself if the work really requires a branch. Branches have many uses, but you want to avoid long-lived branches that must be merged later.

To do mainline development:

- Create a codeline (/main) using the current active code base as a starting point
- Check in all changes to this codeline.
- Follow appropriate per-check in test procedures to keep this mainline useful and correct.
- Mainline development can greatly simplify you development process. Wingerd and Seiwald report that “90% of SCM ‘process’ is enforcing codeline promotion to compensate for the lack of a mainline (Wingerd and Seiwald 1998).”

Unresolved Issues

One you decide to have a mainline, you need figure out how to keep the mainline usable when many people are working on it. *ACTIVE DEVELOPMENT LINE (5)* describes how to manage this.

Further Reading

- SCM Tool manuals, such as those for CVS and Clearcase describe how to manage mainline and release line development. You can download and find out about CVS at http://www.csvhome.com/
- *Open Source Development with CVS* (Fogel and Bar 2001)describes how to use the popular open source tool CVS.
- *Software Release Methodology* by Michael Bays (Bays 1999) has a good discussion about branching strategies and issues around merging.
- *Software Configuration Management Strategies and Rational Clearcase* (White 2000)Brian White discusses branching strategies. While is book is focused on ClearCase, there is good generic information in the book.

Chapter 5
Active Development Line

You have an evolving codeline that has code intended to work with a future product release.You are doing most of your work on a *MAINLINE (4)*. When you are working in dynamic development environment many people are changing the code. Team members are working towards making the system better, but any change can break the system, and changes can conflict. This pattern helps you to balance stability and progress in an active development effort.

◈ ◈ ◈

How do you keep a rapidly evolving codeline stable enough to be useful?

You develop software in teams because you want there to be concurrent work. The more people you have working on a codebase, the more that you need communication among team members. You also have more potential for conflicting changes.

For the team as a whole to make progress you need synchronization points where the work comes together. As in any concurrent system, having a synchronization point means that there is a possibility for deadlock or blocking if we don't manage the coordination correctly. Deadlock can happen when 2 people have mutual dependencies, and a check in before a test finishes can mean that the test would fail the next time we ran it. Blocking can happen when the pre-checkin process takes too long and someone else needs their changes to proceed.

If you think of software development as a set of concurrent processes, we have to synchronize whenever we check a change into the codeline in the source control system. When someone checks in a change into the codeline he can cause delays to the team as a whole if the change breaks someone else's work. Even if you tested the change with what you thought was the latest code, it could still be incompatible with the change that got checked in moments before. But if we put too much effort into testing for changes, making sure that a change works with every change that got checked in while we were running the last set of tests, we may not be able to check in our changes in a reasonable amount of time. That can cause delays too.

Working from a highly tested stable line isn't always an option when you are developing new features. You want to use your version control system to exchange work in progress for integration, and there may be many features that you don't have integration tests for yet because they are so new.

You want to be able to get current code from the source control system and have a reasonable expectation that it will work.You want the codeline to be stable so that it does not interfere with people's work. You can require that people perform simple procedures before submitting code to the codeline, such as a preliminary build, and some level of testing. These tests take time though, and the delay between check ins may work against the some of the projects greater goals. A broken codeline slows down everyone who works off of it, but the time it takes to test exhaustively slows down people as well, and in some cases can provide a false sense of security, since you can never fully test a system that is in flux. Even if you do test your code before checkin, concurrency issues mean that two changes, tested individually, will result in the second one breaking the system.And the more exhaustive — and longer running

— your tests are, the more likely it is that there may be a non-compatible change submitted.

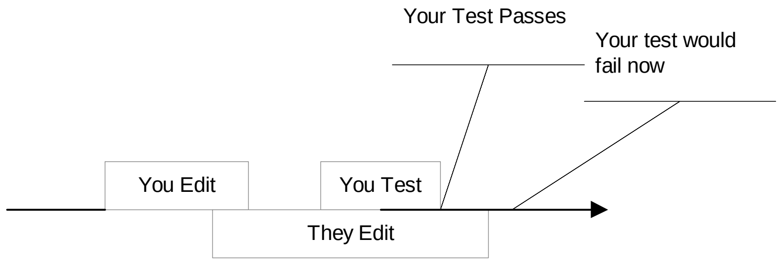


Figure 5-1 .Long RunningTests have MixedValue.

You can prevent changes from being checked into the codeline while you are testing by using semaphores, but then only one person can test and check in changes at a time, which can also slow progress. Figure 5-2 shows a very stable, but very slowly evolving codeline.

You can also make changes to your codeline structure to keep parts of the code tree stable, creating branches at various points, but that adds complexity, and requires a merge.

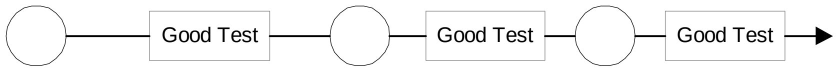


Figure 5-2 .A Stable, but Dead, Codeline.

You can go to the other extreme, and make your codeline a free-for all. Figure 5-3 shows a quickly evolving but unusable codeline.

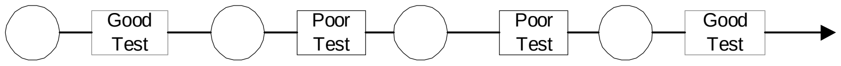


Figure 5-3 .AVery Active, but very Useless Codeline

You can also change the module architecture of the system to reduce the likelihood of conflicting change, but, even then you may still have two people changing the code in a way that breaks something.

Aiming for perfection is likely to fail in all but the most static environments.You can achieve stability on a given codeline, but with process and synchronization overhead, increased merging, and more complicated maintenance and administration. This is not always worthwhile.You want a balance: an active codeline that will more likely than not, be usable most of the time.

Testing Treadmill

I have worked at a number of startup companies and there is a recurring theme that goes like this: Initially, there are only a few people working on the product. They understand what they are doing very well, and even when they step on each other's work, they recover quickly. Then the company grows, and the code in version control is hardly every consistent. The tip of the mainline always breaks. In frustration, someone sets up test suites that people should run before doing a check in to the source control system. The first cut at this test suite is every test that they can think of. The test suite grows and soon it takes an hour to run the pre-checkin tests. People compensate by checking code in less often, causing pain when there are merges or other integration issues. Productivity goes down as well. Someone suggests shortening the test suites, but they are met with resistance justified by cries of "We are doing this to ensure quality." Someone else comments that "the pain is worth it, considering what we went through last year when we had no tests.

But, once we reached a basic level of stability, the emphasis on exhaustive testing lead to diminishing returns as progress as a whole was reduced. This gets worse when the tests are not exhaustive, but simply exhausting to the developers who run them.

Define your goals

Institute policies that are effective in making your main development line stable enough for the work it needs to do. Do not aim for a perfect active development line, but rather for a mainline that is usable and active enough for your needs.

An active development line will have frequent changes, some well tested checkpoints that are guaranteed to be "good," and other points in the codeline are likely to be good enough for someone to do development on the tip of the line. Figure 5-4 shows what this looks like.

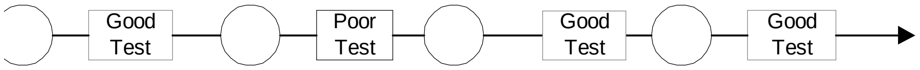


Figure 5-4 .An Active, Alive, Codeline

The hard part of this solution is figuring out how "good" your codeline really needs to be. You need to go through a process that is similar to doing a requirements analysis for building software system. Your clients want perfection and completeness, and they want it quickly and cheaply. These goals are, in reality, unattainable. Do an analysis along the following lines:

- Who uses the codeline?
- What is the release cycle?
- What test mechanisms do we have in place?
- How much is the system evolving?
- What are the real costs will be for a cycle where things are broken?

For example, if the codeline is being used by other teams that are also doing active development, some instability is appropriate and the emphasis should be on speed. If this is the beginning of the new development, or if the team is adding many new features, you expect more instability. If this codeline is basically stable, and being used as a standard component, more validation is appropriate. Right before you want to branch or freeze for a release, you want more stability, so you, you want to test more.

Understand your project's rhythm. Kane and Dikel define rhythm as "the recurring, predictable exchange of work products within an architecture group and across customers and suppliers"(Dikel et al. 2001) A good project rhythm is especially important for architecture centric development, but any project that has concurrent work with dependencies needs a good rhythm. The source control structure can influence how the rhythm is executed, and culture helps define what rhythm you need.

If you have good unit and regression tests, run either by developers or as part of the system build post checkin, errors will not persist as long, so emphasize speed on checkin. If you do not have a good testing infrastructure, be more careful until you develop it. If you want to add functionality, emphasize speed.

If a client needs a good deal of stability, they should only used *NAMED STABLE BASES (20)* of the components, this will allow them to avoid "cutting edge" work in progress. Figure 5-5 shows how these baselines are identified and labeled when they pass tests. But these clients should then be treated more like external clients than members of the active development team.

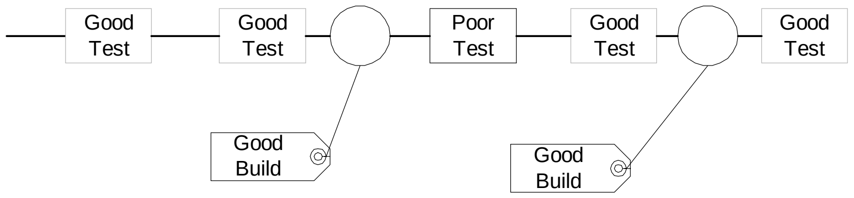


Figure 5-5 .Labeling Named Stable Bases

Don’t be too conservative. People can work with any system as long as they understand the trade-offs and the needs. You don’t want to make the checking process too difficult. If you have a pre-checkin process that takes a long time you run the risk of developers doing larger grained, and less frequent checkins which will slow feature progress. Less frequent checkins increase the possibility of a conflict during testing, and make it harder to back out of a problematic change.

Establish a criteria for how much to test the code before checkins: “The standard needs to set a quality level that is strict enough to keep showstopper defects out of the daily build, but lenient enough to disregard trivial defects (because undue attention to trivial defects can paralyze progress).” (McConnell 1996). Martin Fowler suggests for the purposes of continuous integration (Fowler and Foemmel 2001):a selected suite of tests runs against the system successfully. The more exhaustive the tests, the longer the pre-check-in time. You need to determine how much stability is *really* necessary for your purposes.

Remember that there is a fundamental difference between code that close to release and code that is being actively changed. If you need a stable codeline, perhaps what you want isn’t the active development line, but rather a fully QA’d *RELEASE LINE (17)*. There are significant benefits in the form of catching potential problems early in developing with an *ACTIVE DEVELOPMENT LINE*. You can also push off your more exhaustive testing to a batch process that creates your *NAMED STABLE BASES (20)*.

To prevent total chaos on the mainline set up each developer with a *PRIVATE WORKSPACE (6)* where they can do a *PRIVATE SYSTEM BUILD (8)*, *UNIT TEST (14)* and *SMOKE TEST (13)*.

Have an integration workspace where snapshots of the code are built periodically an subjected to more exhaustive tests.

Any SCM tool that supports “triggers” or automatic events that happen after a change is submitted will help automate the process of verifying that you are meeting the quality metric. You can then set up the system to run a build or a set of tests after a change is submitted. You can also set up the system to run less often.

As Jim Highsmith (Highsmith 2002) writes, “Change is expensive. no question about it. But consider the alternative — stagnation.”

Unresolved Issues

Once you have established that a ‘good enough’ codeline is desirable, you need to identify the codeline that will be like this. *CODELINE POLICY (12)* will establish which lines follow this form, and what the checkin / commit process is for these (and other) codelines.

An individual developer still needs isolation to keep the Active Development Line alive. He can do this by working in a *PRIVATE WORKSPACE (6)*.

When the need for stability gets close, some work will need to be broken off to a *RELEASE-PREP CODE LINE (18)*.

Some long lived tasks may need more stability that an active development line can provide, even though you realize that there may be an integration cost later. For these, use a *TASK BRANCH (19)*. Doing this also insulates the primary codeline from high risk changes.

Further Reading

- One reason people resist applying this pattern is that they think that their problem is that code is not perfect, when in fact the problem is that it is too hard to change and evolve the code. A great book about getting to the core of the “real” problem is *Are Your Lights On?* (Gause and Weinberg 1990).
- *Agile Software Development Ecosystems* (Highsmith 2002) discusses the reality of continuos change in most projects.

Chapter 6
Private Workspace

In an *ACTIVE DEVELOPMENT LINE (5)* you and other developers make frequent changes to the code base, both to the modules that you are working on and to modules that you depend on. You want to be sure that you are working with the latest code, but since people don't deal well with uncontrolled change, you want to be control when you start working with other developer's changes.This pattern describes how you can reconcile the tension between always developing with a current code base, and the reality that people cannot work effectively when their environment is in constant flux.

❖ ❖ ❖

How do you do keep current with a continuously changing codeline, and also make progress without being distracted by your environment changing out from under you?

Developers need a place where they can work on their code isolated from outside changes while they are finishing a task

When a team develops software, people work in parallel, with the hope that the team gets work done more quickly than any individual. Each individual makes changes in parallel with the other team members.You now have the problem of managing and integrating these parallel streams of change. Writing and debugging code, on the other hand, is a fairly linear activity. Since in team development there are concurrent changes happening to the codeline while you are working on your specific changes there is a tension between keeping up to date with the current state of the codeline, and the human tendency to work best in an environment of minimal change. Changes that distract you from your primary purpose interrupt your flow. DeMarco and Lister define "flow" as "a condition of deep, nearly meditative involvement."(DeMarco and Lister 1987). In *PeopleWare*, the authors discuss flow in terms of noise and task related interruptions, but integrating a change that is not related to the task at hand can have a similar effect.

Developing software in a team environment involves the following steps:

- Writing and testing your code changes
- Integrating your code with the work that other people were doing.

There are two extreme approaches to managing parallel change, literal continuous integration, and delayed integration.

You can integrate every change a team member makes as soon as they make it. This is the clearest way to know if your changes work with the current state of the codeline. The down side of this "continuous integration" into your workspace approach is that you may spend much of your time integrating; handling changes tangential to your task. Frequent integration helps you isolate when a flaw appeared. Integrating too many changes at once can make it harder to isolate where the flaw is, as it can be in one of the many changes that have happened since you integrated. Figure 6-1 shows this.

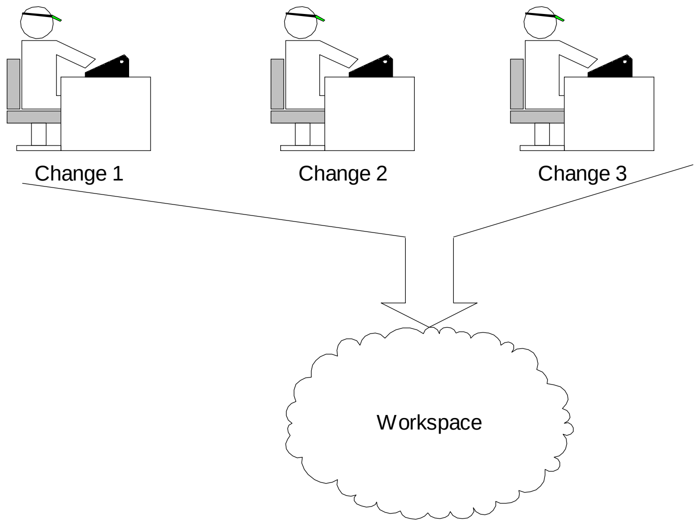


Figure 6-1 .Combining Changes at Once.

Even when you do “continuous integration,” as when you are doing Extreme Programming, you really integrate in discrete steps, as when a day’s work is complete. Figure 6-2 shows this case.

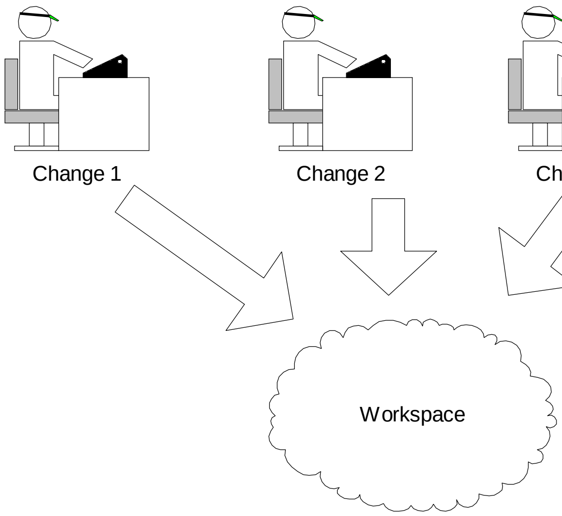


Figure 6-2 .Integrating each change as it happens

You can integrate at the last possible moment. This makes it simplest for you, the developer while you are working, but it means that you may have a large number of outside integration issues to deal with, meaning that it will take longer to integrate at the end.

You can “help” developers keep up to date by having them work out of a shared source / release area, only keeping local copies of the components that they are modifying. But you really don’t want to have things change unexpectedly. Also, a change in one of the other components can have an effect on your work. If you are coding in a language like C++, a change in a header can cause a compilation problem. A change in source can cause a behavior problem. Even with a highly modular architecture, components will interact, making it hard to get consistent results across a change.

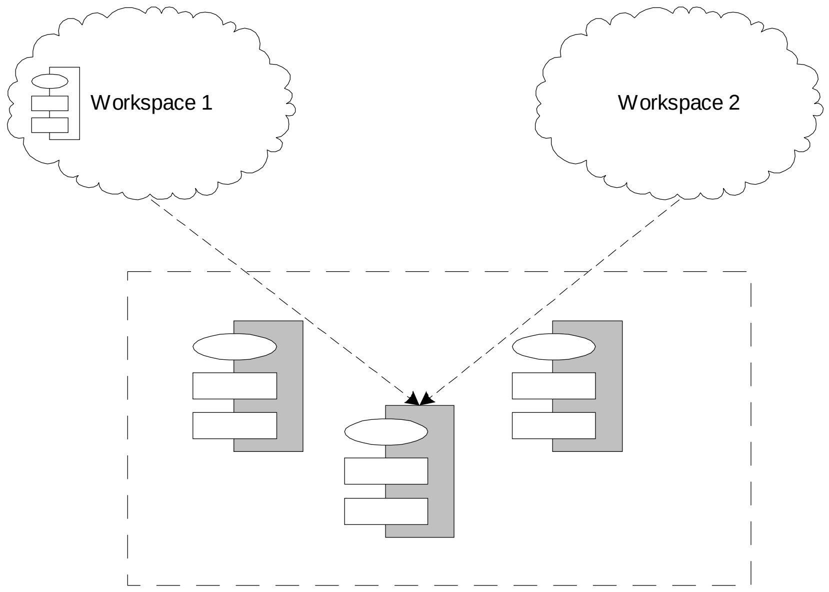


Figure 6-3 .Sharing Some Components between Workspaces.

And there are also times that you are working on things other than the latest code base. You must interrupt your work on the current release to work on the code at an earlier point in time. Or you many need to experiment with a new feature. Sometimes you can't be up-to-date and still do your work.

You can also avoid the problems of continuous updates by taking a snapshot of the entire system, and performing all of your coding tasks against the snapshot. This overly conservative approach can cause problems when you get behind the leading edge of changes. You many find yourself introducing problems into the global environment.

You need a way to control the rate of change in the code that you are developing with without falling too far out of step with the evolving codeline.

A Simple Plan

To some, this sounds like an easy to solve problem with an obvious solution. When I was interviewing for a job at a startup company 6 years into my career I discovered that some obvious solutions are easy to miss if you are not thinking about the context that you have. The company has fully bought into the idea of nightly builds.The problem was that each developer worked out of a shared product area, so after a night of working on a problem, you could come in the next day to find your development environment had changed dramatically, and then have to spend half the day simply getting to where you were the night before.

This illustrates one problem with blindly following a "good idea" without thinking through the reasons for using them.

Isolate Your Work to Control Change

Do your work in a Private Workspace where you control the versions of code and components that you are working on. You will have total control over when and how your environment changes.

Every team member should be able to set up a workspace where they have a consistent version of the software. A concise definition of a workspace is "a copy of all the 'right' versions of all the 'right' files in the 'right' directories."(White 2000) A workspace is also a place "where an item evolves through many temporary and inconsistent states until is checked into the library." (Whitgift 1991). You should have total control of when parts of the system change. You control when changes get integrated into your workspace. The most common situation is when you are working on the tip of the codeline along with other team members, but when you are working on a version that is not the latest, you can recreate any configuration necessary.

A Private Workspace comprises:

- Source code that you are editing.
- Any locally built components.
- Third party derived objects that you cannot or do not wish to build

- Built objects for all the code in the system.You can build these yourself, have references to a shared repository (with the correct version), or copies of built objects.
- Configuration and data that you need to run and test the system.
- Build scripts to build the system in your workspace.
- Information identifying the versions of all of the components in the system.

A Private workspace should not contain:

- Private versions of system-wide scripts that enforce policy. These should be in a shared binary directory so that all users get the latest functionality
- Components that are in version control, but that you copied from somewhere else.You should be able to consistently reproduce the state of your workspace when you were performing a task, by referencing a version identifier for every component in the workspace.
- Any tools (compilers, etc.) that must be the same across all versions of the product. If different versions of the product require different versions of tools, the build scripts can address this by selecting the appropriate tool versions for a configuration.

In addition, a Private Workspace can include tools that facilitate your work, as long as the tools are compatible with the work style of the team.

To do your coding for mainline development follow a procedure similar to this:

- Get up to date. Update the source tree from the codeline that you are working on so that you are working with the current code and build, or repopulate the workspace from the latest system build. If you are working on an different branch or baseline, create a new Private Workspace from that branch.
- Make your changes. Edit the components that you need to change.
- Do a *PRIVATE SYSTEM BUILD (8)* to update any derived objects.
- Test your change with a *UNIT TEST (14)*.
- Update the workspace to the latest versions of all other components by getting the latest versions of all components that you have not changed.
- Rebuild. Run a *SMOKE TEST (13)* to make sure that you have not broken anything.

If your system is small enough, you can simply get source and any binary objects for the correct configuration of all of the product components and build the entire system. You might also consider getting the latest code from the *MAINLINE (4)* and building the entire system if it does not take too long. This will ensure that the system that you are running matches the source code. With a good incremental build environment, doing this should work rather well, allowing for, perhaps, the one time cost of the whole system build.

In more complex systems, or where you are especially intolerant of problems, populate the environment by getting the source and object files from a know good build (*NAMED STABLE BASES (20)*). You can also get all of the source files from the *MAINLINE (4)*, since this will probably simplify debugging. Get whatever external components you need from the *THIRD PARTY CODELINE (10)*. All of these components should be of the correct configuration (version, label, etc.) for the system that you are working on. Get private versions of all of the source components that you will be changing.

If you are working on a multiple tasks, you can have multiple workspaces, each with their own configurations. For example, you can have a 'release 1.1' workspace to fix problems in the old release, while doing new development in a 'release 2' workspace. These can be separate and complete workspaces. It is not work the effort, in most cases, to save space by factoring out common components. (For example, if component X has not changed between release 1.1 and release 2, it is worthwhile to simply have 2 copies of this component. If X changes in release 2 later on it will be easy to update the "release 2" workspace without affecting the release 1 workspace.

Be sure that any tests / scripts / tools / etc. use the correct execution paths so that they run with the correct workspace version, and not a component from another workspace, or an installed version of the product. One way to do this is to deploy all local components in one binary directory and put the current directory in the path. Another way is to start up tests in a script that sets the environment.

Some component environments, such as COM, define certain items on a machine wide basis, so be sure to have a mechanisms to switch between workspaces by unregistering and registering the appropriate servers.

To be sure that you have built all dependencies, do a *PRIVATE SYSTEM BUILD (8)*. Check that your changes integrate successfully with the work others have done in the meantime by getting the latest code from the *MAINLINE (4)* (exclusive of changes you have made). If you are working on multiple tasks at one time, your workspace should have many workspaces.

One risk with a *PRIVATE WORKSPACE (6)* is that developers will work with old "known" code too long, and they will be working with outdated code. You can protect yourself from this by doing periodic *Private System Build*s and making sure that changes do not break the build or fail the *SMOKE TEST (13)*. (The sidebar "Update Your Workspace to Keep Current"discusses the *Workspace Update* in more detail.)

The easiest way to avoid getting out of date is to do fine grained tasks, checking in your changes after each one, and also updating your workspace before starting a new task. Some people find it useful to establish a discipline of creating a brand new workspace periodically to avoid problems that stray files might cause, and preventing the "works for me" syndrome. This is not ideal, but is an adaptation to the reality that some version control tools do an imperfect job of updating, particularly when you move files within the system.

Having a *PRIVATE WORKSPACE (6)* does take more space than working with shared source, but the simplicity that it adds to your work is worth it.

An automated build process should also have its own workspace. Set up this workspace would always get all the updates, if you are doing a "latest" build.

Good tool support makes managing a combination of shared and private components easy, but you can get quite far by using basic version control tools and/or scripts. For example, if your system can be built rather quickly, but uses some third party components, your checkout process can populate your workspace from version control with all the source from your system, and the built objects for the third party components. After you build our product code you will have a complete system.

A *SMOKE TEST (13)* allows you to check that your changes don't break the functionality of the system in a major way. A well designed smoke test will help you to minimize the amount of code that you need to keep in your workspace and rebuild, since the smoke tests should test the features that clients of your module expect.

Some work touches large parts of the codebase, and takes a long time to finish. In these cases a *TASK BRANCH (19)* may be the more appropriate approach.

Depending on your specific goal, there are a number of variations to this pattern, including, Developer workspace, Integration Workspace, and a Task workspace, in which case a developer has a number of workspace in his area concurrently.

There are variants of a workspace that are used for specific purposes. For example, an *integration workspace*, which is where changes are combined with the current state of the system, built and tested. This can also be called a build workspace, and may exist on the integration or build machine.

Update Your Workspace to Keep Current

After a workspace has been populated, the codeline may continue to evolve. If the work in your workspace is isolated for too long, the versions in the workspace can become outdated. A *workspace update* operation will "refresh" the outdated versions in your workspace, replacing them with the versions from the latest stable state of the codeline. If any of the files you changed are also among the set of "newer" files from the codeline, then merge conflicts may occur and will need to be reconciled.

You should do a *Workspace Update* before you merge your changes back to the codeline during a *TASK LEVEL COMMIT (11)*. You will need to rebuild using a *PRIVATE SYSTEM BUILD (8)* or at least recompile immediately after the update to quickly find and fix any inconsistencies introduced by the new changes. If desired, immediately prior to updating your workspace, checkpoint it using a label or *PRIVATE VERSIONS (16)* to ensure you can rollback to its previous state.

You may also update your workspace at known stable points, as well as right before you are about to check out a new set of files, to ensure that your workspace remains stable without growing "stale". This allows you to find out early on if any recently committed changes conflict with any changes in your workspace. You may then reconcile those changes in your private workspace at incremental intervals, instead of waiting until the end to do all of them at once.

Unresolved Issues.

Once you have stability for yourself, you still need to prevent introducing errors into the system when you check in your changes. *PRIVATE SYSTEM BUILD (8)* will let you check that your system does not break the build, and will also allow you to do an incremental build for the parts of your system when you do an incremental update from version control for other components.

You will need to populate your workspace from a *REPOSITORY (7)* containing all of the source and related components. Externally provided components will need to come from a *THIRD PARTY CODELINE (10)*.

Once you are done with your local work, it needs to get incorporated into the rest of the system in a *INTEGRATION BUILD (9)*.

Further Reading

- Brian White in *Software Configuration Management Strategies and Rational Clearcase: A Practical Introduction* (White 2000) has a good description of the various types of workspaces that ClearCase supports (ClearCase calls them "views"). He says that "one of the essential functions of an SCM tools to establish and manage the developers' working environment, often referred to as a "workspace" or a "sandbox."
- Private workspaces are a common practice in successful development organizations; so common they are often not described as such. Managing change, consistent build practices, and other essential components of Private Workspaces are all part of the practices that classic books like *Code Complete* (McConnell 1993) and *Rapid Development* (McConnell 1996), among others, describe.

Chapter 7
Repository

To create a *PRIVATE WORKSPACE (6)* or to run a reliable *INTEGRATION BUILD (9)* you need the right components. This pattern shows you how to build a workspace easily from the necessary parts.

◈ ◈ ◈

How do you build get the right versions of the right components into a new workspace?

Any software development activity that you perform starts off with a workspace where you have the components necessary to build, run and test your software. You need the right versions of everything that comprises the system so that you can build, run, and test your software so that you can accurately diagnose problems before you check in changes.

You want to easily get the elements of your workspace so that you can reliably create an environment that allows you to do your work using the right versions of the software, whether you are working with the current active codeline, or with an earlier version of the code base.

As Figure 7-1 shows, workspace consists of more that just code. Some of the things that you need to build and test a software some of the things that you need include:

- The source code that you are working with.
- Components that you are not working with, either as source, or library files.
- Third party components, such as jar files, libraries, dlls, etc. - depending on your language and platform.
- Configuration files
- Data files to initialize your application
- Build scripts and build environment settings so that you can get a consistent build
- Install scripts for some components

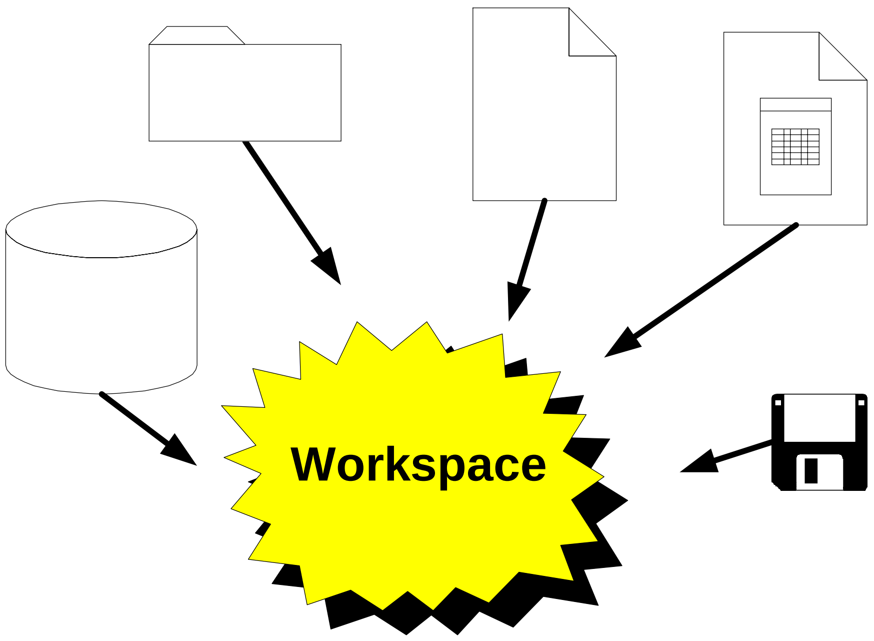


Figure 7-1 .A Workspace is created from many things.

Some of these elements have natural origins.You could get source code from your version control system, you could copy install scripts for some components, and third party built components from a development server. You can get other things that are not in source control from a server. Multiple locations for various resources add a lot of overhead to tasks for already busy software developers. You spend most of your coding time using tools such as an IDE, a compiler, your version control system, and copying files from multiple source points leaves opportunity for error.

If you get tired of performing many manual operations to update your workspace, you may decide to write your own tool to keep your self in synch, and even share the tool with the rest of the team. While developing such tools will save you time, you still run the risk of having the tool get out of synch with any new locations or policies, and you may incur a maintenance burden for the tool that distracts from your work.

You need to be sure that are associating the right versions of each element. For example, you might switch versions of a third party library in the middle of a product release cycle. You can keep people up to date about changes by communication.You can tell them to use a new version of the database API classes, and they may remember to update at the right time. It is difficult, however, for people to reliably know keep track of these details on their own. Someone could be in the middle of a task when they receive an e-mail about a change to a third party component, and forget to update their workspace after a task. It is difficult to remember what configurations went together for an earlier release.

A manual update process takes time. And if you missed the update notice, you may have a number of locations to check. This can take a lot of time, especially if the components are in different places.

Another issues is going back to the correct configuration at a point in time.You can maintain a list of configuration components, and have people refer to it when recreating a test environment, but that is error prone.

You keep your source code in a version control system, and adding more places for people to look for things adds to the complexity of creating a new workspace. If it is too hard to create a workspace, you may feel a natural reluctance to keep up to date, or to create another workspace to work on another codeline.

So Many to Choose From

There have been many times when I wanted to test our product while working from the tip of the active codeline, rather than from the latest install. The latest install often lagged the codeline by a day or so, and I wanted to see if a recent change that someone else made addressed a problem that I was having.

Because of the nature of the product, and structure of our organization, the development environment only had a portion of the configuration data and libraries from other groups kept up to date. There were nightly builds of these components, but they were staged into the development area when someone needed an update. So we could go days, or weeks, without realizing that something had changed.

When I tried to test, there were many mismatches between the server and client components. After 3 days, I discovered the problem and wrote some scripts (with the help of our release team) to enable developers to get the latest library files and configuration files from the source control tree.

How did this situation arise? In part because of lack of communication between the release engineering team, and the various product teams. And in part because of a desire to control change too much. It is admirable not to accept changes in an unexpected manner, but if a library component has changed months ago, it is already in the product. If the development team cannot see a problem, then they are wasting the companies time when they leave the QA team to find all of the integration issues.

The problems start when developers can't reproduce bugs in their development environment because they are (unknowingly) out of synch with the released version. Or they are reluctant to work on the correct version of the code base because it takes too long to set up.

These places are in better shape than if they had no version control system, but they waste lots of developer time and energy.

One Stop Shopping

Have a single point of access, or a Repository, for your code and related artifacts. Make creating a developer workspace as simple and as transparent as possible.

Make the mechanism that you use to create a workspace simple and repeatable.You should be able to create a workspace that contains artifacts from any identifiable revision of the product, including third party components and built artifacts such as library files. The mechanism should also make it easy to determine if there is a new version of an existing element, or a new component that you need when you are working on the tip of a development. Figure 7-2 shows this.

There are many ways to implement this pattern, and the details depend on the features of your version control tool, your build environment, and to a certain extent, your team's culture. The are three requirements on your implementation:

- It should be easy to use and repeatable. This is in the spirit of what Andy Hunt and Dave Thomas refer to as "Ubiquitous Automation" in (Andy Hunt and Thomas 2002a) and (Andy Hunt and Thomas 2002b).
- It should give you *all* the components that you need to create a *PRIVATE WORKSPACE (6)* for working on a particular state of your project, including build scripts and built objects.
- It should work for all versions of the project.

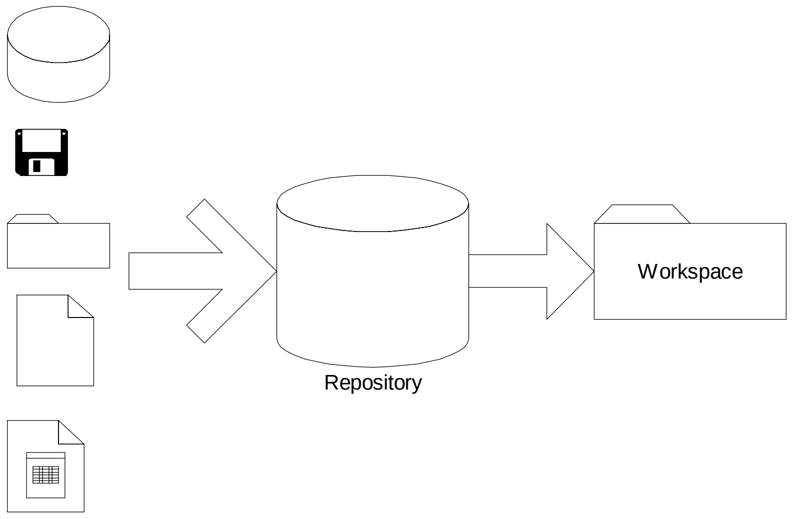


Figure 7-2 .Populate Your Workspace from a Repository

Some common implementations are using the version control system as a repository for all artifacts, and using scripts in combination with the version control system.

Since you have a version control system, a straightforward way of implementing this pattern is to place all source files, configuration files, build scripts, and third party components in it. Identify the set that is relevant to a particular version of product by using labels or creating a branch with the source codeline, third party codeline, etc. This makes it easy to create a new workspace.You issue the 'get' command and specify the version of the product that you want.You can now also identify when something changes. If a third party component changes, your version control system's 'update" command will get you a new version. If a component is the same as you already have, it will not update it.

This way, the version control system mirrors the build environment, and the history of changes in the build environment can also be tracked.

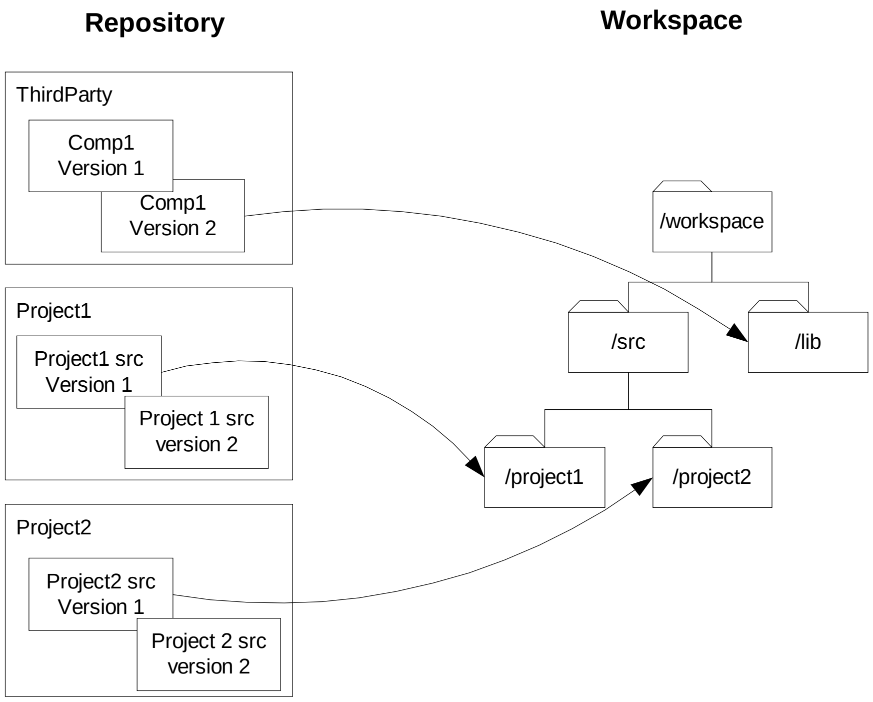


Figure 7-3 .VersionTree for a workspace

If the version control tree is not a direct mapping onto the build tree, and your version control tool does not provide an easy way to perform the mapping, or there is some reason to not keep every component in version control (some version control systems do not handle binary files well, and there may be a need to keep library files elsewhere if they do not change frequently) use a script or a makefile that copies the appropriate versions of the appropriate files to the appropriate places, depending on the version that you want. This might be the easiest and best way to populate your workspace with the result of a nightly build.

All of your configuration files, etc. can also be tied together. If a new database library needs new configuration settings, you can give them the same label, or check them into the tip of the version tree together.

Some tools that you can use to help you create this script are make and ANT. Both have interfaces to common version control systems.

Unresolved Issues

Organize third party code using *THIRD PARTY CODELINE (10).*

Further Reading

- Tools such as make (http://www.gnu.org/software/make/make.html) and ANT(http://jakarta.apache.org/ant/) are very helpful in automating the process of keeping a workspace up-to-date with your repository.
- The book *Java Tools for Extreme Programming: Mastering Open SourceTools Including Ant, Junit, and Cactus.* (Hightower and Lesiecki 2002) discusses how to use some of these tools.

Chapter 8

Private System Build

A *PRIVATE WORKSPACE (6)* allows you, as a developer, to insulate yourself from external changes to your environment. But your changes need to work with the rest of the system too. To verify this, you need to build the system in a consistent manner, including building with your changes. This pattern explains how you can check to see if your code will still be consistent with the latest published code base when you submit your changes.

◈ ◈ ◈

How do you verify that your changes do not break the build or the system before you check them in.

In a development team with liberal codeline policies changes happen very fast.You change existing code, add new modules to the codeline, and perhaps change the dependencies.

The only true test of whether changes are truly compatible is the centralized integration build. But checking in changes that are likely to break the build wastes time. Other developers will have you suffer through mistakes that you could have fixed quickly, and unless the system build turnaround is very short, it will be harder for you to recall what the source of an error will be because you may have lost the context in the meantime. Since the system build will incorporate other changes as well as yours, as Figure 8-1 shows, it is the true test of whether your code integrates with the current state of the work. But this also makes it harder to isolate the source of problems.

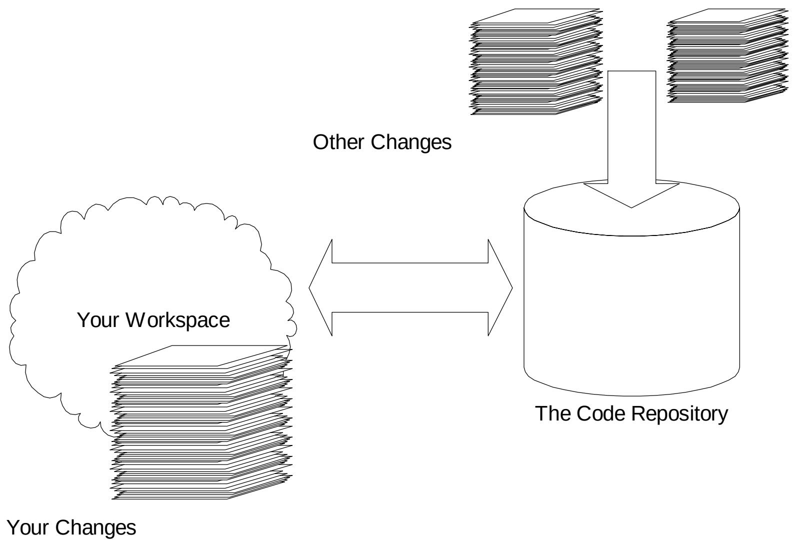


Figure 8-1 .The Build integrates changes from everyone.

You may have times when your pre-check in build works just fine, but the nightly build fails. Or your copy of the system that you get fresh from source control works just fine, but the product install that is made from the nightly build does not work the way that you expect. You could always start debugging from a product install, analyzing logs and other run time debugging facilities, but debugging from your development environment gives you more information. Sometimes the problem in this case is that the product build and install did not incorporate a new file or resource that you added to source control. Having release engineering maintain a list of what gets built and installed adds a sense of reliability and reproducibility to the build, but if developers are the ones adding components to the version tree, and they don't have visibility or control over what gets on the list, then changing the build involves an added layer of communication, which implies an additional chance for error.

Often organizations have very well established formal build procedures, but they don't scale down to the developers. Separate developer and release builds can make things simpler for some developers, but it also means that significant problems can't be found until the system build, which can be as infrequent as daily. This wastes time for anyone who needs a working codeline, and makes it harder to get product release candidates to the testers.

To be able to do a reasonable test of the effect of the changes, you must be able to build all parts of the system that your code has an effect on. This means building components in your own workspace. You can work by patching your workspace with your built objects, for example, by building only the components that you changed, and altering the system PATH or CLASSPATH to use the new components first. But software systems are complicated, and you may not see interactions that a "normal" build and execution process will have. Maintaining two procedures in parallel is difficult and error prone.

The Two True Ways

I was at a small company a while ago we had a fairly comprehensive set of build scripts and local build procedures. A developer could build the system and run tests on the local version by simply typing a few simple commands. This placed this company well ahead of organizations where there was no easy way to reproduce the effects of the nightly build. There were a few differences between the developer process and the release process that caused us grief, however.

We were using CVS which lets you define aliases for groups of directories, called modules. When you check out the files for a workspace, you check out one or more modules. Modules can depend on other modules, so checking out a module called "all" for example, will get you the entire source tree easily. I added a new directory to the CVS “all” module that all the developers used to create the workspace. The “Create a new workspace” script and build worked just fine.

The nightly build failed mysteriously. After much handwringing, and some miscommunication, we discovered that the nightly build scripts didn’t use the “all” module that the developers used, but it checked out each module one at a time. No one, aside from the release engineer, knew about this, and there was no semi-automatic mechanism to know that a change in a module required a change in the build script. We fixed this build error by sending an email to the release engineer, but that isn’t an approach that scales well.

At another places, the installer that release engineering generated from the nightly build did not include the same versions of some components that developers were using. Debugging why an installed version didn’t work, and subsequently fixing it, led to a culture of blame, and development feeling that since they had little control over the process of creating install kits, that solving the problem was out of their hands. By viewing creating an install kit as a process that was somewhat separate from building the software, getting a release out the door got harder.

Think Globally by Building Locally

Before making a submission to source control, build the system using a Private System Build that is similar to the nightly build.

The private system build should have the following attributes:

- Be like the *INTEGRATION BUILD (9)* and product builds as much as possible, though some details that are related to release and packaging can be omitted. It should at least use the same compiler, versions of external components, and directory structure.
- Include all dependencies.
- Include all of the components that are dependent on the change. (For example, various application executables.)

The architecture will help you determine what a sufficient set of components to build is. An architecture that exhibits good encapsulation will make it easier to do this build with confidence. Figure 8-2 illustrates what goes into a build.

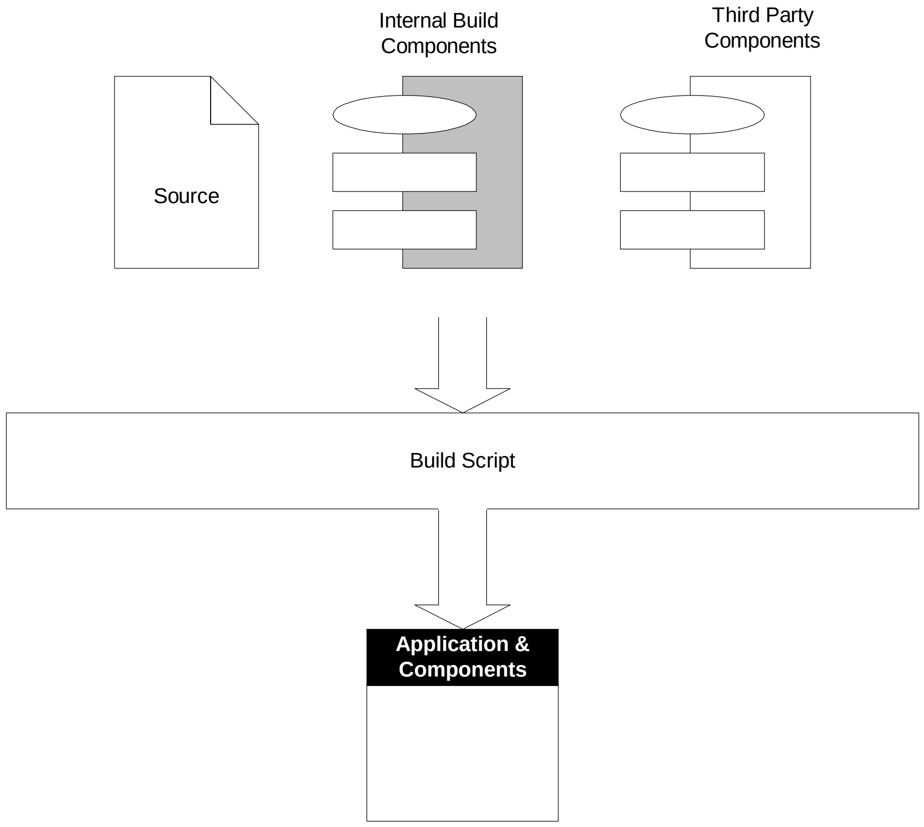


Figure 8-2 .Components of the private system build.

The build should not differ significantly from the nightly build. Wingerd and Seiwald suggest that “developers, test engineers, and release engineers should all use the same build tools,” to avoid wasting time because you are not able to reproduce a problem (Wingerd and Seiwald 1998). In *The Pragmatic Programmer: From Journeyman to Master* (Andy Hunt and Thomas 2002b) Andy Hunt and Dave Thomas say “If you do nothing else, make sure that every developer on a project compiles his or her software the same way, using the same tools, against the same set of dependencies. Make sure that the compilation process is automated.”

If it must, it can differ from the product build in the following ways:

- It can be done in an IDE or other development environment, as long as you know that the compiler is compatible with the one used in the product build process. Beware of differences that cause inconsistencies. And make a integration build script available to enable debugging.
- It can skip steps that insert identifying information into the final product, for example, updating version resources. Even steps like this can cause problems, so it is best to include these steps as well, and just not check in changes that happen because of the build process. For example, if your changes did not change a version resource, do not commit the automatic change to the resource.
- It can skip some packaging steps, such as building installation packages, unless this is what the developer is trying to test.

It is important that the private build mechanism replicate the production build mechanism semantically as much as possible, while still being usable by a single developer.

Examine how you decide what gets put into the build, and what gets put into each component, such as a jar file or a library. Some common approaches are:

- Build everything in version control. This has the advantage of making it easy to decide what to build and include. The disadvantage is that it can discourage developers from using the version control area to share files that may not yet be ready for use, since if they do not build they can create errors. You still need to how to package files into deliverable units.
- Build everything except for items marked to exclude. This allows you to put anything in source control so that it gets included by default, but allows the option of excluding certain files.
- Build only parts of the version control tree that are explicitly included in a release list. The parts can be parts of a directory structure, or individual files. The advantage of this approach is that it is makes it easier to put experimental code into version control without worrying about breaking the build.

Either approach can work well. The build everything approach is simplest. The include/exclude approach can work well if the include/exclude list is maintained in version control and can be built by developers, since the developers drive what code and components get into the product. As long as developers and the nightly build the same files and “install” them in the same places, you will be able to debug problems easily. Any approach that has two different systems will cause conflict and delays.

When rebuilding in your workspace you need to decide whether to do a full build or an incremental build. A full build is best to ensure that you are not missing any dependencies. But a full build may be impractical for active development, since it can take a long time. So under most circumstances, you can do an incremental build if your dependencies are set up correctly. You will want to do a clean build, when:

- You are adding new files to the source control system. In this case you also want to start with an *empty* workspace. This is the only way to check that you added the file to the correct place. It's not unknown for someone to forget to check in a file, but for their builds and tests to pass because the file was in their workspace.
- You make extensive changes involving key functionality. This may be over-cautious, but is best to do if you have any doubts that your dependency checking is in error.

You can also do a "clean" build of an individual component (for example, a library file, or a jar).

They key thing to remember is that you want to do this process repeatedly. Requiring a clean build all of the time will make the process too slow to be useful, but never doing a clean build will expose you to any flaws in the way your tools handle dependency checking. An Integration Build should catch any problems of this sort, but the earlier that you catch the problem, the less expensive it is to fix.When in doubt, do a clean build if time permits.

Once you make sure that your code works in your current environment. Then update your workspace with the latest versions of code from the Codeline that you are working on, and repeat the build and test procedure.

A *PRIVATE SYSTEM BUILD* does take time, but this is time spent by only one person rather than each member of the team should there be a problem. If building the entire system is prohibitive, build the smallest number of components that your changes effect.

If you changes a component that other components depend upon as an interface it can become very difficult to cover every case. Ideally you would build all clients. In this situation, let the *SMOKE TEST (13)* determine what executables to build.

As in many aspects of software development, this is a situation where communication is very helpful. If you think that you will be making a sweeping change, run all of the tests, and then announce the pending change widely. This will allow people to let you know about their dependencies, and also allow them to identify your change as a potential conflict if they see a problem later.

Related to the question of clean versus incremental build is the question of “what” to build. Start with whatever you need to run smoke tests. If you are developing a component that is used by one or more applications, consider building one or more of these executable applications. Ideally you would build all the applications that you know about. You don’t need to be exhaustive though. If you miss something, your integration build and related testing will find it. This approach is not “passing the buck;” while each team member needs to attend to quality, everyone cannot take an infinite amount of time to do this. Consider the time in the release cycle, the reliability of your incremental build tools, and the time that it takes for a full build.

Regardless of what approach you take on a daily basis, it should be possible for you to start from an empty workspace and recreate the products of the nightly build when necessary.

Unresolved Issues

Once you know that you can build the system, you still need to know if you are not breaking the functionality.To make sure that the system still works, do a *SMOKE TEST (13)*. This pattern enables you to do a smoke test.

If the system is very large, it may not be efficient to build every component that use your comoponents.These left over dependencies will get validated in an *INTEGRATION BUILD (9)*.

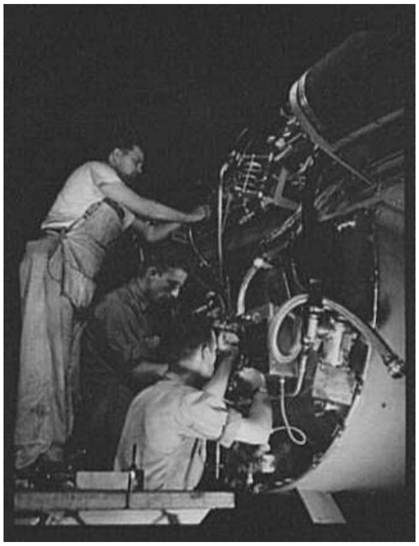
What do you do when you find a build error in some other code that is related to your changes? Ideally you should merge your own changes if you can identify them, and they are not too extensive. If you change a widely used interface, and the team has agreed to the change before hand, it may make sense to communicate the timing of your change so that other team members can change code that they are responsible for themselves. Your team dynamics will best decide the answer to this question.

Further Reading

- Steve McConnell discusses the need to do a build before checking in code in *Rapid Development*(McConnell 1996).

- *The Pragmatic Programmer: From Journeyman to Master* (Andy Hunt and Thomas 2002b) Andy Hunt and Dave Thomas has much good advice about build automation.

Chapter 9
Integration Build



Each developer is working in their own *PRIVATE WORKSPACE (6)* so that he can control when he sees other changes. This helps individual developers make progress, but people are making independent changes in many workspaces that must integrate together, and the whole system must build reliably. This pattern addresses mechanisms for helping to ensure that the code for a system always builds.

◈ ◈ ◈

How do you make sure that the code base always builds reliably?

Since many people are making changes, it really isn’t possible for a lone developer to be 100% sure that the entire system will build after they integrate their changes into the mainline. Someone can be making a change in parallel with you that is incompatible with your change. Communication can help you to avoid these situations, but problems still happen.

When you check in code changes it is possible that, despite your best intentions, that you may introduce build errors. You may not need to build the entire code base before a check in; there may be components that you don’t know about, or that are not part of another team’s work. Your build environment may be inconsistent with the “release” build environment at any point in time. For example, if you work on a PC or Workstation, you may be slightly out of date with respect to the standard, in terms of compiler or operating system version, or version for a third party component. Or you may be trying out a new version of component that seems to be compatible with the version that everyone else is using. Duplicating your effort on multiple systems when the risks of a problem are small seems wasteful.

The best that you can do is to try to build everything. A complete, clean build may take up more time than you, or any other developer, can really afford to spend. On the other hand, the time that it takes for one person to do a complete may be small compared to the time that the team takes to resolve the problem; if you break the build, it will slow other people down. It is better to localize the problem as soon as you can. As Figure 9-1 shows, integration can be tricky, akin to putting puzzle pieces together.

A complete, centralized build may address some of these problems, but a centralized build will work off checked in code, so the damage is already done.

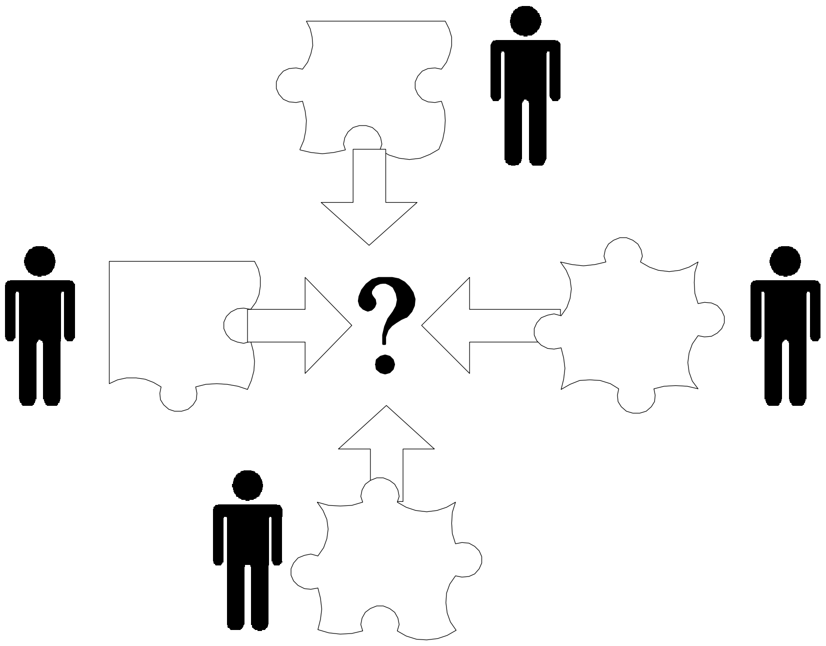


Figure 9-1 .Integration can be Difficult.

Some users of the system may not want, need, or be able to build the entire code base. If they are developing software that simply builds on top of another component then they worrying about integration build issues will be a waste of their energy. They really want a snapshot of the system that they know builds.

Tracking down inconsistent change sets is frustrating work for other developers, so the smoother the build, the higher morale. You need a way to ensure that these inconsistencies are caught as quickly as possible, in an automated, centralized manner.

De-integration Build

At a couple of start-ups I worked at, "builds" were done by individual developers and then "released" for testing. This caused a number of problems when the organization grew; different developers were experimenting with different versions of third party components, and compliers, which is reasonable if you want to minimize risk by exploring alternatives. But then when the build did not work, or the built system didn't run, it was hard to figure out who was responsible, since there was no one standard configuration or set of configurations that each developer would run on their machines.

The fix was not to standardize development machines totally, since that prevented experimentation. A build machine, and a nightly build fixed the problems.

Do a Centralized Build

Be sure that all changes (and their dependencies) are built using a central integration build process.

This build process should be:

- Reproducible
- As close as possible to the final product build. Minor items, such as how files are version labeled might vary, but it is best if the Integration Build is the same as the Product build. At the end of the integration build, you should have a candidate for testing.
- Automated, or requiring minimal intervention to work. The harder a build is to run, the more even the best-intentioned teams will skip the process occasionally. If your source control system supports triggers, you could have the build run on every check-in.
- A notification or logging mechanism to identify errors and inconsistencies. The sooner that build errors are identified, the sooner they can be fixed.

Also, rapid notification makes it easier to track the change that broke the build.

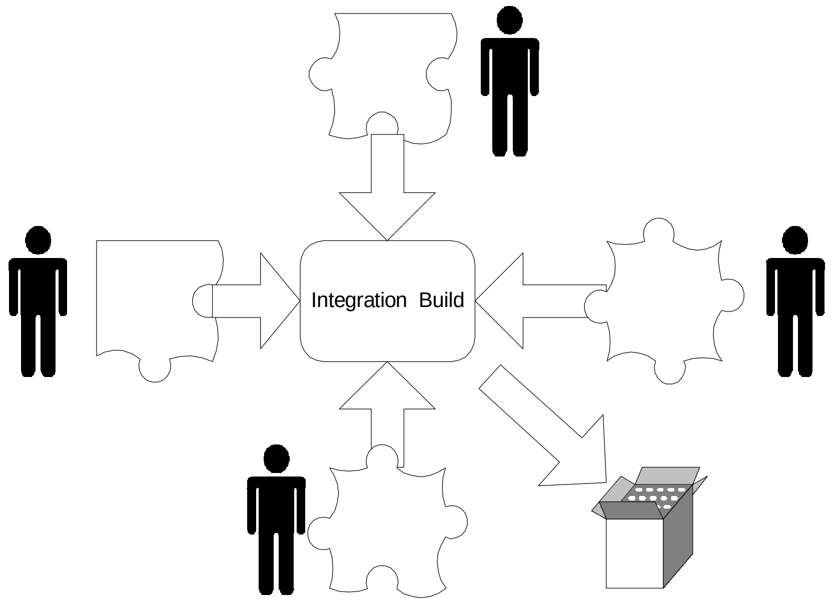


Figure 9-2 .An Integration Build Process Assembles the Pieces

Perform the build in a workspace that contains the components being integrated. Determine how often to run the integration build based on the following factors:

- How long it takes to build the system
- How quickly changes are happening

If the system takes a long time to build, or if the product is fairly static, consider at least a staged daily build, with an option to run additional builds as needed.

If the system can be built fairly quickly, consider running the build on every submission (check in) to source control. While this may seem resource intensive, it will make it very easy to determine the sequence of changes that broke the build.The trade-off is that if your version control system does not serialize changes adequately, you may have build failures simply due to inconsistencies.

Identify this build with a label in your version control system.

The integration build should be repeated on all supported platforms when the system supports them. Having individual developers do multiple builds can be a time sink.

Remember, the intent of the integration build is to catch build issues that fall through the cracks. Only if the builds fail consistently for the same reason should you add additional pre-checkin verification steps.

If appropriate, use the integration build as the basis for an install kit.

You check in a change to the repository. The source control system responds to the check in by extracting all of the files for the system, and it builds the resulting system. Errors in the build get reported to the build master as well as the person who submitted the change.

Unresolved Issues

Even if the system builds it may still not work. Follow up the Integration Build with a *SMOKE TEST (13)* to ensure that the integration build is usable. If this build is to be published as a named stable baseline, also do a *REGRESSION TEST (15)*

Further Reading

- *Rapid Development* (McConnell 1996)describes a Daily Build and Smoke Test.
- The Daily Build and Smoke Test Pattern first appeared in Coplien's pattern language(Coplien 1995).

Chapter 10

Third Party Codeline

You want to focus on building the components for which you can add the most value, and not on basic functionality that you can easily buy.Your codeline is associated with a set of external components that you will ship with your product.You may customize some of these to fit your needs. You need to associate versions of these component with your product. When you create your *PRIVATE WORKSPACE (6)*, or when you build a release for distribution, you need to associate these components with the version you are checking out. You also want your *REPOSITORY (7)* to contain the complete set of components that comprise your system. This pattern shows how to track the third party components in the same way that you track your own code.

◈ ◈ ◈

What is the most effective strategy to coordinate versions of vendor code with versions of product code?

Using components developed by someone else means letting go of control of both the implementation and the release cycle for what could be a key building block of your system. The essence of source control and release management is the identification of what components go together to reconstruct a given version of a product.You still need to be able to reconstruct old builds for debugging and support purposes, so you need a way to track which vendor release goes with what version of your code. If this was all your own code, you could simply label it in the version control system at the time that you made a release. But vendor release cycle are different than your release cycles, as Figure 10-1 shows.

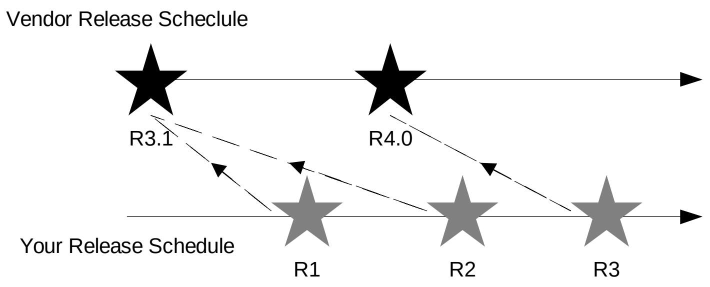


Figure 10-1 .Vendor Releases andYour Releases are not in Sync.

You could have a manifest or list that listed what versions of third party components went with which versions of your product. Youneed to easily identify which versions of components go with which versions of your product.

Using a list to associate a vendor release with your products version can be tricky during development. It is easy to use the third party component’s installation process if you need to are working with a static version, but in a dynamic environment you want to make installation with the correct version simple for developers.This can get complicated during development as well, as you need to coordinate third party code versions with your code when you build a workspace, and you want to be able to build and update a workspace automatically. Consulting a list, and installing the right versions of components adds an element of risk.

When you decided to use third party code, it was because you wanted to save yourself work, and because the third party added more value in a specific domain.There are times when the outside code is not perfect and needs changes and adaptations to work. You could be using publicly available code, and need to customize it to fit your particular needs, or you might need to fix a bug in vendor code if you have access to the source. If you make custom changes to the vendor-provided code, you need to provide a way to integrate these changes into your subsequent releases until the vendor makes a release with the changes you need. In some cases your changes may never be in the vendor release, and you need to re-apply them to subsequent vendor releases.

Even with binary only components that you do not make changes to, you still need to associate releases of the outside code with releases of your product.

Using third party code is, by its nature, risky, but treating third party code as 'outside' your system is risky if your system depends on it.

A Few Stitches, too Late

I've seen variations on the following scenario at a number of companies that I have worked. A company has a centralized build process that works fairly well. They use a number of third party packages, and they handle version issues by using install kits. When a developer needs to work on an older release, use the install kits. This works fine until they find a need to patch the third party code. Then it is tricky to describe what custom version of the code goes with a point in the development cycle.

Another frequent issue is having difficulties debugging an old release that get tracked down to the wrong version of a particular third party component component.

When people do their development against shared server systems, the problem is made worse since there is a belief that only the server need be updated. This does not reflect the reality that you may ned to fix a bug on an old version.

The problems have almost always been the result of a desire to avoid extra work to track the third party code. As is often true, the short term “savings” are more than lost in the long run.

Use the tools you already have

Create a codeline for third party code. Build workspaces and installation kits from this codeline.

Use your version control system to archive both the versions of the software you receive from the vendor, as well as the versions you deliver to your customer. Use the branching facility of the VCS to track separate but parallel branches of development for the vendors code, and your customized versions of the vendors code. When you get vendor code, make it the next version in the vendor branch and then merge the code from that branch into your customized branch. Your version control system should maintain enough information to build any version of your product using the correct versions of all components, internal and external.

What is “third party code?” Third party code is any code supplied by someone outside your organization, or fixes and enhancements to that code. “Plug-ins” and extensions to a third party framework are not third-party code and should be treated as product code. For example, if you have a library for parsing XML, that is third party code. If you find that this version of the parser does not parse certain XML correctly, the fix for that should be made to the third-party codeline. Parsing event handlers that you write to use in your application are not part of the third party codeline, though they certainly depend on the third-party code.

To accept vendor code, do the following:

- Add the vendor code to the appropriate directory of a vendor codeline. Add it to this codeline in exactly the way it unpacks from the distribution medium. If the component is something that you can build, you should be able to build it from the checkout area.
- Label the checkin point with a label identifying the product and version.
- Immediately branch this new codeline. All the projects that will use this version of the vendor code will use the code off of the branch, making it possible to customize the code when you have source available. If you need

to build the components locally, build them on this branch and check in the derived objects.
- Check in derived objects here to save time and effort; they should not change frequently.
- When a new vendor release appears, add it to the mainline portion of vendor codeline. Branch again, and merge any relevant changes from the prior branch into the new branch.

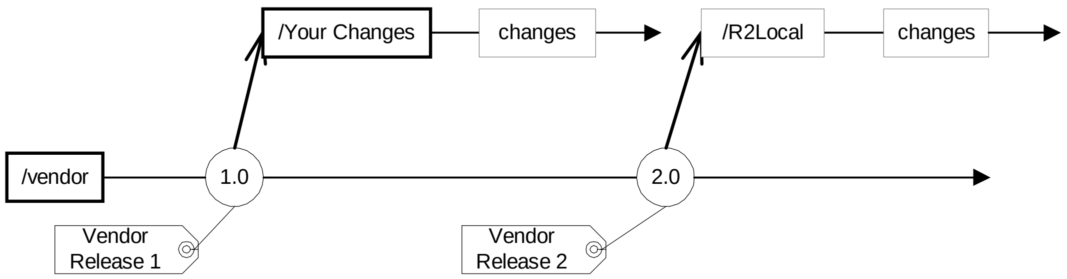


Figure 10-2 .Third Party Codeline

Figure 10-2 shows the resulting codeline.

You can now easily reproduce prior versions of your own releases as well as those of the vendor. Customization differences can easily by isolated and reproduced so you can see what you had to change for a given release. Differences between vendor releases can easily be isolated and reproduced to see what the vendor changed from release to release By tracking customization changes on a separate branch from vendor changes, you are basically applying a divide and conquer approach of orthogonalization: instead of one big change, you logically partition it into vendor changes and custom changes from a common base version. This reduces merge complexity. The resulting project version tree reflects the real-world development path relationships between the vendor and your group. This requires more storage space than simply keeping one source tree or one set (branch) of versions of the source tree. Requires the oft-despised merging of parallel changes. There are many who feel that “merging is evil!” However, in this case, you are not the one who controls the development of the code. You are at the mercy of the 3rd party supplier for this. The best you can hope for is that they incorporate *all* of your changes into their code-base. Thus, merging is really unavoidable here.

When you release a version of your product, label the code on the third party codeline that your product has been built and tested against with the same label as the product release.

Even if you make no changes to the vendor code, the release history is now traceable, and the vendor releases can get labelled with the appropriate product releases. Even if all you have access to are interface files (header files) and derived objects (libraries, jar files, etc.), track these using version control as well, even though you will not have the same amount of delta information available. If you do make changes you should check in ‘compiled’ versions of the product that include the changes. When you get a new vendor release, you can compare the source code in the various branches and consider doing a merge if your changes are not in the later release.

To create a developer workspace, make sure that you check out the third party components are part of the product check in. If your version control system supports the concept of sets of related parts of the source tree (i.e., modules), when you check out a given point in time of your product, you get the appropriate version of the third party component for free. If your version control system allows you to re-arrange the locations of objects during a check in, check binary objects into the appropriate common ‘bin’ directory. Otherwise, be sure to alter PATHs and CLASSPATHs appropriately so that build and runtime environments point to the correct version.

Include the appropriate third party product branches when labelling the release. For some components you may have licensing constraints that say that you must use the vendor installer. In this case, you still have tracability built into your version control system, and you know that what you are shipping matches what you are developing with. You may also be able to simply integrate the third party code into your own installation process by placing binary objects in the same place that your product specific code lives.

To reproduce a prior build, including the correct third party code check out the appropriate label into a new workspace, and have the correct versions of all components. For some software component systems, such as COM, you will need to deal with the issues about system-wide registration of specific component versions.

If you have customized versions of build tools, for example, gcc, or if your product is dependent on a particular version of a particular tool, you can handle it by thinking about what you need to ship. If you need to ship all or part of the tool as a run-time component, use this approach. If the tool is used only at build time, you can still track it in your system using a third party codeline, but dependencies between the tool version and the product version can be handled by flags and identifiers in Makefiles, for example.

The procedures here apply to any run-time component, for example language extensions to languages such as Perl, Python, or Tcl. The specific version of the interpreter environment is another story.

When you are using a dynamically loaded third party component that is a shared resource that other products may use, you have to decide how to install it if it already exists on the target system. The options are to upgrade existing installations, require that your version be the correct one, or, if the component technology supports this, install the version you expect on the target system in addition to any existing versions. This may be tricky, for example, in the case of a COM component, where the vendor has not followed the appropriate version conventions. Since only one version of a COM component can be the latest, it may be impossible to have more than one installed. In other cases, you can install multiple versions by altering the PATH or CLASSPATH environment when you load your system. This is not so much a technical issue as a support and positioning issue. Multiple copies means more space, but running with an unknown version of a third party component makes verification and testing easier.

Interpreted languages present a special case of this problem. If your system depends on a specific version of Python or Perl you can install the additional version of the interpreter in a ‘special’ path, you can overwrite an existing installation, affecting all users of the product. Some of these tools allow you to build an executable that has an embedded interpreter, increasing isolation at the cost of a larger executable, and less access to the source code, which eliminates a benefit of using a scripting language.

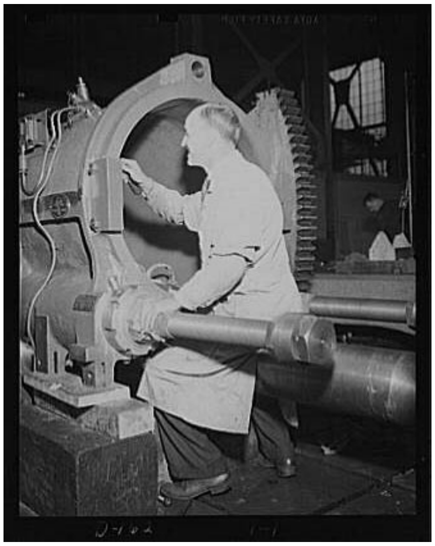
Unresolved Issues

If you are using a third party product that is very stable, or which you will never customize, you may not need to create a branch. The cost of the branch in this case is small, and it gives you the flexibility to make changes later, if you need to.

Further Reading

?????

Chapter 11
Task Level Commit



An *INTEGRATION BUILD (9)* is easier to debug if you know what went into it. This pattern discusses how to balance the needs for stability, speed, and atomicity.

◈ ◈ ◈

How much work should you do between submission to the Version Control System? How long should you wait before checking files in?

When you make changes to the code base you want to focus on the act of coding. Administrative tasks, like checking in changes, pre-check in testing, etc. are a distraction. Coding is a sequence of changes, bug fixes, and enhancements, and you need to track these changes.The revision history in your version control tool should reflect the way that file changes map to functionality changes.

To add one feature, or to fix one defect, you may need to make changes across many parts of the code base. But every change introduces a potential instability in the mainline. You would like to be able to rollback or remove a change if it causes unexpected problems. For this to work changes need to be consistent and complete.

Depending on your *CODELINE POLICY (12)* a check in may involve a long sequence of steps, including testing, that will take up time. You may want to avoid this overhead and delay checking in a change for as long as possible. In the case of multiple feature changes that require changes to more than one module, it is certainly easier to do all of the changes to that module at once. But the longer that you are working on your own copy of the module, the more likely you may be in conflict with other people's changes, and the harder it will be to roll back a particular change.

When an integration build breaks you want to be able to track down the change the broke it. A long list of changes in an integration build report is more work to process, but a more detailed change history makes it possible to selectively remove changes that may have broken the build.

It can be tricky to decide what a task oriented change really is. Sometimes the definition of a task is natural. You may be fixing a problem that was assigned an issue number. If the issue can be fixed by changing one file, you have a natural atomic check in. If a change spans multiple components across various systems, it is not obvious what changes go together unless they are checked in. If you err on the side of smaller units of work per check in, you gain the overhead of processing the check in. If you err on the side of larger grained units of work, you lose the ability to back out small changes.

You want to be able to maintain a stable codeline, and associate changes with features or defects that were fixed.

Coarse Grained Tasks

We were working in an organization that had a very rigorous pre-check in validation process. As a result, developers would check in code at most once a day, occasionally less often. Because of this, each check in would often mean more complicated merges (in the worst case) or failures that were hard to track down (better case). While the motivation was a reluctance to check things in, the deeper problem was that each check in covered multiple tasks, so it was hard to say what the real source of a problem was, and harder to roll back changes.

After enough of these situations, people started doing smaller grained check-ins.

Do One Commit per small-grained task

Do one commit per small grained, consistent task.

Strive to have a each commit of code changes reflect one task. This will allow you to do fixes in the sequence that allows you to make the most important changes first.The unit of work can be a new feature (or part), and problem report, or a refactoring task.

It is OK to batch changes if it really makes sense and batching does not add significantly to the time that you are working on a local copy of the code. Consider the complexity of the task, how many files, or components you need to change to implement the task, and how significant of an effect the change will have on the system (how risky it is). Each change should be represent a consistent state of the system.

Example of reasonable change tasks are:

- A problem report (but if the problem is a broad problem, it may have two or more check ins associated with it.)
- Changing calls to a deprecated method to use a new API for an entire system.
- Changing calls to a deprecated method for a coherent part of the system.
- A consistent set of changes that you accomplished in a day.

When in doubt, error on the side of more check ins because it is easier to roll back changes, and also see the effects of integration with other people's work. Also, since your revision control system is an indicator of the 'pulse' of the development work, strive to check in changes at least once a day, if it makes sense.

Sometimes it will make sense to have a check in for multiple defect reports or features. A degenerate example of this is a one or two line section of code that was responsible for numerous problem reports. In this circumstance multiple check ins would not really be possible. But whenever you are changing separate parts of the code base, do one check in per feature or defect report.

A pre check in policy that is too vigorous for an can discourage this practice, so consider ways to streamline the pre-check in validation to test only what is necessary. Extended code freezes make it difficult to maintain this practice. Extended code freezes are bad for many reasons, as they interrupt the flow of work.

Before you check in, be sure to catch up your workspace to the current state of the codeline and test for compatibilities.

Unresolved Issues

Some changes are far reaching, and inherently disruptive and long lived. In this case consider using a *TASK BRANCH (19)*. If there are multiple people working on a task branch, then perform small grained commits on the task branch.

Unit Tests, Smoke Tests, and Regression Tests, as well as the Edit Policy provide guidance on how encourage small grained checkins.

Good integration between the development environment and the version control system will make the commit process fit in better with the flow of the developer's work.

Chapter 12
Codeline Policy

When you have multiple codelines, developers need to know how to treat each one. A *RELEASE LINE (17)* might have strict rules for how and when to check things in, but an *ACTIVE DEVELOPMENT LINE (5)* might have less strict rules. This pattern describes how to establish the rules for each codeline to suit its purpose.

◈ ◈ ◈

How do the developers know which codeline to check their code into, and when to when to check it in, and what tests to run before check in?

Each codeline has a different purpose; one codeline might be intended for fixing bugs in a particular release; another codeline might be used for porting existing code to another platform, yet another is for day to day development. These codelines have different requirements for stability. If code is checked into a codeline ignoring the rules productivity will suffer. A developer needs to know which codeline they should be using, and what the policies are on that codeline.

You can identify different codelines by their names. A codeline's name can tell you something about its purpose, but it can't express all the finer points of codeline usage such as the policies. For example, a “release line” can be very restricted, or only slightly slow, depending on the organization’s strategy and vision. And it can be hard to come up with good, unambiguous naming conventions. Figure 12-1 shows how we might diagram the association between a codeline and a policy.

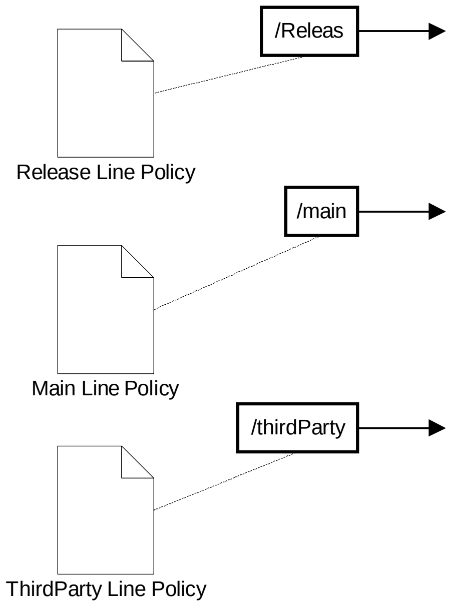


Figure 12-1 .Each Code Line needs different rules.

Once you decide on how stable a codeline needs to be, and how to realize that level of stability through processes, you need to inform developers of these policies and then enforce them.You can provide formal documentation describing the finer points of codeline usage, but this requires extra effort for documentation and maintenance; once the documentation gets out of synch with the policy, it will be useless. Also developers may perceive the formality of such a document as overbearing, or as if it were some draconian tactic to interfere with “real work.” It is easier to have people use a policy that they understand and believe in than one that seems arbitrary (Karten 1994).

You want developers to behave properly and follow the policy, but even well intentioned people forget things in the heat of a deadline. You can use peer pressure or punishments to enforce policies, but that can break down the team.

You can use automation to enforce policies, but it is hard to implement automated procedures correctly. Sometimes some steps in a process do not make sense, and it is hard to encode policies that allow you to skip a step. If you do provide a mechanism for sidetracking parts of the process, developers may ignore the process, out of a well intentioned sense of experience, if nothing else.

For a codeline, different roles might have different degrees of leeway with executing the processes. For example, you may want to forbid permanently deleting a file's history on a release line, but there are times when it makes sense, such as when someone checks in a new file accidentally. Someone needs to be able to fix this, or you will end up with useless files in your codeline. If you are too restrictive with permissions, you need to be able to give people permissions they need quickly. For example, if only the Director of Development can perform certain operations, but the Director also travels much, you may be stuck.You may not be able to have an open codeline in all cases without having a free for all.

Impedance Mismatch

I worked at a software company that developed a number of products that were based on some common components. We were doing major work on one of the components, so there would be points in time when interfaces were evolving, or there might even be some bugs or inconsistencies. We were told that this was an evolving codeline and to develop accordingly. The other clients of the component were upset at having to adapt to changes, even though they were staged with plenty of warning.

The problem here was that the codeline policy didn't mesh with the needs of all of the users. There were a number of possible ways to address the problem. A stricter policy would have slowed progress. We could also have treated the other teams as external customers, providing them with 'released' versions of the library.

Define the Rules of the Road

For each branch or codeline, formulate a policy that determines how and when developers should make changes. The policy should be concise and auditable.

The codeline policy explicitly states the rudimentary policies an organization has about how to conduct concurrent development and how to manage releases. Vance says that "a codeline policy defines the rules governing the use of a codeline or branch" (Vance 1998). In addition to using naming conventions and meaningful codeline names, formulate a coherent purpose for each codeline. Describe the purpose in a clear and concise policy. The policy should be brief, and should spell out the "rules of the road" for the codeline, including:

- The kind of work encapsulated by the codeline, such as development, maintenance, a specific release, function, or subsystem;
- How and when elements should be checked-in, checked-out, branched and merged;
- Access restrictions for various individuals, roles, and groups;
- Import/export relationships: the names of those codelines it expects to receive changes from, and those codelines it needs to propagate changes to;
- The duration of work or conditions for retiring the codeline;
- The expected activity-load and frequency of integration

Make the policy short and to the point: a good rule of thumb is 1-3 paragraphs, with one page as an absolute maximum.

Keep in mind that not all codeline policies will require all of the above information. Only specify what is essential. Some VC tools allow you to associate a comment with each branch and codeline name. This is an ideal place to store the description of a suitably brief codeline policy. Developers can run a branch-description for the codeline, instead of digging around for its documentation. Otherwise, store the codeline policy in a well known, readily accessible place. You could perhaps provide a simple command or macro that will quickly display the policy for a given codeline name.

You should create a branch whenever you have an incompatible policy.

Some example of policies for include:

- Development codeline: interim code changes may be checked in; affected components must be buildable. (Wingerd and Seiwald 1998)
- Release codeline: software must build and pass regression tests before check-in; check-ins limited to bug fixes; no new features or functionality

may be checked in; after check-in, branch is frozen until entire QA cycle is completed.(Wingerd and Seiwald 1998)

- Mainline: all components must compile and link, and pass regression tests; completed, tested new features may be checked in. (Wingerd and Seiwald 1998)

Enforce parts of the policy by using any mechanism that your version control tool supports, such as triggers. If automatic enforcement becomes too constraining, use automation to report on adherence to the policy.

Unresolved Issues

To enforce a codeline policy effectively you need to balance the utility of using automation and of the group's culture. You should look into your tools to see what mechanisms they provide. You can also consider using a tool such as ANT, and write ANT tasks that enforce and/or audit your policies.

Further Reading

- You can enforce codeline policies with the mechanisms in various tools. CVS, Perforce, and other tools support triggers that run before or after check in. Tools such as ANT allow you to codify many build-related activities as tasks

Chapter 13

Smoke Test

An *INTEGRATION BUILD (9)* or a *PRIVATE SYSTEM BUILD (8)* are useful for verifying build-time integration issues.But even if the code builds, you still need to check for runtime issues that can cause you grief later. This verification is essential if you want to maintain a *ACTIVE DEVELOPMENT LINE (5)*. This pattern addresses the decisions you need to make to validate a build.

❖ ❖ ❖

How do you know that the system will still work after you make a change?

You hope that you tested the code adequately before checking it in. The best way to do this is to run tests after every build and before you check something in to source control, but you need to decide which tests to run.

You can write tests that target the most critical or failure prone parts of the code, but it is hard to develop complete tests.

You can play it safe and test everything that you can think of, but it is time consuming to run exhaustive tests. and your progress can be very slow. Also if the test takes too long, you may lose focus and have a harder time correlating your recent changes with any problems. Long running tests encourage larger grained changes between testing. Unstructured and impromptu testing will help you to discover new problems, but it may not have much of an effective yield.

Running detailed tests is time consuming, but if you check in a change that breaks the system, you waste everyone’s time. Rapid development and small grained checkins means that you want the cost of pre-checkin verification to be small.

The Right Balance

Smoke tests are important on many levels. At one place I worked, release candidates were built periodically, and the first developer to try them got the pleasure of finding (and sometimes fixing) all of the bugs. This led to people being reluctant to be the first to use the new builds, and led to a fairly strong culture of blame.

At another place, the pre-checkin test process was so exhaustive that developers feared it, and they did as few checkins as possible, making many changes per checkin., thus not isolating changes. This had a negative effect on productivity. Also, it was very likely that someone would check in a conflicting change in the 60 minutes that the test ran. Running the long tests reduced productity and quality.

The lessons here are that smoke tests are an essential pre-checkin step, and that more testing (pre-checkin) is not always better.

Verify Basic Functionality

Subject each build to a smoke test that verifies that the application has not broken in an obvious way.

A smoke test should be good enough to catch “show stopper” defects, but disregard trivial defects(McConnell 1996). The definition of “trivial” is up to the individual project, but you should realize that the goal of a smoke test is not the same as the goal of the overall quality assurance process.

The scope of the test need not be exhaustive. It should test basic functions, and simple integration issues. Ideally it should be automated so that there is little cost to do it. The *SMOKE TEST* should not replace deeper integration testing. A suite of unit tests can form the basis for the smoke test if nothing else is immediately available. Most importantly, these tests should be self scoring. They should return a test status and not require manual intervention to see if the test passed. An error may well involve some effort to discover the source, but discovering that there is an error should be automatic.

Running a Smoke test with each build does not remove the responsibility for a developer to test his changes before submitting them to the repository. Developers should run the smoke test should be run manually prior to committing a change. A smoke test is most useful for bug fixes, and for looking for inadvertent interactions between existing and new functionality. All code should be unit tested by the developer, and where reasonable, run through some scenarios in a system environment. A smoke test can also be run as part of the build process in concert with more through tests, when the build is to be a release candidate

When you add new basic functionality to a system, extend the smoke test to test this functionality as well. But do not put exhaustive tests that better belong in Unit Tests or Regression tests.

DAILY BUILD AND SMOKE TEST (20) (Coplien 1995) describes the role of smoke test in maintaining quality. Having a Smoke Test as part of a Daily build is key to establishing *NAMED STABLE BASES (20)*, which form the basis for workspaces.

A smoke test should be:

- Quick to run, where ‘quick’ depends on your specific situation
- Self scoring, as any automated test should be.
- Provide broad coverage across the system that you care about
- Be runnable by developers, as well as part of the quality assurance process.

The hardest part about a self scoring test is to determine input/output relationships among elements of a complex system. You don’t want the testing and scoring infrastructure to be buggy. You want the test to work with realistic data exchanged between parts of the system.

To get meaningful results from a Smoke Test you need to work off of a consistent build. A *Private System Build (8)* will let you build the system in a way that will give meaningful test results.

Canned inputs are fine as long as they are realistic enough. If your testing infrastructure is too complicated, you add risks around testing the test.

If the quality goals are such that you need to do exhaustive testing, consider using Task Branches, or have a different codeline policy. Also consider branching release lines.

A Smoke test is an end to end test, more black box than white box.

Unresolved Issues

A Smoke test does leave gaps that should be filled by a more thorough QA procedure and *Regression Test (15)* suite to do more exhaustive testing to identify degradation in behavior. Developers should also work develop a *Unit Test (14)* for every module that they need. Use a *Unit Test (14)* to verify that the module you are changing still works adequately before you check the change in.

The trade-off we need to make involves the speed of check in versus the thoroughness of the test. The longer the pre-checkin test, the longer the check in. Longer check ins my encourage developers to have larger granularity commits. This goes against an important goal of using version control.

Further Reading

- *Rapid Development* (McConnell 1996) has some good advice on various testing strategies, including the trade-offs between completeness and speed.
- *Mythical Man Month* (Frederick P. Brooks 1995) also has advice on making these trade-offs, and it is a class book that every software developers should read at some point.

Chapter 14
Unit Test

Sometimes a *SMOKE TEST (13)* is not enough to test a change in detail when you are working on a module, especially when working on new code. This pattern shows you how to test detailed changes so that you can ensure the quality of your codeline.

⟐ ⟐ ⟐

How do you test whether a module still works as it should after making a change?

Checking that a class, module or function still works after you make a change is a basic procedure that will help you maintain stability in your software development. It is also easier to understand what can go wrong at a low-level interface than it is at a system level. On the other hand, testing small scale units all of the time can become tedious.

Integration is where most of the problems become visible, but when you have the results of a failed integration test, you are still left with the question: “What broke?” Also, testing integration level functions can take longer to set up, they require many pieces of the system to be stable. You want to be able to see if any incremental change to your code broke something, so being able to run the tests as often as you like had benefits. You also want to run comprehensive tests on the item that you are changing before checkin.

Since a Smoke Test, is by its nature somewhat superficial, you want to be able to ensure that each part of a system works reasonably well. When a system test, or a smoke test, fails you want to figure out what part of the system broke. You want to be able to run quick tests in development to see the effect of a change. Additional testing layers add time. Tests that are too complex take more effort to debug than the value that they add.

We want to isolate integration issues from local changes and we want to test the contracts that each element provides locally.

Safety First

I’d worked at a number of places where testing was a bit ad-hoc. We did system tests, but never really focused on unit tests. When system tests failed, we’d run code through the debugger, and sometimes we found a problem. Other times we found that the problem was that a client violated an interface contract. It took more effort that we really needed to spend. After the XP book came out, and having been inspired by chatting with Kent Beck and Martin Fowler at OOPSLA, I took unit testing a bit more seriously.

The next project my colleague and I wrote unit tests using the CPP unit framework. It took some effort to convince them of the value, but when we started to isolate problems quickly (often to parts of the code that did not have unit tests!), my colleague became convinced. Not only that, but the unit tests made making code changes less scary

Test The Contract

Develop and Run Unit Tests.

A unit test is a test that tests fine grained elements of a component to see that they obey their contract. A good unit test has the following properties (Beck 2000):

- Automatic and self evaluating. A unit test can report a boolean result automatically. A user should not have to look at the detailed test results unless there is an error.
- Fine grained. Any significant interface method on a class should be testing using know inputs. It is not necessary to write tests to verify trivial methods like accessors and setters.To put it simply, the test tests things that might break.
- Isolated. A unit test does not interact with other tests. Otherwise one test failing may cause others to fail.
- It should test the contract.The test should be self contained so that external changes do not effect the results. Of course, if an external interface changes, you should update the test to reflect this
- Simple to run. You should be able to run a unit test by a simple command line or graphical tool. There should not be any setup involved.

You should run unit tests:

- While you are coding
- Just before checking in a change and after updating your code to the current version.

You can also run all of your unit tests when you are trying to find a problem with a smoke test, regression test, or in response to a user problem report.

Try to use a testing framework like JUnit (or cppUnit, PyUnit, and other derived frameworks). This will allow you to focus on the Unit Tests, and not distract yourself with testing infrastructure.

Unit testing is indispensable when making changes to the structure of the code that should not effect behavior, such as when you are refactoring.(Fowler 1999)

Grady Booch in *Object Solutions* suggests that during evolution you carry out unit testing of all new classes and objects, but also apply regression testing to each new complete release (Booch 1996).

I've found that if I can't come up with a good unit test for a class or set of classes, I should make sure that my design is not overly complicated, and not abstract enough.

Unresolved Issues

Writing unit tests can be tedious. You should try to use a framework like JUnit to simplify some of the tedious parts of writing test cases.

If your public interface is narrow, but you want to test other functions, you need to decide whether to open up your interface to allow for testing, or do something else. There are a number of approaches to this problem.

Further Reading

- Unit Testing is a key part of *Extreme Programming* (Beck 2000; Jeffries et al. 2000).
- To simplify your testing, try the testing framework JUnit (http://www.junit.org) for Java programming, and the related frameworks for many other languages, available at http://www.xprogramming.com.
- *The Art of Software Testing* by Glen Meyers (Myers 1979)is a classic book on testing, and has a good discussion on black box versus white box testing. The book is based on work on mainframe systems, but it is still useful.

Chapter 15

Regression Test

A *SMOKE TEST (13)* is quick but not exhaustive. For it to be effective, you need to do the hard work of exhaustive testing as well. If you want to establish release candidates, you need to be sure that the code base is robust. This pattern explains how to generate builds that are no worse than the last build.

◈ ◈ ◈

How do you ensure that existing code doesn't get worse as you make other improvements?

Software systems are complex and with each change or enhancement to a system comes the possibility of breaking something seemingly unrelated to your changes. Fixing a defect has a substantial chance of introducing another(Frederick P. Brooks

1995). Without change, you can’t make progress, but the impact of a change in hard to measure, especially in terms of how the a unit of code interacts with the rest of the system.

You can exhaustively test your system after each build. Exhaustive testing takes time, but if you don’t do this testing you waste developer, and perhaps, customer time. If everyone runs exhaustive tests all of the time, they will not be able to spend much time coding.

Even if you decide to do exhaustive system-level testing periodically on the code base, you are left with the problem of how to structure the tests. You can write tests from first principles by doing an analysis of the inputs and outputs, but the payoff for writing tests like this on a system level may be small relative to the amount of effort that they take to write.You can also make intelligent guesses about what to test, but a software system is always full of surprises, especially in the ways that it can fail.

When you solve the problem of which system tests to execute, you are still left with the problem of when to run them. Some integration tests may need resources that are not on every development machine. What is a good development environment may not always be the best test of the system. Some problems may require large data sets to reproduce, or multiple clients in a client-server system. You may not have the resources to run these sorts of tests all of the time.

When the system does break, you want to identify the point in time when something broke. If you run the tests on every build or check in to source control, you will be able to identify when something failed. But your testing may not keep up with your check in and build process if the tests take long enough.

If your exhaustive tests find a problem, and you fix it, you want to be sure that you can identify when this problem happens again, since you don’t want to waste time in known issues. A problem that happens once can happen again (this is what we mean by regression, after all). This means that we should accumulate a set of test cases as we discover problems. Especially since we may not be able to guess all of the problems ahead of time. These test cases can add up over time.

We need to be able to check for these recurring failure modes.

Two Steps Back

I worked for a small software product company that had a code base combined of newer, cleaner code, and also code that evolved. On any given day, it was not clear whether you could get an update from source control and have a working system, or whether you could would have to spend the day getting the system to a point where you could do your work. The problem was that there was no automated testing of the core APIs. People would avoid moving to a current code base in fear of wasting a day, but this eventually caused other problems.

The lack of a way to check for old problems recurring caused many easily preventable quality issues.

Test for Changes

Run regression tests on the system whenever you want to ensure stability of the codeline, such as before you release a build, or before a particularly risky change. Create the regression tests from test cases that the system has failed in the past.

Regression tests are end to end black box tests that cover actual past or anticipated failure modes. A Regression test can identify a system level failure in the code base, but may not necessarily identify what broke. When a regression test fails, debugging and unit tests may be necessary to determine what low-level component or interface broke.

Regression Tests test changes in integration behavior. They are large grained, and test for unexpected consequences of integrating software components. Unit tests can be thought through fairly easily. As you add component interactions it is harder to write tests based on ‘first principles.’

Build regression test cases out of:

- Problems that you find in the pre-release QA process
- Customer and user reported problems
- System level tests based on requirements.

As you discover problems, write a test that reproduces the problem and add that scenario to the test. Over time you will end up with a large suite of tests that cover your most likely problem areas. Each problem may involve more than one test case.You can include running all unit tests in your Regression testing, but it is better if the tests involve system input.

Regression Testing is designed to make sure that the software has not taken a step backwards (or regressed) Always run the same tests for each regression cycle. Add tests as you find more conditions or problematic items to test. Always add test cases to the regression test suite; If a problem happened once, it is likely to happen again, so remove test cases only for very well thought through reasons.

Since regression tests can take a long time to run, you don't want to run them before every check-in, or even after every build (unless resources permit). There are advantages, however to having an automated procedure to run the regression test after each change, so that you can identify the point at which the system regressed. Run the regression tests as part of the nightly build. Developers should also run a regression test before any significant sweeping change. If something breaks, you can always run the unit tests to localize the change.You also have to investigate if the unit test inputs no longer match the system. Institute a policy of automated regression testing tied to each release. (Booch 1996)

Further Reading

- Steve McConnell has a lot of information about testing of all kinds in *Code Complete* (McConnell 1993)
- *The Art of Software Testing* (Myers 1979)by Glen Meyers is a classic.

Chapter 16

Private Versions

Sometimes you want to rapidly evaluate a complex change that may break the system while maintaining an *ACTIVE DEVELOPMENT LINE (5)*. This pattern describes how to maintain local traceability without affecting global history unintentionally.

◈ ◈ ◈

How can you experiment with a complex change and benefit from the version control system without making the change public?

Some programming tasks are best done in small, retractable, steps. If you are making a complex change, you may want to checkpoint an intermediate step so that you can back out of a change easily. For example, if at there are a number of design choice points, you may want to explore one path, and be able to back out to an earlier decision point when you see a problem with the implementation, as in Figure 16-1

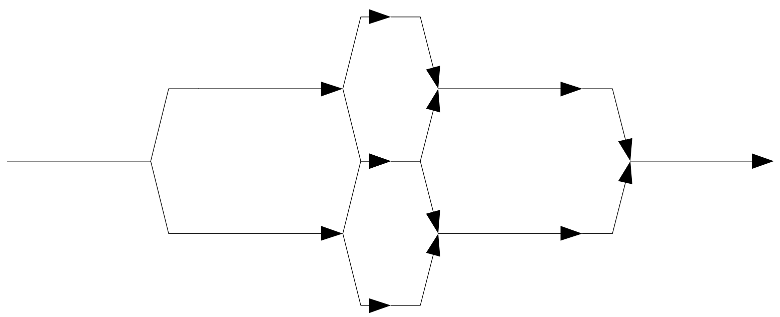


Figure 16-1 .Each decision leads to more choices, until you pick the solution.

Your version control system provides a way of checkpointing your work, and reverting to earlier states of the system. When you check something in to the active codeline, you subject every other member of your team to the changes. When you are exploring implementations, you may not want to share your choices until you have fully evaluated them. You may even decide that the change was a dead end.

If you don't want to use the version control system to track your changes, and not cause other developers delay, you will have to test your changes in accordance with your codeline policy. You may also need to integrate changes so that other code works with any API changes that you made. This can take an unjustifiably long time if you are just experimenting with options. If you decide to skip the tests, you will cause problems for other developers if you break the build, or the built system. If you skip integrating your changes with the rest of the system, you will break the codeline.This is too much work for a change that you may throw away in an hour, or a day. Figure 16-2 shows still another option; you can check in changes at and revert at each step. This generates many superfluous change events for the rest of the team, and only gives you one the ability to retrace your changes one step at a time.

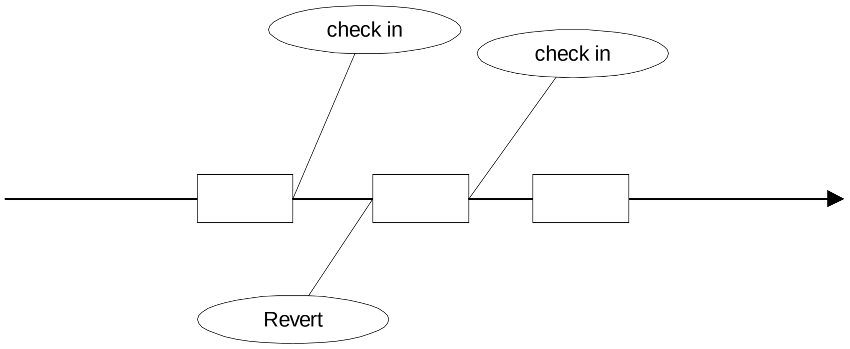


Figure 16-2 .Using the Code Line for staging generates a lot of noise.

Even if you did put the effort into validating changes before checking them in, you will be cluttering the version history with changes that are not salient to the main change path. Since the version history for a component is important, it may be important to keep the version history uncluttered with insignificant changes.

You might consider just doing all of the changes in your workspace and not checking in any intermediate work. If you don’t check in any changes, you can’t back off changes.You can take an ad-hoc approach to saving the state of your work at various points in time by copying files or using some other mechanism, but then what you are really doing is creating a minimalist version control system. It’s optimistic to expect to develop a mechanism that gives you the features that you want reliably without investing more work than is appropriate is optimistic.

You can also use private versions to test how integrating with the current state of the codeline will work with your software. You can check your changes into your private version, then catch up with the codeline, and then easily roll back if there is a problem.

The dilemma is that you want to use the tools of your trade, including version control, to create a stable, useful, codeline, but you want to do this privately.

Use Caution

Being able to version control without publishing changes is one of the things that you only miss when you don't have it. When the practice isn't established, people often avoid the issue entirely. The times that I've seen this used in practice are situations involving major refactorings, or developing proof of concepts.

Being able to rollback bad ideas before everyone sees them is very helpful.

A Private History

Provide developers with a mechanism for check pointing changes at a granularity that they are comfortable with. This can be provided for by a local revision control area, Only stable code sets are checked into the project repository

Set up a developers workspace so that they can check in changes to a non-public area when they are making an appropriate change. The mechanism should allow them to also integrate their working code base with the current state of the active development line. This private repository should use the same mechanisms as the usual version control system so that the developers do not need to learn a new meachanism.This allows developers to experiment with complex changes to the code base in small steps, without breaking the codeline if an experiment goes awry. This allow developers to make small steps in confidence, knowing that they can abandon part of a change if it takes longer than they expect.

There are many ways to implement this. One way is to have an entire *PRIVATE WORKSPACE (6)* dedicated to a task. This is appropriate when we want to experiment with a global change (for example, to an interface) and wants to evaluate the consequences of the change to see if they are manageable in the time that they have. If your change involves only a small portion of the source tree (one java package, or one directory) you can map that part of the workspace to a 'private' repository, for example, a local CVS repository, or a developer specific branch of the main repository that is not integrated with the active line. You then redirect certain check ins to the private repository. When you are done with your work, check the files into the main repository, either by specifying the repository, or copying files from the test workspace to your real one. Be sure to follow all of the procedures in your standard codeline policy before checking the code into the active codeline.

Some tools provide for promotion levels or stages. You can create private stages to use version control and not publish changes to the rest of the team.

It is important to make sure that developers using Private Versioning remember to migrate changes to the shared version control system at reasonable intervals. While one way to implement this is to provide a separate source control repository for each developer, in addition to the shared repository, this can also be implemented within the framework of the existing revision control system. If the revision control mechanism provides a means for restricting access to checked-in versions that are not yet ready for use by others, we can use the common version control system as a virtual Private Repository.

The important principle is to allow the developer to be allowed to use the version control system to checkpoint changes in an granularity which meet their needs, without any risk of the changes (which may be inconsistent) being available to anyone else.

Chapter 17

Release Line

You want to maintain an *ACTIVE DEVELOPMENT LINE (5)*. You have released versions that need maintenance and enhancements, and you want to keep the released code base stable. This pattern shows you how to isolate released versions from current development.

◈ ◈ ◈

How do you do maintence on released versions without interfering with your current development work?

Once you release a version of a product or component, it may need to evolve independently of your primary development. While it might be ideal from your perspective for your customers to simply update to new releases to get bug fixes, the reality is that there are many circumstances where you need to make a fix based on the already shipped version of the codeline. You may need to fix an urgent problem, and the new release will not be ready in time. If your application has data-migration or a complicated deployment process for it, your customers may not be willing or able to upgrade immediately. You are often faced with the problem of how to conduct development of a future release while at the same time responding in a timely manner to all the many bug reports and enhancement requests that are inevitably going to be logged against the active development.

You need to identify what code was part of the release, and what code is in the main development stream. One way to identify what is in a release is by labeling the release in the current codeline, shipping that snapshot and then continuing to work on the mainline. Figure 17-1 illustrates this approach. Doing things this way does not allow you to fix something in a fixed product independently of the mainline. While

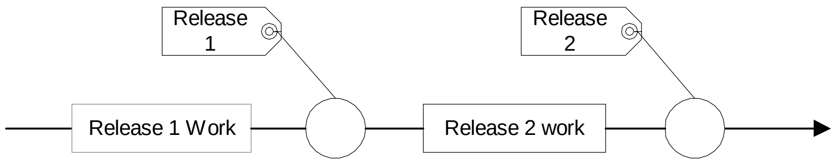


Figure 17-1 .Doing all your work on the mainline

you need to fix bugs on released products, the current development line may be evolving in a direction that is quite different that the soon to be delivered release, and it may not be easy to quickly deliver a fix.

You can create a branch when the product ships to isolate the release line from current work. Then if there are fixes that apply to both the branch and the mainline, you need to merge the changes, or duplicate the work. Figure 17-2 shows this situation.

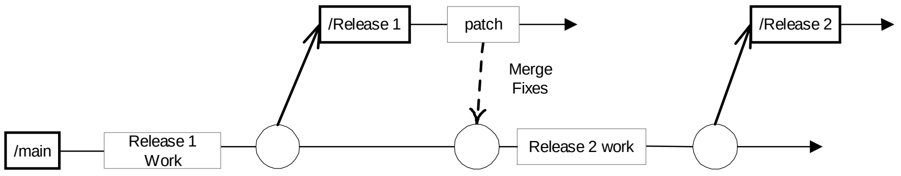


Figure 17-2 .Create a Branch when you Ship

You can put your new work on a branch, and ship the mainline. You then can merge back. This means that most developers need to merge their work; hopefully the released code won't change too much over time. You may have more than one customer, each with variations of the released software; you may need to keep track of multiple releases that are derived from other releases. You can model this by the staircase structure in Figure 17-3. This structure makes it very hard to figure out what code is common among the releases.

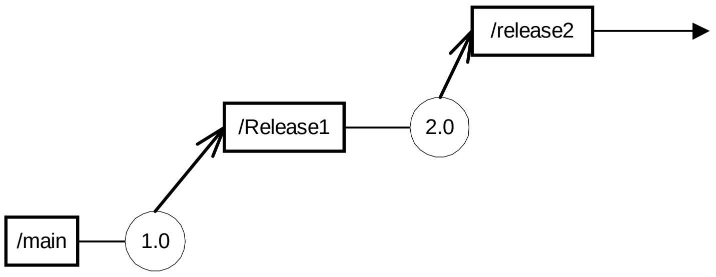


Figure 17-3 .Staircase of Dependent Branches.

You can try to keep customers only on major releases.Critical bug-fixes and enhancements need to be effected immediately, often well before the next major release is ready to ship.

Maintenance effort (bug-fixes and enhancements) in the current release may be incompatible with some of the functionality or refactoring already implemented in the next release

Linear Development

Mainline development has a number of advantages. It reduces complexity and redundant effort. Some early stage companies work closely enough with their few customers that they can focus development on what these customers need, and bug fixes are simply additions to the codeline.You can label each release point and then have your customers mover to the new release. They'll probably get some additional features, so they won't mind, and all of your work may be short term enough that you'll be able to always ship your code.

With success and planning comes a circumstance where the main codeline might not always be shippable, since the infrastructure needed to support a new feature might not be ready until after the code is. Also, with more than one customer, you may not be able to get your customer base to upgrade at the same time.

You need a way to get let released code evolve independently of mainline code so that you can do bug fixes.

Split maintence/release and active development into separate codelines. Keep each released version on a release line. Allow the line to progress on its own for bug fixes. Branch each release off of the mainline.

Rather than trying to accommodate maintenance of the current release and development of the next release in the same codeline, split maintenance and development off into separate codeline. All bug-fixes and enhancements to the current release take place in the maintenance line, effort for the next major release takes place in the development line. Ensure that changes in the maintenance line are propagated to the active development line in a regular fashion. Figure 17-4 shows this structure.

Propagate bug fixes from the mainline to the release line where possible. Once the mainline has progressed, you may still be making changes to the released line; Code on the release line becomes dead-end code when that release is no longer supported.

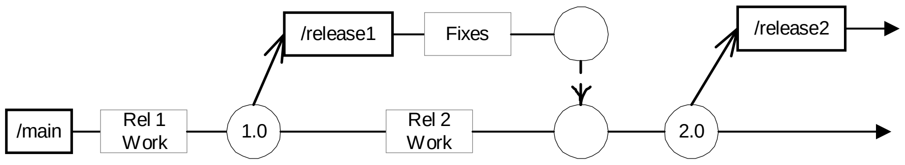


Figure 17-4 .Release Line

When you are ready to ship, label the code on the mainline and branch. Fix any errors on the released codeline on the branch, and merge any relevant changes back into the mainline before the next release. All work for future releases goes on the mainline.

At release time branch all code, including third party code.

Further Reading

- *Streamed Lines* (Appleton et al. 1998) describes more branching patterns,
- *Software Release Methodology* by Michael Bays (Bays 1999) discusses various types of codelines.

Chapter 18

Release-Prep Code Line

You're finishing up a release and also need to start continue development on the next release. You want to maintain an *ACTIVE DEVELOPMENT LINE (5)*.

◈ ◈ ◈

How do you stabilize a codeline for an impending release while also allowing new work to continue on an active codeline?

Before a release is ready to ship, there is often much work to do to get the active development line shippable. There are last minute bugs to fix, details related to installation and packaging and other last minute details to tend to. It is best to not do any major new work on the active development codeline while this clean up is going on, since you don’t want to introduce any new problems. You will want to have very restrictive check in and QA policies during this “clean-up” period.

One solution is to freeze development on the active development line until the release stabilizes. You can institute a strict policy that only essential changes are made to the codeline. If things go well, this may only take a day or so. But the stabilization work may involve part of the team however, and there is new work to do for the next release, so you really do not want to stop work at all. And if the code freeze lasts longer than this is very wasteful of resources, and frustrating to developers who are doing new work, since they will have to work without the version control system. If people work on the changes without checking them in to the version control stream, you lose all of the benefits of having a version control system, and out yourself at risk of having chaos when the freeze is lifted.

Another possible solution is to branch the codeline into a release codeline, and do all of your work on the branch. If you branch too early, you will have to do a lot of merging between the release line and the mainline. Branching gives you isolation, but at the expense of added work in doing merges.

Developers want to get work done, and avoid merges. Management wants the current code to be stable.

Idle Hands

Lots of places I’ve worked have instituted code freezes before releases. This is a good idea in principle, but when the code freeze lasts days or weeks, it seems like less of a good idea. The stated reason for freezing instead of doing something that allows parallel work is that it is less work. Of course, it adds frustration, and delays the release.

I’ve seen this so many times, often at the same time when there is lots of pressure to deliver code. Version control tools offer the ability to allow concurrent work, but they aren’t used often, or well.

Branch instead of Freeze

Create a release-engineering branch when code is approaching release quality. Finish up the release on this branch, and leave the mainline for active development. The branch becomes the release branch.

You mark the release of a product with a branch. Instead of branching off immediately *after* release, branch before the release. This allows you to branch instead of freeze. Instead of freezing the maine codeline during release engineering activities, create a separate line for release integration and engineering and allow other development to continue taking place on the development line.

Create the release engineering line when the code is approaching stability. The closer to "done" code you create the branch, the less merging you will have to do between this line and the mainline. The trade-off is that you wait longer, you may find yourself in a code-freeze situation.

This anti-freeze (release-engineering) line becomes the release-maintenance after a successful release. It still serves the same purpose of "sync and stabilize" but now it is an ongoing effort that continues even after the release.

In reality, there may be a small "freeze" window -- as long as it takes to create a consistent branch. If you can avoid this, all the better, but even if your "freeze" is short, you are still ahead of where you were when you had to freeze until you ship and release. Figure 18-1 illustrates this structure.

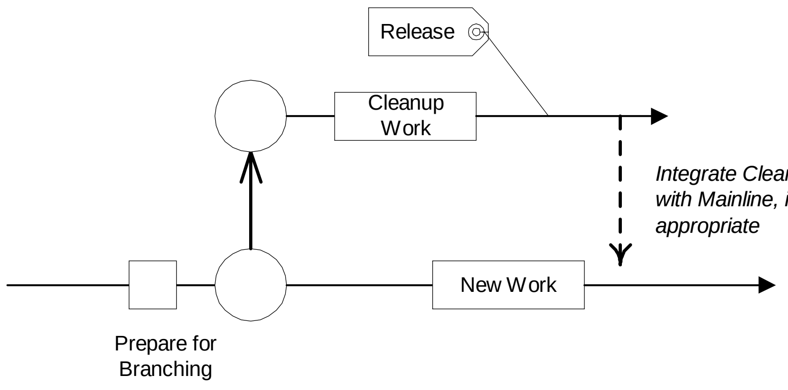


Figure 18-1 .Release-Prep Code Line

Changes can take place in each of the two codelines at the appropriate pace. Critical fixes and enhancements can be implemented and delivered without immediately impeding future development. Maintenance releases or “patches” can be periodically released without severely impacting development on the next release. The Code Line Owner of the development line can set a policy for how and when changes are propagated from the maintenance line to the development.

Unresolved Issues

If only a few people are working on the next release, instead of starting a release prep branch, start a Task Branch for the new work.

To keep the codeline in good shape while you are doing a potentially disruptive task, consider using a *TASK BRANCH (19)*. This pattern forms the basis for a *RELEASE LINE (17)*.

Chapter 19
Task Branch

Handling Long Lived Tasks

Some development tasks take a long time to implement, and intermediate steps are potentially disruptive to an *ACTIVE DEVELOPMENT LINE (5)*. This pattern describes how to reconcile long term tasks with an active development line.

◈ ◈ ◈

How can your team make multiple, long term overlapping changes to a codeline without compromising its consistency and integrity?

You usually want to use the version control system as a mechanism to keep the entire team up to data on what everyone is doing. Under normal circumstances, you will check in changes frequently. Some development tasks should not be integrated into the mainline to share with the rest of the team until they are complete. For example, you may be making a major change to an interface and you need to be sure that it works before you publish it to the team.

When you are doing parallel development without controlling the interaction between everyone's concurrent changes you can end up with wasted effort and rework. Most of the time you want to use the version control system as your communications mechanism, since you want to share your work with everyone as soon as you think that it is ready, since frequent integration is a good way to improve global stability. You also want to put your changes into version control as soon as is reasonable to ensure that you have traceability and recoverability. Also, on a practical note, often development systems do not have the same backup and recovery infrastructure around them as the drives that contain the version control software.

Some changes would destabilize the active codeline if check them in. For example, a major refactoring can't easily be done in stages, yet you don't want to wait a week before checking in the complete set of changes. Also, since your version control system is a good mechanism for communicating changes to other developers who are working on the same task, you need a place to check your code into. The alternatives, which include sharing files, and other mechanisms that bypass the version control system, can easily cause you to get out of synchronization with each other, even if your communication is good, which is often not the case because we get distracted. Figure 19-1 illustrates this concept.

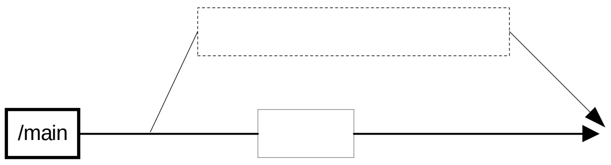


Figure 19-1 .Some Tasks are for the future

Another example of a situation where a small group of developers is working on a task without that can cause conflict is if you are approaching a product release, but a small part of your development team is working on a new feature for a subsequent release. You want this subset of your team to share code changes using some sort of version control system, but the changes cannot go into the active development line, since they are not for this release. If only a small part of the team is working on these changes, the overhead of creating a release line might be too great, since everyone is doing work on the release line, and the mainline should remain in sync with it. With some tools the only reliable way to synchronize both lines, is to check code into both places. Figure 19-2 illustrates this.

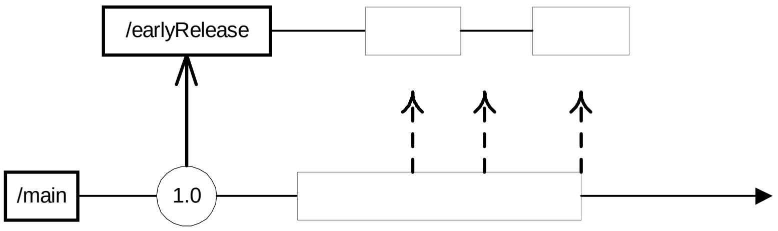


Figure 19-2 .Creating a Release Line too Early isTroublesome

A small team of developers working on a future task can have a more lax codeline policy with respect to keeping there work synchronized with the active mainline because they can communicate among themselves more efficiently than the larger team can.

You need a way to allow a team of developers to work on a task that is paralled to the mainline for a short period of time, while keeping all of the benefits of the version control tools.

Parallel Lines

We were changing the persistence mechanism for our application. This had far reaching implications. We were changing the mechanism because there were problems with the existing one, so there were bug fixes going on as well. The fact that there was some isolation at the module level between the persistence code and the other parts of the system made this a bit easier.

By creating a branch with the files that we were updating, we could allow a group of people to enhance the persistence mechanism while other work was going on.

Use Branches for Isolation

Fork off a separate branch for each activity that has significant changes for a codeline.

A branch is a mechanism for reducing risk. You can use a task branch as “a mechanism for physically isolating riskier development ventures from the code base” (Vance 1998).Once the activity is complete and all of its file changes have been tested, checkin files on the activity branch. When the task in complete, merge with the mainline. Michael Bays describes the scenario for a task branch (though not calling it by that name) (Bays 1999):

> It is common for a single developer to create a branch from the trunk in order to execute a change that will require a non trivial amount of time and not be impeded by changes others are making in the same files. As work in this developer branch continues, the source trunk will continue to iterate through revisions with the work of others. Eventually, the developer will want to take the files that she has changed and merge them back into the latest file versions on the trunk.

You need to be sure that all of the code that has changed on the mainline since the time you started your branch, still works with yours. Either merge the activity branch into the appropriate codeline (as a single transaction). Otherwise notify the appropriate codeline owner that the change is complete and ready to be merged and provide the codeline-owner with any other necessary information for finding, merging, and testing the change-task.

If your VCS supports ‘lazy branching’ where your branch inherits all the code from the mainline unless it change on the branch, you can use a task branch when you have to isolate a small amount of ‘future work’ from the mainline Figure 19-3 shows this structure.

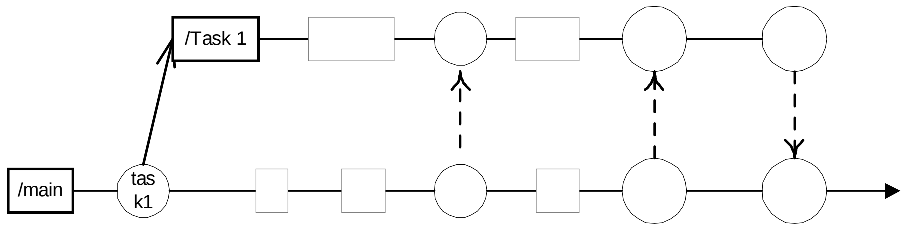


Figure 19-3 .Task Branch

Task branches are especially useful when you have multiple people sharing work that needs to be integrated together. For this reason, you can also use a task branch for integration prior to releasing a change to the active development line. You can integrate a number of changes into the task branch, and then merge the task branch back into the active development line when the changes pass your tests.

It is important to integrate changes from the active development line into the task branch frequently. You want the final integration of the task branch with the active codeline to go as smoothly as possible.

Chapter 20
Referenced Patterns

This chapter provides a brief summary of patterns from other sources that we reference in this book.

Named Stable Bases

Originally published in *A Generative Development Process Pattern Language* (Coplien 1995).

Intent: How frequently do you integrate? Stabilize system interfaces no more thanonce a week. Other software can be changes and integrated more frequently (summary from summaries are from the *Patterns Almanac* (Rising 2000)).

Daily Build and Smoke Test

Originally published in *A Generative Development Process Pattern Language* (Coplien 1995).

Intent: How do you keep the changes from getting out of hand and contain the potential for errors in the build? At least daily, build the software, and perform a smoke test to determine that the software is still usable.

Chapter A
Bibliography

Adolph, Steve, Paul Bramble, Alistair Cockburn, and Andy Pols. 2003. *Patterns for Effective Use Cases*. Boston, MA: Addison Wesley.

Alexander, C. 1979. *A Timeless Way of Building*: Oxford University Press.

Alexander, C., S. Ishikawa, and M. Silverstein. 1977. *A Pattern Language*: Oxford University Press.

Alexander, C., M. Silverstein, S. Angel, S. Ishikawa, and D. Abrams. 1975. *The Oregon Experiment*: Oxford Universiy Press.

Allen, Thomas J. 1997a. Architecture and Communication among Product Development Engineers. Cambridge, MA: MIT Sloan School. International Center for Research on Management Technology.

———. 1997b. Organizational Structure for Product Development. Cambridge, MA: MIT Sloan School. International Center for Research on Management Technology.

Allen, Thomas J., Breffni Tomlin, and Oscar Hauptman. 1998. Combining Organizational and Physical Location to Manage Knowledge Dissemination. Cambridge, MA: MIT Sloan School. International Center for Research on Management Technology.

Alpert, Sherman R., Kyle Brown, and BobbyWoolf. 1998. *The Design Patterns Smalltalk Companion, The Software Patterns Series*. Reading, Mass.: Addison-Wesley.

Appleton, Brad, Steve Berczuk, Ralph Cabrera, and Robert Orenstein. 1998. Streamed Lines: Branching Patterns for Parallel Software Development. Paper read at Fifth

Annual Conference on Pattern Languages of Programs, August 11-14, at Monticello, IL.

Babich, Wayne A. 1986. *Software Configuration Management : Coordination for Team Productivity*. Reading, Mass.: Addison-Wesley.

Bass, Len, Paul Clements, and Rick Kazman. 1998. *Software Architecture in Practice, Sei Series in Software Engineering*. Reading, Mass.: Addison-Wesley.

Bays, Michael E. 1999. *Software Release Methodology*. Upper Saddle River, N.J.: Prentice Hall PTR.

Beck, Kent. 2000. *Extreme Programming Explained : Embrace Change*. Reading, MA: Addison-Wesley.

Berczuk, Stephen. 1994. Finding Solutions through Pattern Languages. *IEEE Computer* 27 (12 (Dec. 1994)):75-76.

Berczuk, Stephen P. 1995. A Pattern for Separating Assembly and Processing. In *Pattern Languages of Program Design*, edited by J. Coplien and D. Schmidt. Reading, MA: Addison-Wesley.

———. 1996a. Organizational Multiplexing: Patterns for Processing Satellite Telemetry with Distributed Teams. In *Pattern Languages of Program Design*, edited by J. Vlissides, J. Coplien and N. Kerth. Reading, MA: Addison-Wesley.

Berczuk, Steve. 1996b. Configuration Management Patterns. Paper read at Third Annual Conference on Pattern Languages of Programs, at Monticello, IL.

Berczuk, Steve, and Brad Appleton. 2000. Getting Ready to Work: Patterns for a Developer's Workspace Paper read at Pattern Languages of Programs, at Monticello, IL.

Booch, Grady. 1996. *Object Solutions : Managing the Object-Oriented Project*. Menlo Park, Ca.: Addison-Wesley Pub. Co.

Booch, Grady, James Rumbaugh, and Ivar Jacobson. 1999. *The Unified Modeling Language User Guide, Addison-Wesley Object Technology Series*. Reading, MA: Addison Wesley Longman.

Brooks, Fred. 1975. *The Mythical Man Month*. Reading, MA: Addison Wesley.

Brooks, Frederick P. 1995. *The Mythical Man-Month : Essays on Software Engineering*. 20th Anniversary ed. Reading, Mass.: Addison-Wesley Pub. Co.

Brown, William J., Hays W. McCormick, and Scott W. Thomas. 1999. *Antipatterns and Patterns in Software Configuration Management*. New York: Wiley.

Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 1996. *Pattern-Oriented Software Architecture: A System of Patterns*. Chichester, England: John Wiley & Sons.

Cabrera, Ralph, Brad Appleton, and Steve Berczuk. 1999. Software Reconstruction: Patterns for Reproducing the Build. In *Proccedings of the Sixth Annual Conference on Pattern Languages of Program Design*. Monticello, IL.

Cockburn, Alistair. 2000. *Writing Effective Use Cases, The Crystal Series for Software Development*. Boston: Addison-Wesley.

Conradi, Reidar, and BernhardWestfechtel. 1998.Version Control Models for Software Configuration Management. *ACM Computing Surveys* 30 (2):232-278.

Coplien, James O. 1995. A Generative Development Process Pattern Language. In *Pattern Languages of Program Design*. Reading, MA: Addison-Wesley.

Coplien, James O., and Douglas Schmidt, eds. 1995. *Pattern Languages of Program Design*. Reading, MA: Addison Wesley.

Dart, Susan. 1992. The Past, Present, and Future of Configuration Management: Software Engineering Institute.

DeMarco, Tom, and Timothy R. Lister. 1987. *Peopleware : Productive Projects and Teams*. New York, NY: Dorset House Pub. Co.

Dikel, David M., David Kane, and James R. Wilson. 2001. *Software Architecture : Organizational Principles and Patterns*. Upper Saddle River, NJ: Prentice Hall.

Fisher, Roger, William Ury, and Bruce Patton. 1991. *Getting to Yes : Negotiating Agreement without Giving In*. 2nd ed. New York, N.Y.: Penguin Books.

Fogel, Karl Franz, and Moshe Bar. 2001. *Open Source Development with Cvs*. 2nd ed. Scottsdale, AZ: Coriolis Group Books.

Fowler, Martin. 1999. *Refactoring: Improving the Design of Existing Code*. Edited by J. C. Shanklin, *Object Technology Series*. Reading, MA: Addison-Wesley.

Fowler, Martin, and Matthew Foemmel. 2002. *Continuous Integration* 2001 [cited March 7 2002]. Available from http://www.martinfowler.com/continuousIntegration.html.

Gabriel, Richard P, and Ron Goldman. 2000. *Mob Software: The Erotic Life of Code* : Deamsongs Press.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley.

Gause, Donald C., and Gerald M. Weinberg. 1990. *Are Your Lights On? How to Figure out What the Problem Really Is.* New York, NY: Dorset House.

Goldfedder, Brandon. 2002. *The Joy of Patterns : Using Patterns for Enterprize Development, The Software Patterns Series*. Boston: Addison-Wesley.

Grinter, Rebecca. 1995. Using a Configuration Management Tool to Coordinate Software Development. Paper read at ACM Conference on Organizational Computing Systems, August 13 - 16 1995, at Milpitas, CA.

Harrison, Neil, Brian Foote, and Hans Rohnert, eds. 2000. *Pattern Languages of Program Design 4*. Edited by J. M. Vlissides, *Software Patterns Series*. Reading, MA: Addison Wesley Longman.

Highsmith, Jim. 2002. *Agile Software Development Ecosystems*. Edited by A. Cockburn and J. Highsmith, *Agile Software Development Series*. Boston, MA: Pearson Education.

Hightower, Richard, and Nicholas Lesiecki. 2002. *Java Tools for Extreme Programming : Mastering Open Source Tools Including Ant, Junit, and Cactus.* New York Chichester: Wiley.

Hunt, Andrew, and David Thomas. 2000. *The Pragmatic Programmer : From Journeyman to Master.* Reading, Mass: Addison-Wesley.

Hunt, Andy, and Dave Thomas. 2002a. Software Archaeology. *IEEE Software* 19 (2):20-22.

———. 2002b. Ubiquitous Automation. *IEEE Software* 18 (1):11-13.

Jeffries, Ron, Ann Anderson, and Chet Hendrickson. 2000. *Extreme Programming Installed*. Boston, MA: Addison-Wesley.

Karten, Naiomi. 1994. *Managing Expectations. Working with People Who Want More, Better, Faster, and Sooner.* New York, NY: Dorset House.

Kernighan, Brian W., and Rob Pike. 1999. *The Practice of Programming, Addison-Wesley Professional Computing Series*. Reading, MA: Addison-Wesley.

Krutchen, Philippe. 1995. The 4+1 Model View of Architecture. *IEEE Software* 12 (6):42-50.

Leon, Alexis. 2000. *A Guide to Software Configuration Management*. Norwood, MA: Artech House.

Manns, Mary Lynn, and Linda Rising. 2002. *Introducing Patterns into Organizations* 2002 [cited May 23 2002].Available from http://www.cs.unca.edu/~manns/intropatterns.html.

Martin, Robert C., Dirk Riehle, and Frank Buschmann. 1998. *Pattern Languages of Program Design 3, The Software Patterns Series*. Reading, Mass.: Addison-Wesley.

McConnell, Steve. 1993. *Code Complete : A Practical Handbook of Software Construction*. Redmond, Wash.: Microsoft Press.

———. 1996. *Rapid Development, Taming Wild Software Schedules*. Redmond, WA: Microsoft Press.

———. 2000. What's in a Name? *IEEE Software* 17 (5):7-9.

———. 2002. Closing the Gap. *IEEE Software* 19 (1):3-5.

Mikkelsen, Tim, and Suzanne Pherigo. 1997. *Practical Software Configuration Management : The Latenight Developer's Handbook*. Upper Saddle River, NJ: Prentice Hall PTR.

Myers, Glenford J. 1979. *The Art of Software Testing*. New York: Wiley.

Olson, Don Sherwood, and Carol L. Stimmel. 2002. *The Manager Pool : Patterns for Radical Leadership, The Software Patterns Series*. Boston: Addison-Wesley.

Oshry, Barry. 1996. *Seeing Systems. Unlocking the Mysteries of Organizational Life*. San Francisco, CA: Barrett-Koehler.

Rising, Linda. 2000. *The Pattern Almanac 2000, Software Patterns Series*. Boston: Addison-Wesley.

Roche, Ted, and Larry C. Whipple. 2001. *Essential Sourcesafe*: Hentzenwerke Corporation. Original edition, June 1, 2001.

Schmidt, Douglas C., Michael Stal, Hans Rohnert, and Frank Buschmann. 2000. *Pattern-Oriented Software Architecture : Patterns for Concurrent and Distributed Objects*. 2nd ed. New York: Wiley.

Shaw, Mary, and David Garlan. 1996. *Software Architecture : Pespectives on an Emerging Discipline*. Upper Saddle River, N.J.: Prentice Hall.

Tichy, Walter F. 1985. A System for Version Control. *Software Practice and Experience* 15 (7).

Ury, William. 1993. *Getting Past No : Negotiating Your Way from Confrontation to Cooperation*. Rev. ed. New York: Bantam Books.

Vance, Stephen. *Advanced Scm Branching Strategies* 1998 [cited. Available from http://svance.solidspeed.net/steve/perforce/Branching_Strategies.html.

Vlissides, John. 1998. *Pattern Hatching : Design Patterns Applied, The Software Patterns Series*. Reading, Mass: Addison-Wesley.

Vlissides, John, James Coplien, and Norm Kerth, eds. 1996. *Pattern Languages of Program Design 2*. Reading, MA: Addison-Wesley.

Weigers, Karl E. 2002. Fightin'Words. *StickyMinds.com*.

Weinberg, Gerald M. 1986. *Becoming a Technical Leader*. New York, NY: Dorset House.

———. 1991a. *Quality Software Management*. New York, N.Y.: Dorset House Pub.

———. 1991b. *Quality Software Management: Volume 1 Systems Thinking*. New York, N.Y.: Dorset House Pub.

———. 1993. *Quality Software Management, Volume 2: First Order Measurement*. New York, N.Y.: Dorset House Pub.

———. 2002. *More Secrets of Consulting : The Consultant's Tool Kit*. New York: Dorset Horse Pub.

White, Brian. 2000. *Software Configuration Management Strategies and Rational Clearcase : A Practical Introduction, Addison-Wesley Object Technology Series*. Boston, MA: Addison-Wesley.

Whitgift, David. 1991. *Methods and Tools for Software Configuration Management, Wiley Series in Software Engineering Practice*. Chicester, England: Wiley.

Wingerd, Laura, and Christopher Seiwald. 1998. High-Level Best Practices

in Software Configuration Management. Paper read at Eight International Workshop

on Software Configuration Management, July, 1998, at Brussels.

Chapter B
SCM Resources On-line

This section describes some useful SCM resources available on theWorld-Wide Web.

The Configuration Management Yellow Pages

- http://www.cmtoday.com/yp/configuration_management.html

If you only go one place for the most comprehensive set of links on the topic of software configuration management, this is the place to go. It has *everything* (or at least it always seems to) and is updated regularly. It used to be maintained by Andre van der Hoek, but has since been passed on to the folks who host the CM today newsletter and *cmtalk* mailing list

CM Crossroads - Online Community and Resource Center for CM Professionals

- http://www.cmcrossroads.com/

If you only go two places for the most comprehensive set of links on the topic of software configuration management, this is the second place to go. It hasn't been around as long as the CM Yellow Pages, but it is very comprehensive, has a regular newsletter, and growing list of resource links on several topics related to SCM as well as SCM.

CM Today - Daily Configuration Management News

- http://www.cmtoday.com/

A daily CM newsletter and resource site updated daily. The same site houses the CM Yellow pages and CM job listings. It also has a web page where you can subscribe to the cmtalk mailing list, and view the message archives of the mailing list.

UCM Central - Unified Configuration Management

- http://www.ucmcentral.com

Another good site with resources and a portal and links to information about various tools and vendors. It also contains several informative articles, diagrams, and templates

ACME - Assembling Configuration Management Environments (for Software)

- http://acme.bradapp.net

This is the web site created by one of the authors of this book in their quest to uncover and disseminate SCM patterns and best practices. The site has a rather extensive list of SCM definitions, a set of SCM recommended readings, several papers and presentations by the authors on the subject of SCM patterns and best practices, and a very respectable collection of links to SCM research and practice (unfortunately, the links are very much in need of some serious updating).

The Software Engineering Institute's SCM Publications

- http://www.sei.cmu.edu/legacy/scm/

Some links to general resources, FAQs, and best of all, the SEI CM documents archive. Although they are somewhat dated, most of these papers are excellent, and many are considered "classics" in the field. Of particular note are the following papers by Susan Dart and/or Peter Feilir:

- *The Spectrum of Functionality in CM Systems*, by Susan Dart
- *Concepts in CM*, by Susan Dart
- *The Past, Present and Future of CM*, by Susan Dart
- *Configuration Management Models in Commercial Environments*, by Peter Feiler
- *Transaction-Oriented CM: a Case Study*, by Peter Feiler and Grace Downey

Steve Easterbrook's Configuration Management (CM) Resource Guide

- http://www.cmiiug.com/sites.htm

A great resource for addresses, points of contacts, and titles in a rather plain FAQ-like format. The parent site is also the home of the Institute for Configuration Management (www.icmhq.com)

The Software Configuration Management FAQ

- http://www.daveeaton.com/scm/CMFAQ.html

A classic, compiled and maintained by Dave Eaton, who put together the first such FAQ for the comp.software.config-mgmt Usenet newsgroup. The FAQ actually consists of three separate FAQs: one on SCM in general, one on version control and SCM tools, and another on problem-tracking tools.

The Association for Configuration and Data Management

- http://www.acdm.org/

Software Engineering Resource List for Software Configuration Management

- SCM - http://wwwsel.iit.nrc.ca/favs/CMfavs.html

A very nice SCM site from the Software Engineering Group (SEG) at the Institute for Information Technology

R.S. Pressman and Associates Software Engineering Resources for SCM

- http://www.rspa.com/spi/SCM.html

Roger Pressman’s site of SCM liks for software engineering (Pressman is the author of several well known software engineering books).

SEweb Software Configuration Management resources at Flinders University

- http://see.cs.flinders.edu.au/seweb/scm/

Pascal Molli’s “CM Bubbles” SCM resources page

- http://www.loria.fr/~molli/cm-index.html

The Usenet newsgroup comp.software.config-mgmt

- news:comp.software.config-mgmt

These can now also be read, searched, and posted to using Google:

- http://groups.google.com/groups?hl=en&lr=&group=comp.software.config-mgmt.

Chapter C
Tool Support for SCM Patterns

*with assistance from Bob. Ventimiglia (**http://www.bobev.com/**)*

This section describes how the SCM patterns in this book map to the concepts implemented by several commonly used software version control (VC) tools. The intent is

TABLE 1-1. **Some Commonly Used Version Control Tools**

Tool	**Vendor**	**Web Site**
VSS - Visual Source Safe	Microsoft	http://msdn.microsoft.com/ssafe/
CVS - the Concurrent Versions System	CVS is Open Source - development is hosted by Collab.Net	http://www.cvshome.org/
Perforce	Perforce Software	http://www.perforce.com/
BitKeeper	BitMover Inc.	http://www.bitkeeper.com/
AccuRev	AccuRev Inc.	http://www.accurev.com/

TABLE 1-1. **Some Commonly Used Version Control Tools**

Tool	**Vendor**	**Web Site**
ClearCase	Rational Software	http://www.rational.com/products/clearcase/
UCM - Unified Change Management	Rational Software	http://www.rational.com/products/clearcase/
CM Synergy	Telelogic	http://www.telelogic.com/products/synergy/
StarTeam	Starbase	http://www.starbase.com/products/starteam/
PVCS Dimensions	Merant PVCS	http://www.merant.com/pvcs
PVCS Version Manager	Merant PVCS	http://www.merant.com/pvcs
MKS Integrity	MKS Inc.	http://www.mks.com/products/sie/

not to give the operational details for a “cookbook” or “HowTo” but to instead refer to the tool-specific mechanisms, and how they work together to support our SCM patterns. This should provide a conceptual base for looking up detailed usage instructions in the product documentation.

For each tool, its basic concepts and terminology are briefly described in order to provide a high-level overview of how to use it to perform the following common activities:

- Create a *PRIVATE WORKSPACE (6)*
- Configure and populate it from a *REPOSITORY (7)* or from a codeline
- Create a change-task and / or a *TASK BRANCH (19)*
- Update the workspace with the latest state of the codeline
- Perform a *TASK LEVEL COMMIT (11)* of the changes from the workspace into the codeline
- Create a new codeline (for a *MAINLINE (4)*, an *ACTIVE DEVELOPMENT LINE (5)*, a *RELEASE LINE (17)*, a *RELEASE-PREP CODE LINE (18)*, or a *THIRD PARTY CODELINE (10)*)
- Create a label or version-identifier for one of the *NAMED STABLE BASES (20)*

VSS - Visual Source Safe

VSS is one of the more commonly used VC tools among those using a Microsoft-

TABLE 1-2. **Mapping of SCM pattern concepts to VSS concepts**

SCM Pattern concept name	**VSS concept name**	**Comments**
Repository	master project	
Development Workspace	working directory	
Codeline	share & branch	see also "pin"
Change Task	N/A	
Workspace Update	get project	from the master/codeline project into the working directory
Task-level Commit	checkin project	
Task Branch	N/A	
Label	label project	

based integrated development environment (IDE) such as Visual C++. Although it has a nice GUI and very seamless integration with the programming language environment, VSS is among the less capable tools described here. VSS has relatively limited support for branching and parallel development. It is not intended for projects which regularly require multiple codelines.

A repository in VSS corresponds to a database which holds one or more *master projects*. VSS supports "project-oriented" operations that operate on all selected files, or all the files defined to be in a specified project. A master project is simply a VSS project which holds the master copies of a set of files and their latest versions (it also serves as the mainline).

A development workspace is created in VSS by associating a project with a *working directory.*

The working directory is populated by performing a project-oriented *get* operation to retrieve the latest versions of a project's files into the working directory for read-only access. A *checkout* operation is used to copy a writable copy of a file into the working directory.

The working directory may be updated with the latest project versions by doing another *get* project operation from the master (or codeline) project into the working directory.

Changes from the working directory may be “committed” to the codeline by doing a *checkin* project operation from the working directory into the master (or codeline) project where the latest versions of the project are stored for that codeline.

VSS doesn’t have a separate notion of a change-task, it relies upon project operations in the working directory to be performed in task order. And since branching in VSS is limited, branches are almost never created for the purpose of a single task.

A new codeline is created for a project by *“sharing and branching”* the files in that project to create a new copy of that project:

- The *share* operation (formerly called “pin”) links and copies the selected file versions from the master project into a new project for the new codeline.
- The *branch* operation makes sure the selected files are copies (rather than links) and may have their contents revised and evolved independently of the project from which they were initially shared.

A new label is created by performing the *label* operation on the selected set of files in the project.

Caveats for using VSS:

- The get operation works properly for a workspace update only when you check out the files you want to change *before* making your local changes to them. If you modify your local copies, but don’t invoke the get operation until just before you are ready to check them in, then VSS wont know that your locally modified files were checked out before other versions that were checked in after you made your changes (but before you checked the files out). If you make sure to checkout the files before you make changes to your local copies, VSS will know to perform the merge operation for any files that you changed that also changed in the codeline
- VSS doesn’t effectively support a branching depth of more than 2 levels. If you do this, VSS wont remember the full merge ancestry more than one level deep and merging and reconciling changes back more than one level will be more difficult.

CVS - The Concurrent Version System

CVS is OpenSource software and is probably one of the most widely known and used

TABLE 1-3. Mapping of SCM pattern concepts to CVS concepts

SCM Pattern concept name	CVS concept name	Comments
Repository	Repository	also known as CVSROOT
Development Workspace	*Working Directory* or *Working Copies*	create using "*cvs checkout -R ...*" or "*cvs update -R ...*"; also see "*cvs export*"
Codeline	branch	create using "*cvs tag -b*"
Change Task	N/A	
Workspace Update	update	"*cvs update*"
Task-level Commit	commit	"*cvs commit*"
Task Branch	branch	create using "*cvs tag -b*"
Label	tag	see also *rtag*
Third Party Codeline	vendor branch	Use "*cvs import -b*"

version control tools in common use today. CVS supports the majority of these pattern concepts using the *copy-modify-merge* model:

- *Copy* - A developer sets up a working directory and requests a working copy of the files in the project by checking out files from the project into the working directory. The syntax would be "checkout <*options*> <*module-name*>" (or "update <*options*> <*codeline-name*>" to populate the workspace with the latest files from the codeline).
- *Modify* - The developer edits any working copies of files in the working directory using the "checkout" command
- *Merge* - the developer performs a "commit" operation to checkin their modifications to the repository

The main commands used are *tag, checkout, update,* and *commit*. One typically uses the "-R" option with the *update, commit,* and *tag* commands to recursively apply to all files in the working directory. Branches for a codeline or a task branch are created using the "*tag*" command with the "-b" option. The "*update*" command synchronizes the developer's workspace with the latest state of the named codeline. When the time comes to create a label (or, if desired, a checkpoint), the "*tag*" command can again be used (this time without the "-b" option).

As a special case, CVS was specifically designed to support the concept of a "*vendor branch*" (a THIRD PARTY CODELINE *(10)*). The "*import*" command was tailor made for this purpose!

Some caveats for using CVS:

- CVS doesn't have as nice GUI support for branching and merging that many of the commercial tools have (but other OpenSource GUI's exist and more seem to be written every year). This places the burden upon you to mentally draw the visual picture of the codeline and branching structure in your mind.
- CVS also isn't quite as expert at tracking merge-history as some of the more advanced commercial tools. If a version has been merged, and a particular set of changes within the file were intentionally left unchanged (or changed differently), then subsequent merges will continue to present those same changes, even though the version they came from already had its differences "reconciled." Merge technology in many of the more capable version control tools know not to look at the contents of changes for versions that are already part of the "merge ancestry" of the current version.

Perforce

Perforce is a very commonly used, simple but powerful commercial VC tool with outstanding support for branching and merging. It bills itself the “fast software configuration management system” and prides itself on performance, and distributed TCP / IP client / server operation.

TABLE 1-4. Mapping of SCM patterns concepts to Perforce concepts

SCM Pattern concept name	**Perforce concept name**	**Comments**
Repository	depot	
Development Workspace	*client workspace* and *client spec*	p4 client and p4 sync
Codeline	branch	p4 branch and p4 integrate
Change Task	changelist	also see *jobs*
Workspace Update	sync	p4 sync and p4 resolve
Task-level Commit	submit	p4 submit
Task Branch	branch	p4 branch and p4 integrate -b
Label	label	p4 label and p4 labelsync

A repository in Perforce is called a *depot*. The Perforce server centrally manages access to the depot from connecting clients.

Perforce workspaces are called a *client workspace* and are configured by specifying a *client spec* to create a *client view* in the workspace. A codeline is called a *branch* in Perforce (as is a task branch). Perforce tracks the changes between parent and child branches, so it knows when a file version has already been merged and reconciled or not.

Perforce uses an “*atomic change transaction*” model where an operation either succeeds on all files involved in an operation or else on none of them (an unexpected interrupt should never create a partially completed update or commit). Perforce’s *branches* (and resulting branch tree) are depot-wide rather than file-specific. This makes for a conceptually powerful branching model for managing parallel development with multiple codelines. (Other VC tools achieve this result with “projects” or “workspaces” or “streams” using a hierarchical structure.) Perforce also has a means of allowing configurable change review notifications, daemons, and triggers to add customized checks and verification where desired.

A Perforce workspace is setup by appropriately setting the *P4PORT* and *P4CLIENT* environment variables, and running "*p4 client*" to edit the client spec to the desired client view.

Once the workspace is setup and configured, "*p4 sync*" will populate the workspace with read-only copies of the latest files selected by the client spec. Changes are made by using the "*p4 edit*" command to obtain a writable copy of a file to modify.

The "*p4 sync*" command is also used to do a workspace update. Checked-out files will not be modified by "*p4 sync*." A "*p4 resolve*" command must be used to reconcile the differences between the checked-out files and the latest versions in the codeline.

Files revised with the edit, add, and delete commands are added to a *changelist* that Perforce maintains for your workspace. Those changes can be committed to the depot using the "*p4 submit*" command. If any of your files aren't the most recent version on the codeline, you will get a submit error. Submit errors and merge conflicts are resolved using the "*p4 resolve*" command. You can use "*p4 resolve -n*" to see which files need to be resolved.

Branches are created using "*p4 integrate*." The *integrate* command can also be used with a *branch spec* to make the branch name automatically remember the mappings for the branched files. (This is convenient to use for codelines and task-branches when you have planned for them in advance.)

The "*p4 label*" command will create labels for releases, builds, and checkpoints. The "p4 labelsync" command can revise the set of files belonging to a label.

BitKeeper

BitKeeper bills itself as the "distributed" scalable SCM system. It also exhibits excel-

TABLE 1-5. **Mapping of SCM pattern concepts to BitKeeper concepts**

SCM Pattern concept name	**BitKeeper concept name**	**Comments**
Repository	master repository	
Development Workspace	developer repository	create with "*bk clone*"
Codeline	integration repository	also see Line of Development (LOD)
Change Task	change-set	use "*bk citool*" or "*bk commit*"; also see "*bk revtool*"
Workspace Update	pull and resolve	use "*bk pull*" and "*bk resolve*"
Task-level Commit	commit and push	use "*bk citool*", "*bk commit*" and "*bk push*"; also see "*bk ci*"
Task Branch	developer repository	also see Line of Development (LOD)
Label	change-set, also a tag	use "*bk tag*"; also see "*bk commit -s*"
Checkpoint	change-set, also a tag	or just use "*bk unpull*" to rollback to previous state after an unsuccessful "*bk pull*"

lent performance and reliability. Rather than a client/server model, BitKeeper instead uses a *fully replicated peer-to-peer model* of operation which allows for fully disconnected use from the "master" repository. BitKeeper provides triggers that allow for customization of the most common repository-wide and file-specific operations.

The key to understanding BitKeeper's operational model is to understand that every BitKeeper workspace is also a repository. Changes, in the form of *change-sets*, are made in a workspace, and developers can propagate change-sets back and forth between workspaces using *push* and *pull* operations. All change-set operations are atomic in BitKeeper.

Due to this simple but powerful model of distributed workspaces as repositories for transmitting and receiving change-sets, both codelines and task-branches can be represented as workspaces in BitKeeper. BitKeeper workspaces operate like an unnamed branch. With BitKeeper, you plan your codelines and create a hierarchy of integration workspaces for each codeline in your hierarchy. You don't have to create branches in addition to codelines: when a change-set is pulled from another repository, the receiving workspace knows at that time which file had concurrent changes creates branches only for those files. (Think of it as “branch on demand”).

A codeline corresponds to an integration repository used for the purpose of pulling in change-sets and then pushing them to higher-level “codelines” in the hierarchy. A task-branch corresponds to a development repository for a single development task, allowing for private local changes to be made which won’t be seen by other repositories until they choose to “pull” them in.

So developers don’t need to keep track of both a task-branch and a workspace. They just work in their workspace, and after their changes are done, BitKeeper takes care of worrying whether or not any files need to be branched. This eliminates the need in a lot of other VC tools to have merges between branches that don’t change the file contents. This, combined with BitKeeper’s merging technology makes it very easy in BitKeeper to find the change-set that first introduced a particular line of code that was propagated across several codelines.

Another important aspect of BitKeeper is that a *change-set* is repository wide: it not only captures all the changes, but the context of the changes as well (the state of the repository in which the changes were made). This lets a change-set also act as a label or a checkpoint. It also means BitKeeper is “time-safe” in its ability to track the historical evolution of change-sets as they are pushed and pulled throughout a “promotion hierarchy” of workspaces.

Like most of the other tools described her, Bitkeeper may be used via a GUI or from the command line. The typical developer scenario for making a change is:

Create a new workspace using “*bk clone*” from the parent repository (or use an existing workspace without any pending changes). One can use the “*bk get*” or “*bk edit*” operations to populate and access files in the repository to make changes (*get* obtains read-only copies of files whereas *edit* obtains writable copies).

Use “*bk pull*” to update your workspace with the latest changes from the parent repository (the “codeline”). Use “*bk resolve*” immediately afterward to merge and resolve any resulting conflicts from the *pull* operation.

When changes are complete, the “*bk citool*” operation will check them in to your local repository. The “*bk commit*” operation will then create a change-set for your changes. The “*bk push*” operation will then propagate the changes back to the parent repository.

Create labels using “*bk tag*” or with the “-s” option to the commit operation.

AccuRev

AccuRev is a more recent VC tool offering that is not as well known as most of the other tools described here. Like Perforce, AccuRev uses the term *depot* to refer to a

TABLE 1-6. Mapping of SCM pattern concepts to AccuRev concepts

SCM Pattern concept name	AccuRev concept name	Comments
Repository	depot	
Development Workspace	workspace	*accurev mkws* and *accurev mksnap*
Codeline	stream	also see *backing stream* and *base stream*
Change Task	transaction	also see *workspace stream*
Workspace Update	update	See also the *pop* command and *merge -o*
Task-level Commit	promote	*promote -k* moves changes from the workspace stream so they are visible in the codeline
Task Branch	workspace stream	also see *dynamic stream*
Label	real version	See also “virtual versions” and “snapshot streams”
Checkpoint	checkpoint	create using the ‘*keep*’ command

repository. Also like Perforce, AccuRev prides itself on performance, distributed TCP/IP client/server operation, and atomic change-transactions (as well as integrated issue tracking).

One of AccuRev’s distinguishing features is that it is *TimeSafe*: it doesn’t just version all your data, it also versions all your metadata. So in addition to being able to track and reproduce prior contents of files, it can also track and reproduce prior definitions of labels (which it calls *checkpoints*) and *streams*, as well as other AccuRev metadata.

Central to understanding AccuRev is its very simple but powerful notion of a *stream*. AccuRev streams can be used as codelines, workspaces, and labels. An AccuRev *stream* is a logical set of files and file versions in the depot. An AccuRev *workspace* is simply a place in which you perform work on a stream.

Streams may be static or dynamic. Static streams may not have their contents changed and may serve as known stable configurations or even fill the same purpose as a label. Dynamic streams can have their contents changed in a workspace.

Dynamic streams may be linked together in a hierarchical fashion to create a promotion hierarchy for workflow and integration. The topmost stream in the hierarchy is called the "*base stream*" and serves as a "mainline." Other parent streams serve as a "*backing stream*" to their child streams and provides a starting point off the mainline for other development projects (codelines) that will be long-lived. The child streams are "*workspace streams*" where developers make their changes to the codeline.

So, in AccuRev, a *base stream* acts as a mainline, a *backing stream* acts as a codeline, and a *workspace stream* acts as both a change-task and a task-branch (with the changes in the stream associated with a transaction). A *static stream* is a snapshot that may serve as a label.

AccuRev also has "*real versions*" and "*virtual versions.*" A real version is created anytime a stream is checkpointed using the *keep* command. A virtual version is an alias for a real version, but allows the name of the virtual version to be used as the corresponding real version evolves dynamically over time. So the history of a virtual version is a progression of real version names and their sequential numbers.

AccuRev may be used via a GUI or from the command line. The typical developer scenario for making a change is:

Create your workspace using the "*accurev mkws*" command. Then edit files at will in your workspace stream

To update your workspace, do an "*accurev update*" followed by an "*accurev merge -o*" to resolve any resulting conflicts.

To commit your changes, do an "accurev keep -m" to create a *checkpoint* of your stream, then do "*accurev promote -k*" to commit your changes from the workspace stream to its backing stream.

ClearCase - base functionality (non-UCM)

ClearCase is among the more popular and more sophisticated version control tools

TABLE 1-7. Mapping of SCM pattern concepts to base ClearCase concepts

SCM Pattern concept name	**base ClearCase concept name**	**Comments**
Repository	Versioned Object Base (VOB)	
Development Workspace	view and config-spec, also view profile	cleartool mkview, cleartool edcs, Make Branch with View Profile
Codeline	project branch	cleartool mkbrtype, mkbranch config-spec rules, View Profiles
Change Task	N/A	
Workspace Update	findmerge	cleartool findmerge, MergeManager
Task-level Commit	findmerge	cleartool findmerge, MergeManager
Task Branch	private branch	cleartool mkbrtype, Make Private Branch
Label	label	cleartool mklbtype, cleartool label

on the market. Like Perforce, ClearCase also has very conceptually powerful parallel development and branching capabilities. ClearCase has a "base" option which lets you role your own process and policies based on the framework it provides, as well as a "UCM" option which supports Rational's "Unified Change Management" for activity-based SCM using higher-level concepts than base clearcase alone.

ClearCase is one of the most configurable tools on the market, providing customizable triggers for just about every operation imaginable. One thing that ClearCase does which most other VC tools do not is that it version is directories as well as files. That means it remembers which files participated in which version of a particular directory.

The base ClearCase option does not directly support the notion of a change-task, although task-branches are commonly used for this purpose. The ClearCase GUI on Windows platforms supports something called *"Private Branches*" and "*View Profiles*" which are a way of automating support for task branches and creating workspaces with the proper configuration of versions from the project branch. Using private branches, the typical development scenario is to:

Create a view, using an appropriate view profile for the desired project. This will also ask whether or not a private branch should be used.

Perform checkouts in the view (and checkins if using a task-branch) to make the necessary changes.

“Finish” the private branch by merging the changes back into the codeline

From the command-line, without private branches, the typical scenario is:

Create a view using the *mkview* command.

Create a branch (if desired) using the *mkbrtype* command.

Configure the view to select the appropriate versions using *edcs* or *setcs*.

> **NOTE:** the above three steps are typically automated and simplified into a single script or batch-file that takes a “task” name (to use as part of the name for the view and branch) and some kind of “project” identifier to determine the view configuration. The details of where the view’s file storage is created, and which specific configuration rules to use are often encapsulated into this script so the typical developer doesn’t have to worry about those details.

Perform checkouts and (if using a task-branch) checkins as desired. If you are working on a task-branch, any checkins are effectively private versions that are captured in the VOB for posterity, but appear only on your private branch and not in the codeline.

When desired, update the view using the *findmerge* command (typically with the *-flatest* or *-fversion* option). This is not necessary if your view is a *dynamic view* selecting the */LATEST* versions of the codeline. ClearCase’s virtual file systems ensures that the versions in a dynamic view are always synchronized with the */LATEST* versions specified by the view’s config-spec. Snapshot views, and views which select a labeled state of the codeline (rather than */LATEST*) will need to be updated before doing a commit.

Commit changes by going into an integration view and doing a *findmerge -ftag* (or *-fversion*) of the changes from your development task/view into the integration view (here again, a script or batch file is often used to simplify the interface and conceptual complexity of the *findmerge* command)

Labels are created using the *mklbtype* command to create th.e label name and the *mklabel* command to apply the label to selected versions. A checkpoint maybe created by applying a label to the entire view, or just the latest versions on the task branch, or even just recording a timestamp for the most recent checkpoint.

ClearCase - Unified Change Management (UCM)

The Unified Change Management (UCM) option for ClearCase adds higher level concepts and provides direct support for change-tasks, update, and commit operations (to a much greater degree than "view profiles" and "private branches"). A codeline may be regarded as a UCM *project* to which developers may subscribe. Each project may have an *integration stream* (where changes are merged to the codeline) and one or more *development streams*. A UCM stream is at once both a "workspace" and a "task branch" of sorts. UCM also directly supports the notion of an "*activity*",

TABLE 1-8. **Mapping of SCM pattern concepts to UCM ClearCase concepts**

SCM Pattern concept name	**UCM concept name**	**Comments**
Repository	Versioned Object Base (VOB)	
Development Workspace	development stream	cleartool mkstream
Codeline	project	*cleartool mkproject* plus an integration stream and "base"
Change Task	activity	cleartool mkactivity
Workspace Update	rebase	cleartool rebase
Task-level Commit	deliver	cleartool deliver
Task Branch	activity	an activity in a development stream
Label	baseline	cleartool mkbl
Codeline Policy	project policy settings	
Integration Workspace	integration stream	

and allows a stream to encompass more than one activity (so it need not be limited to a single task). UCM directly supports each of the SCM pattern concepts as follows:

Start a new change task by creating a new *development stream* for an associated *project* (or using an existing development stream). Then create a new *activity* to work on in that development stream as the *current activity*. UCM will capture the versions associated with that activity as its *change-set*.

Updating one's workspace corresponds to performing a "*rebase*" operation for the current activity in a development stream. Committing the changes corresponds to a "*deliver*" operation.

A UCM *project* can be configured with certain codeline policy elements, such as requiring a rebase prior to delivery.

UCM *baselines* correspond to labels and may be full or partial / incremental. Baselines may have a promotion-level associated with them to indicate the level of verification and / or quality assurance they have undergone (e.g.,*initial, rejected, built, tested, released*). Each baseline must be associated with a UCM *component* (which is a source-tree of files in a VOB).

CM Synergy

CM Synergy was once known as Continuus and is a very powerful process-centered

TABLE 1-9. Mapping of SCM pattern concepts to CM Synergy concepts

SCM Pattern concept name	CM Synergy concept name	Comments
Repository	project	also see *base model*
Development Workspace	work area	also see *working project*
Codeline	project object	
Change Task	task	
Workspace Update	update members	also see *reconfigure*
Task-level Commit	complete task	also called *checkin task*
Task Branch	task	
Label	baseline	
Codeline Policy	project template	

SCM tool that makes use of tasks, projects, and folders along with highly configurable workflow.

CM Synergy uses the terms repository and database interchangeable. A CM synergy database holds one or more projects or products. All the files for a particular system are usually mapped to a top-level project.

Workspaces in CM Synergy are called *work areas* or *working projects*. A working project is created simply by checking out the entire project into the designated work area. The work area provides a file-system-based view of a project in terms of a component source-tree of files and directories.

The logical unit of change in CM Synergy is a *task*. CM Synergy has no need to create a new branch for a task. Developers simply set up their workspace and work on their assigned task(s) in that workspace. An "*update members*" (or *reconfigure*) operation is used to update the work area's working project. In this manner, a *project object* corresponds to a codeline.

When changes for a task are finished, a *complete task* (or *checkin task*) operation commits the changes to the project for other developers to see.

CM Synergy also supports the notion of a *release stream*, which is a more formal sort of codeline corresponding to the ***RELEASE LINE*** pattern. To create a label in CM Synergy, one does a *release* (checkin) of an entire project. This captures the current configuration of a project and associates a release name with it by automatically setting the release attribute on all object members of the project.

StarTeam

StarTeam, a relative newcomer on the process oriented SCM tool scene that is getting more well known, bills itself as an easy to use tool with a configurable workflow that can read and work with VSS and PVCS VM repositories. StarTeam uses a centralized, SQL compliant database as its repository. Windows, Unix, and Web clients access *Projects* containing *folder/file hierarchies* managed by the repository. A *StarTeam View* is a set of *working folders* and *working files* in a *project* representing both workspaces and codelines. All development activities take place in a *StarTeam View*.

TABLE 1-10. **Mapping of SCM pattern concepts to StarTeam concepts**

SCM Pattern concept name	**CM Synergy concept name**	**Comments**
Repository	repository	also see *project*
Development Workspace	working files/folders	also see *view*
Codeline	project view	also see *reference view* and *branching view*
Change Task	task	
Workspace Update	check in	also see *update status*, and *view compare/ merge*
Task-level Commit	merge	
Task Branch	branching view	also see *process rules, project view* and *reference view*
Label	label	see also *view label* and *revision label*
Checkpoint	view label	

A StarTeam distinguishing feature is its ability to combine in a single tab based view, definition and linking of files, changes, requirements, tasks, and topics. So in addition to tracking and reproducing changes to files it can also track and electronically link files with *changes, tasks,* and *topics*, and with each other. StarTeam *topics* allow Usenet-style threaded conversations to occur between team members and allows the topic to be linked to tasks and other StarTeam objects.

Changes and *tasks* are the logical unit of change in StarTeam and, although it is possible, there is no need to create a new branch for a change or task. StarTeam *process rules* achieve the same goal. Process rules ensure that no file revision is created that is not also linked and pinned to a change or task. Developers need only to follow the link to check out and work on their assigned files.

The workspace, or StarTeam view is updated by checking in or merging files that have been modified. Tasks are completed by updating them. And finally, a new view label is created from the updated view and completed tasks.

A new mainline is created using File / Project / New dialog. An initial baseline workspace is created using the Folder / New dialog which relates the StarTeam folder hierarchy to the users operating system directory structure.

A new task is created by selecting theTaskTab thenTasks / NewTask dialog. The Link tool is used to associate files to the task. The linked files are edited using the File / Edit dialog or using the operating system editor.

The workspace is updated using the File / Check-in dialog, which updates the workspace and the codeline.

A revision label identifying all the files in the view is created using the View / Labels dialog and selecting revision as the label type.

The task is then updated by selecting the task and RMB / update dialog

PVCS Dimensions

PVCS Dimensions gained it’s fame under the name PCMS from SQL software. It has since undergone several new versions since being acquired by the Merant and incorporated into the PVCS product line. It is a very complete SCM tool that covers not only version management but also change management and process management.

TABLE 1-11. **Mapping of SCM pattern concepts to PVCS Dimensions concepts**

SCM Pattern concept name	PVCS Dimensions concept name	Comments
Repository	Base database	typically a Product or Project within a base-database, depending on the repository level
Development Workspace	private workset	
Codeline	named branch	also could be a workset
Change Task	work package	see also change document
Workspace Update	N/A	updates the workset with the latest versions from the codeline (also see checkin)
Task-level Commit	promote action	promotes a change-task from an implementation status to a verification status
Task Branch	work package	see also change document and named branch
Label	baseline	also see Create Revised Baseline and certain kinds of named branches
Codeline Policy	control plan	captures team development process and rules

PVCS Dimensions supports the concept of both *products* and *projects*. A product will typically map to a top-level project in the database.

A workspace in PVCS corresponds to a *workset*. Developers can share a common workset, or may be allowed to create a private workset to hold their changes.

Change-tasks are called *work packages* and may also be made to correspond to what Dimensions refers to as a *change document* using Dimensions’s object relationships and linking capabilities.

Changes are committed by *promoting* the changes in a workset from an implementation state to a verification state.

Codelines are created by making a named branch, and/or by using an integration workset into which other worksets are merged for the codeline.

Labels correspond to *baselines* that may be created for a product.

PVCS Version Manager

PVCS VM has been around a long time and, as a result, is one of the more commonly

TABLE 1-12. **Mapping of SCM pattern concepts to PVCSVersion Manager concepts**

SCM Pattern concept name	**PVCS VM concept name**	**Comments**
Repository	project database	also see *project root*
Development Workspace	workspace	also see *subproject and workfile location*
Codeline	branch	
Change Task	N/A	
Workspace Update	N/A	
Task-level Commit	check in	
Task Branch	N/A	
Label	version label	also see *baselining*
Codeline Policy	promotion groups	also see project configuration options

used commercial VC tools. It is well known for its *promotion modeling* capability which allows customers to define team development tasks, worksets, projects, and their progression from initial development of changes to increasing promotion levels of stability and quality assurance.

A PVCS *project database* can hold one or more *projects* each of which can have its own project configuration options. A *promotion model* is a set of *promotion groups*, where each promotion group defines a level or milestone in a particular development cycle.

A developer will typically create a workspace by opening a project database and selecting a project to view in the workspace, specifying the *workfile location* from which files will be checked in and out.

The workspace is populated by getting unlocked versions of the files (*get*) in the project view. A file may be edited by performing a checkout to create a writable copy. This locks the file against checkout by other developers in the project.

Typical PVCS operation really doesn't encourage parallel development for individual projects and their codelines (so there is no direct equivalent of a workspace update that reconciles changes without doing a checkin), but *variant projects* and *subprojects* may be used to create branches that correspond to alternate codelines of development.

The *checkin* operation is the way in which changes are committed to the project.

Version labels may be created for a project to represent version identifier labels.

MKS Integrity (Enterprise edition)

TABLE 1-13. Mapping of SCM pattern concepts to MKS Source Integrity concepts

SCM Pattern concept name	MKS Integrity concept name	Comments
Repository	top-level project	
Development Workspace	sandbox	
Codeline	development path	also see projects and subproject
Change Task	change package	
Workspace Update	resync	
Task-level Commit	checkin	
Task Branch	change package	
Label	project checkpoint	also labels

Source Integrity (SI) is very similar in its operation to PVCS Version Manager. SI *projects* are sets of files that are grouped together as a single body or scope of work. Projects maybe broken down into subprojects. Project *members* are source files. A workspace is called a "*sandbox*" in SI. Codelines and task-branches are called *development paths*, version identifiers are called *project checkpoints* and may be assigned project *labels*.

One of SIs most powerful features are *change packages*, which can be used to hold a single change-task, or a collection of changes to move back and forth between a master project and a development path for a variant project (and vice versa).

A sandbox is created and associated with a project and the latest versions of its members. Files are checked out in the sandbox as needed. Checkouts lock the file in the project as with PVCS VM.

The sandbox may be updated via a *resync* command, and all the changes made in the sandbox may be grouped into a change package if desired.

Eventually, a *checkin* command commits the changes to the project.

Codelines are created by making a variant project of a master project and associating it with a development path.

Checkpointing a project creates a new version identifier for the last state of the project. Labels may be created explicitly and associated with a checkpoint or with the current versions in a project.

Further Reading

See the vendor home pages mentioned above for the definitive source of information on all of these tools. Also see the following:

- The book *Essential SourceSafe* by Ted Roche and Larry C. Whipple (Roche and Whipple 2001) is a basic guide for VSS installation, administration, and usage
- VSS technical FAQ and resources at **http://msdn.microsoft.com/ssafe/technical/** (and an unofficial FAQ located at ***http://www.michaelis.net/SourceSafe/Faq.htm***)
- *Open Source Development with CVS* (Fogel and Bar 2001) is an excellent guide to using CVS
- *Practical Software Configuration Management* (Mikkelsen and Pherigo 1997) also covers some aspects of CVS usage, but is not as recent as Fogel's book.
- Pascal Molli maintains an excellent collection of CVS related links and information at **http://www.loria.fr/~molli/cvs-index.html**
- Bryan White's *Software Configuration Management Strategies and Rational ClearCase* (White 2000) provides an overview of UCM
- Christian Goetze maintains an "unofficial" ClearCase FAQ at **http://www.cg-soft.com/faq/clearcase.html**
- Alexis Leon's *Guide to Software Configuration Management* (Leon 2000) has an overview of several dozen SCM tools in its appendices.
- Susan Dart's *Configuration Management: The Missing Link in Web Engineering* (Dart 1992) contains a great deal of information, including several sections on evaluating and selecting CM tools for the enterprise

Chapter D
Photo Credits

Page 11	Photo by Russel Lee, May 1938. Library of Congress, Prints & Photographs Division, FSA-OWI Collection, Reproduction number LC-USF33-011474-M3 DLC

Page 25	Photo by Arthur Rothstein. Library of Congress, Prints & Photographs Division, FSA-OWI Collection LC-USF34-024346-D (b&w film neg.)

Page 37	Photo by Russel Lee, May 1938. Library of Congress, Prints & Photographs Division, FSA-OWI Collection, Reproduction Number: LC-USF33-011692-M4 (b&w film neg.)

Page 55	Photo by John Vachon. Library of Congress, Prints & Photographs Division, FSA-OWI Collection, Reproduction number LC-USF34-064602-D DLC (b&w film neg.)

Page 63	Library of Congress, Prints and Photographs Division, Detroit Publishing Company Collection. Reproduction number. LC-D418-31625 DLC (b&w glass neg.)

Page 71	Photo by David Meyers. Library of Congress, Prints & Photographs Division, FSA-OWI Collection Reproduction Number LC-USF33-015598-M2 (b&w film neg.).

Page 83	Photo by Russell Lee. Library of Congress, Prints & Photographs Division, FSA-OWI Collection Reproduction Number LC-USF33-013141-M1 (b&w film neg.)

Page 91	Photo: MBDMOTI+EC001, Modern Times with Charlie Chaplin. Reprinted with permission of Everett Collection, Inc.

Page 101	Photo by Alfred T Palmer Library of Congress, Prints & Photographs Division, FSA-OWI Collection Reproduction Number LC-USW361-138 (color film copy slide)

Page 107	Photo by John Vachon. Library of Congress, Prints & Photographs Division, FSA-OWI Collection, reproduction number LC-USF34-061836-D (b&w film neg.)

Page 115 Photo by Alfred T Palmer. Library of Congress, Prints & Photographs Division, FSA-OWI Collection, Reproduction number LC-USE6-D-000162 (b&w film neg.).

Page 119 Photo By John Collier. Library of Congress, Prints & Photographs Division, FSA-OWI Collection, LC-USF34-084002-C (b&w film neg.)

Page 125 Photo by Lee Russell. Library of Congress, Prints & Photographs Division, FSA-OWI Collection,Reproduction Number LC-USF33-011632-M3 (b&w film neg.)

Page 129 Photo by Alfred T. Palmer. Library of Congress, Prints & Photographs Division, FSA-OWI Collection, Reproduction Number LC-USE6-D-005032 (b&w film neg.)

Page 133 Photo by Alfred T. Palmer. Library of Congress, Prints & Photographs Division, FSA-OWI Collection, Reproduction Number LC-USE6-D-007389 (b&w film neg.)

Page 137 Photo by Marjory Collins, Library of Congress, Prints & Photographs Division, FSA-OWI Collection Reproduction Number LC-USW3-009019-D (b&w film neg.)

Page 143 Photo by Jack Delano, Library of Congress, Prints & Photographs Division, FSA-OWI Collection,LC-USW3-014014-E (b&w film neg.)

Page 149 Photo by Jack Delano. Library of Congress, Prints & Photographs Division, FSA-OWI Collection, Reproduction Number LC-USW3-012717-D (b&w film neg.)

Page 153 Photo By John Collier. Library of Congress, Prints & Photographs Division, FSA-OWI Collection, Reproduction number, LC-USW3-010723-C (b&w film neg.)

Chapter E
About the Photos

Chapter 1 (page 11): Barn erection. View of roofing operation from beneath, showing construction of the roof system. Southeast Missouri Farms Project.

Chapter 2 (page 25): Elevated structure and buildings. Lower Manhattan,December, 1941

Chapter 3 (page 37): Patterns of tools painted on wall for easy identification, Lake Dick Project, Arkansas, Sept 1938.

Chapter 4 (page 55): Bowdle, South Dakota. On the main line. Feb 1942.

Chapter 5 (page 63): Construction, Grand Central Terminal, New York, N.Y. bet 1905 & 1915.

Chapter 6 (page 71): Washington, D.C., 1939 A government clerk's room, showing a desk with books, telephone and directory, and a desk lamp on it.

Chapter 7 (page 83): Cases of canned salmon in warehouse, Astoria, Oregon. September, 1941.

Chapter 8 (page 91): Charlie Chaplin in Modern Times.

Chapter 9 (page 101): Making wiring assemblies at a junction box on the fire wall for the right engine of a B-25 bomber, North American Aviation, Inc., [Inglewood], Calif. July, 1942.

Chapter 10 (page 107): Man who operates small grocery store and secondhand furniture store in his home. Chanute, Kansas.November, 1940.

Chapter 11 (page 115): Proud of his job. Smiling worker in an eastern arsenal hand finishes the interior surface of a cradle for an 8-inch gun, railway carriage. 1942.

Chapter 12 (page 119): Richwood, West Virginia. Louise Thompson, daughter of a newspaper editor in Richwood. She is a printer’s devil. September, 1942.

Chapter 13 (page 125): Fighting fire of rice straw stack in rice field near Crowley, Louisiana.1938 Sept.

Chapter 14 (page 129): All the parts of an airplane engine, which has just undergone severe tests in a Midwest plant, are spread out for minute inspection. Continental Motors, Michigan. Feburary, 1942.

Chapter 15 (page 133): An experimental scale model of the B-25 plane is prepared for wind tunnel tests in the Inglewood, California plant of North American Aviation, Incorporated. October, 1942

Chapter 16 (page 137): New York, New York. "Morgue" of the New York Times newspaper. Clippings on every conceivable subject are filed here for a reference. Editors and writers phone in for information. September, 1942.

Chapter 17 (page 143): January, 1943. Freight operations on the Chicago and Northwestern Railroad between Chicago and Clinton, Iowa. The rear brakeman signals the engineer to test the brakes by applying and releasing them. This is the signal for "apply".

Chapter 18 (page 149): hicago, Illinois. Train pulling out of a freight house at a Chicago and Northwestern Railroad yard. The wooden trestle is part of a long chain belt used to carry blocks of ice from the ice house to the freight house.December, 1942.

Chapter 19 (page 153): Pittsburgh, Pennsylvania (vicinity). Montour no. 4 mine of the Pittsburgh Coal Company. There are miles and miles of track in a mine and the maintenance of the roadbed, ballast and switches keeps a crew working constantly. November 1942.

Production Reference Materials

List of Figures

Repository 83

How do you build get the right versions of the right components into a new workspace? 83
Have a single point of access, or a Repository, for your code and related artifacts. Make creating a developer workspace as simple and as transparent as possible. 86

Private System Build 91

How do you verify that your changes do not break the build or the system before you check them in. 91
Before making a submission to source control, build the system using a Private System Build that is similar to the nightly build. 94

Integration Build 101

How do you make sure that the code base always builds reliably? 101
Be sure that all changes (and their dependencies) are built using a central integration build process. 104

Third Party Codeline 107

What is the most effective strategy to coordinate versions of vendor code with versions of product code? 108
Create a codeline for third party code. Build workspaces and installation kits from this codeline. 110

Task Level Commit 115

How much work should you do between submission to the Version Control System? How long should you wait before checking files in? 115
Do one commit per small grained, consistent task. 117

Codeline Policy 119

How do the developers know which codeline to check their code into, and when to when to check it in, and what tests to run before check in? 119
For each branch or codeline, formulate a policy that determines how and when developers should make changes. The policy should be concise and auditable. 122

Smoke Test 125

How do you know that the system will still work after you make a change? 125
Subject each build to a smoke test that verifies that the application has not broken in an obvious way. 126

Unit Test 129

How do you test whether a module still works as it should after making a change? 129
Develop and Run Unit Tests. 131

Regression Test 133

How do you ensure that existing code doesn't get worse as you make other improvements? 133
Run regression tests on the system whenever you want to ensure stability of the codeline, such as before you release a build, or before a particularly risky change. Create the regression tests from test cases that the system has failed in the past. 135

Private Versions 137

How can you experiment with a complex change and benefit from the version control system without making the change public? 137
Provide developers with a mechanism for check pointing changes at a granularity that they are comfortable with. This can be provided for by a local revision control area, Only stable code sets are checked into the project repository 140

Release Line 143

How do you do maintence on released versions without interfering with your current development work? 143
Split maintence / release and active development into separate codelines. Keep each released version on a release line. Allow the line to progress on its own for bug fixes. Branch each release off of the mainline. 146

Release-Prep Code Line 149

How do you stabilize a codeline for an impending release while also allowing new work to continue on an active codeline? 149
Create a release-engineering branch when code is approaching release quality. Finish up the release on this branch, and leave the mainline for active development. The branch becomes the release branch. 151

Task Branch 153

How can your team make multiple, long term overlapping changes to a codeline without compromising its consistency and integrity? 153
Fork off a separate branch for each activity that has significant changes for a codeline. 156

AU 1
AU 2
BL 104
BL 104
BL 106
BL 110
BL 110
BL 111
BL 111
BL 117
BL 117
BL 12
BL 122
BL 122
BL 122
BL 122
BL 122
BL 127
BL 127
BL 131
BL 131
BL 131
BL 132
BL 135
BL 168
BL 168
BL 168
BL 169
BL 169
BL 169
BL 172
BL 172
BL 172
BL 172
BL 172
BL 172
BL 174
BL 18

BL 192
BL 192
BL 192
BL 192
BL 192
BL 192
BL 192
BL 192
BL 192
BL 2
BL 2
BL 2
BL 20
BL 20
BL 20
BL 20
BL 20
BL 21
BL 21
BL 21
BL 21
BL 21
BL 23
BL 23
BL 24
BL 27
BL 27
BL 28
BL 28
BL 28
BL 29
BL 31
BL 32
BL 32
BL 33
BL 34
BL 34
BL 34

BL 34
BL 34
BL 35
BL 35
BL 35
BL 35
BL 36
BL 36
BL 36
BL 36
BL 40
BL 40
BL 42
BL 42
BL 42
BL 42
BL 42
BL 42
BL 43
BL 43
BL 43
BL 45
BL 45
BL 45
BL 46
BL 46
BL 46
BL 46
BL 46
BL 46
BL 46
BL 46
BL 46
BL 50
BL 50
BL 50
BL 50
BL 50

BL 61
BL 61
BL 61
BL 62
BL 62
BL 67
BL 67
BL 67
BL 76
BL 76
BL 77
BL 77
BL 77
BL 77
BL 77
BL 77
BL 77
BL 77
BL 8
BL 8
BL 8
BL 8
BL 8
BL 81
BL 84
BL 84
BL 84
BL 84
BL 84
BL 87
BL 87
BL 94
BL 96
BL1 104
BL1 105
BL1 106
BL1 110
BL1 117

BL1 12
BL1 12
BL1 122
BL1 122
BL1 123
BL1 127
BL1 128
BL1 131
BL1 131
BL1 132
BL1 135
BL1 136
BL1 147
BL1 167
BL1 167
BL1 168
BL1 168
BL1 168
BL1 168
BL1 168
BL1 169
BL1 169
BL1 169
BL1 169
BL1 170
BL1 170
BL1 172
BL1 174
BL1 174
BL1 175
BL1 175
BL1 175
BL1 176
BL1 176
BL1 18
BL1 2
BL1 20
BL1 20

BL1 21
BL1 23
BL1 23
BL1 26
BL1 28
BL1 29
BL1 31
BL1 32
BL1 32
BL1 34
BL1 34
BL1 35
BL1 36
BL1 40
BL1 42
BL1 43
BL1 45
BL1 46
BL1 50
BL1 60
BL1 61
BL1 61
BL1 62
BL1 67
BL1 69
BL1 7
BL1 72
BL1 76
BL1 77
BL1 77
BL1 84
BL1 87
BL1 89
BL1 94
BL1 96
BL1 96
BL1 97
BL1 98

Body 1
Body 1
Body 1
Body 1
Body 1
Body 1
Body 1
Body 1
Body 1
Body 1
Body 1
Body 1
Body 101
Body 102
Body 102
Body 103
Body 103
Body 103
Body 104
Body 105
Body 105
Body 105
Body 105
Body 106
Body 106
Body 106
Body 106
Body 106
Body 108
Body 108
Body 108
Body 109
Body 109
Body 109
Body 11
Body 110
Body 110
Body 110

Body 111
Body 111
Body 111
Body 112
Body 112
Body 112
Body 112
Body 112
Body 112
Body 113
Body 113
Body 113
Body 113
Body 113
Body 114
Body 115
Body 116
Body 116
Body 116
Body 116
Body 116
Body 117
Body 117
Body 117
Body 117
Body 117
Body 118
Body 118
Body 118
Body 118
Body 118
Body 118
Body 119
Body 12
Body 12
Body 12
Body 120
Body 120

Body 121
Body 121
Body 121
Body 122
Body 122
Body 122
Body 122
Body 122
Body 123
Body 123
Body 123
Body 125
Body 126
Body 126
Body 126
Body 127
Body 127
Body 127
Body 127
Body 127
Body 127
Body 127
Body 128
Body 128
Body 128
Body 128
Body 128
Body 128
Body 128
Body 129
Body 130
Body 130
Body 130
Body 131
Body 131
Body 131
Body 131
Body 131

Body 131
Body 131
Body 132
Body 132
Body 133
Body 134
Body 134
Body 134
Body 134
Body 134
Body 134
Body 135
Body 135
Body 135
Body 135
Body 136
Body 136
Body 136
Body 137
Body 138
Body 138
Body 139
Body 139
Body 139
Body 139
Body 14
Body 14
Body 14
Body 140
Body 140
Body 141
Body 141
Body 141
Body 143
Body 144
Body 144
Body 145
Body 145

Body 145
Body 145
Body 145
Body 146
Body 147
Body 147
Body 147
Body 149
Body 15
Body 15
Body 15
Body 15
Body 15
Body 150
Body 150
Body 150
Body 151
Body 151
Body 151
Body 151
Body 151
Body 152
Body 152
Body 152
Body 153
Body 154
Body 154
Body 154
Body 155
Body 155
Body 156
Body 156
Body 156
Body 157
Body 157
Body 159
Body 159
Body 159

Body 159
Body 159
Body 159
Body 16
Body 16
Body 16
Body 16
Body 16
Body 167
Body 167
Body 167
Body 168
Body 168
Body 168
Body 168
Body 169
Body 169
Body 169
Body 17
Body 170
Body 171
Body 172
Body 173
Body 173
Body 173
Body 173
Body 173
Body 174
Body 174
Body 174
Body 174
Body 174
Body 175
Body 175
Body 176
Body 176
Body 177
Body 177

Body 177
Body 177
Body 178
Body 178
Body 178
Body 178
Body 178
Body 178
Body 179
Body 179
Body 179
Body 18
Body 18
Body 18
Body 180
Body 180
Body 180
Body 180
Body 180
Body 180
Body 180
Body 180
Body 181
Body 181
Body 181
Body 181
Body 182
Body 182
Body 182
Body 182
Body 182
Body 182
Body 182
Body 183
Body 183
Body 183
Body 184
Body 184

Body 184
Body 184
Body 184
Body 184
Body 184
Body 184
Body 184
Body 184
Body 184
Body 185
Body 185
Body 185
Body 185
Body 186
Body 186
Body 186
Body 186
Body 186
Body 187
Body 187
Body 187
Body 187
Body 188
Body 188
Body 188
Body 188
Body 188
Body 188
Body 188
Body 189
Body 189
Body 189
Body 189
Body 19
Body 19
Body 19
Body 19
Body 19

Body 19
Body 19
Body 190
Body 190
Body 190
Body 190
Body 190
Body 190
Body 190
Body 191
Body 191
Body 191
Body 191
Body 191
Body 191
Body 192
Body 192
Body 192
Body 192
Body 192
Body 192
Body 2
Body 2
Body 2
Body 2
Body 20
Body 20
Body 21
Body 21
Body 21
Body 22
Body 22
Body 22
Body 22
Body 22
Body 22
Body 23
Body 23

Body 23
Body 23
Body 23
Body 25
Body 25
Body 26
Body 26
Body 26
Body 27
Body 27
Body 27
Body 28
Body 29
Body 29
Body 29
Body 3
Body 30
Body 30
Body 30
Body 30
Body 31
Body 31
Body 31
Body 31
Body 31
Body 32
Body 32
Body 32
Body 32
Body 32
Body 33
Body 33
Body 33
Body 33
Body 33
Body 34
Body 34
Body 34

Body 34
Body 35
Body 35
Body 36
Body 37
Body 37
Body 38
Body 38
Body 38
Body 38
Body 38
Body 38
Body 38
Body 39
Body 39
Body 39
Body 39
Body 39
Body 4
Body 40
Body 40
Body 40
Body 41
Body 41
Body 41
Body 41
Body 41
Body 41
Body 41
Body 42
Body 42
Body 42
Body 42
Body 43
Body 43
Body 43
Body 44
Body 44

Body 44
Body 44
Body 44
Body 45
Body 45
Body 45
Body 46
Body 46
Body 48
Body 5
Body 5
Body 5
Body 50
Body 50
Body 50
Body 56
Body 56
Body 56
Body 57
Body 57
Body 57
Body 57
Body 58
Body 58
Body 59
Body 59
Body 59
Body 6
Body 6
Body 60
Body 60
Body 60
Body 60
Body 60
Body 60
Body 61
Body 61
Body 61

Body 64
Body 64
Body 64
Body 64
Body 64
Body 65
Body 65
Body 65
Body 65
Body 66
Body 66
Body 67
Body 67
Body 67
Body 67
Body 67
Body 68
Body 68
Body 68
Body 68
Body 68
Body 69
Body 69
Body 69
Body 69
Body 69
Body 69
Body 7
Body 72
Body 72
Body 72
Body 72
Body 72
Body 74
Body 74
Body 74
Body 75
Body 75

Body 75
Body 76
Body 76
Body 77
Body 77
Body 77
Body 77
Body 78
Body 78
Body 78
Body 78
Body 78
Body 78
Body 79
Body 79
Body 79
Body 79
Body 79
Body 79
Body 79
Body 79
Body 80
Body 80
Body 80
Body 83
Body 84
Body 84
Body 85
Body 85
Body 85
Body 85
Body 85
Body 85
Body 87
Body 87
Body 87
Body 88
Body 88

Body 88
Body 89
Body 89
Body 89
Body 92
Body 92
Body 92
Body 93
Body 93
Body 94
Body 95
Body 95
Body 95
Body 96
Body 96
Body 96
Body 97
Body 97
Body 97
Body 97
Body 97
Body 97
Body 97
Body 98
Body 98
Body 98
Body 98
Body 98
Body ix
Body ix
Body ix
Body ix
Body x
Body x
Body x
Body xi
Body xi
Body xi

Body xi
Body xi
Body xii
Body xii
Body xii
Body xii
Body xii
Body xiii
Body xiii
Body xiii
Body xiii
Body xiii
Body xiii
Body xiv
Body xiv
Body xiv
Body xiv
Body xv
Body xv
Body xv
BT 1
BX 104
BX 105
BX 117
BX 12
BX 12
BX 12
BX 122
BX 123
BX 127
BX 128
BX 131
BX 131
BX 132
BX 135
BX 136
BX 147
BX 168

BX 174
BX 18
BX 2
BX 20
BX 21
BX 21
BX 23
BX 24
BX 27
BX 29
BX 31
BX 32
BX 33
BX 34
BX 34
BX 35
BX 36
BX 40
BX 42
BX 43
BX 46
BX 47
BX 51
BX 60
BX 61
BX 61
BX 62
BX 67
BX 69
BX 72
BX 77
BX 77
BX 77
BX 8
BX 81
BX 84
BX 89
BX 94

BX 96
BX 96
BX 96
BX 97
BX 99
ContextBody 101
ContextBody 107
ContextBody 115
ContextBody 119
ContextBody 125
ContextBody 129
ContextBody 133
ContextBody 137
ContextBody 143
ContextBody 149
ContextBody 153
ContextBody 55
ContextBody 63
ContextBody 71
ContextBody 83
ContextBody 91
EX 156
EX 16
EX 184
EX 32
EX 38
EX 38
EX 39
Figure 103
Figure 105
Figure 108
Figure 111
Figure 120
Figure 138
Figure 139
Figure 144
Figure 145
Figure 145

Figure 147
Figure 151
Figure 154
Figure 155
Figure 156
Figure 29
Figure 3
Figure 4
Figure 45
Figure 46
Figure 49
Figure 5
Figure 5
Figure 56
Figure 57
Figure 6
Figure 60
Figure 65
Figure 65
Figure 65
Figure 67
Figure 68
Figure 73
Figure 74
Figure 75
Figure 84
Figure 87
Figure 88
Figure 92
Figure 95
FTN 2
FTN 45
HA 1
HA 1
HA 1
HA 101
HA 107
HA 11

HA 115
HA 119
HA 125
HA 129
HA 133
HA 137
HA 143
HA 149
HA 153
HA 159
HA 161
HA 167
HA 171
HA 25
HA 37
HA 55
HA 63
HA 71
HA 83
HA 91
HB 1
HB 1
HB 1
HB 1
HB 101
HB 107
HB 11
HB 115
HB 119
HB 125
HB 129
HB 133
HB 137
HB 143
HB 149
HB 153
HB 159
HB 161

HB 167
HB 171
HB 25
HB 37
HB 55
HB 63
HB 71
HB 83
HB 91
HB iii
HB ix
HB xiii
HB xv
HC 1
HC 104
HC 106
HC 106
HC 110
HC 113
HC 113
HC 117
HC 118
HC 12
HC 122
HC 123
HC 123
HC 126
HC 128
HC 128
HC 131
HC 132
HC 132
HC 135
HC 136
HC 14
HC 140
HC 147
HC 15

HC 151
HC 152
HC 153
HC 156
HC 159
HC 159
HC 161
HC 173
HC 175
HC 177
HC 179
HC 18
HC 181
HC 183
HC 185
HC 186
HC 187
HC 189
HC 19
HC 190
HC 191
HC 192
HC 22
HC 22
HC 22
HC 23
HC 23
HC 25
HC 27
HC 30
HC 30
HC 33
HC 35
HC 36
HC 38
HC 39
HC 40
HC 42

HC 42
HC 44
HC 50
HC 50
HC 51
HC 59
HC 6
HC 61
HC 62
HC 66
HC 69
HC 69
HC 7
HC 76
HC 80
HC 81
HC 86
HC 89
HC 89
HC 94
HC 98
HC 98
HC x
HC xi
HC xi
HC xii
HC xii
HC xiii
HD 167
HD 167
HD 168
HD 168
HD 168
HD 168
HD 169
HD 169
HD 169
HD 169

HD 169
HD 169
HD 169
HD 169
HD 170
Indented2 171
INFTC 171
INFTC 171
INFTC 171
INFTC 171
INFTC 171
INFTC 171
INFTC 171
INFTC 171
INFTC 171
INFTC 171
INFTC 171
INFTC 171
INFTC 171
INFTC 171
INFTC 171
INFTC 172
INFTC 172
INFTC 172
INFTC 172
INFTC 172
INFTC 172
INFTC 173
INFTC 173
INFTC 173
INFTC 173
INFTC 173
INFTC 173
INFTC 173
INFTC 173
INFTC 173
INFTC 173
INFTC 173

INFTC 173
INFTC 173
INFTC 173
INFTC 173
INFTC 173
INFTC 173
INFTC 173
INFTC 173
INFTC 173
INFTC 173
INFTC 173
INFTC 173
INFTC 173
INFTC 175
INFTC 175
INFTC 175
INFTC 175
INFTC 175
INFTC 175
INFTC 175
INFTC 175
INFTC 175
INFTC 175
INFTC 175
INFTC 175
INFTC 175
INFTC 175
INFTC 175
INFTC 175
INFTC 175
INFTC 175
INFTC 175
INFTC 175
INFTC 175
INFTC 175
INFTC 175
INFTC 175
INFTC 175

INFTC 175
INFTC 175
INFTC 177
INFTC 177
INFTC 177
INFTC 177
INFTC 177
INFTC 177
INFTC 177
INFTC 177
INFTC 177
INFTC 177
INFTC 177
INFTC 177
INFTC 177
INFTC 177
INFTC 177
INFTC 177
INFTC 177
INFTC 177
INFTC 177
INFTC 177
INFTC 177
INFTC 177
INFTC 177
INFTC 177
INFTC 179
INFTC 179
INFTC 179
INFTC 179
INFTC 179
INFTC 179
INFTC 179
INFTC 179
INFTC 179
INFTC 179
INFTC 179
INFTC 179

INFTC 179
INFTC 179
INFTC 179
INFTC 179
INFTC 179
INFTC 179
INFTC 179
INFTC 179
INFTC 179
INFTC 179
INFTC 179
INFTC 179
INFTC 179
INFTC 179
INFTC 179
INFTC 179
INFTC 179
INFTC 181
INFTC 181
INFTC 181
INFTC 181
INFTC 181
INFTC 181
INFTC 181
INFTC 181
INFTC 181
INFTC 181
INFTC 181
INFTC 181
INFTC 181
INFTC 181
INFTC 181
INFTC 181
INFTC 181
INFTC 181
INFTC 181
INFTC 181
INFTC 181

INFTC 181
INFTC 181
INFTC 181
INFTC 181
INFTC 181
INFTC 181
INFTC 183
INFTC 183
INFTC 183
INFTC 183
INFTC 183
INFTC 183
INFTC 183
INFTC 183
INFTC 183
INFTC 183
INFTC 183
INFTC 183
INFTC 183
INFTC 183
INFTC 183
INFTC 183
INFTC 183
INFTC 183
INFTC 183
INFTC 183
INFTC 183
INFTC 183
INFTC 183
INFTC 183
INFTC 183
INFTC 185
INFTC 185
INFTC 185
INFTC 185
INFTC 185
INFTC 185
INFTC 185

INFTC 185
INFTC 185
INFTC 185
INFTC 185
INFTC 185
INFTC 185
INFTC 185
INFTC 185
INFTC 185
INFTC 185
INFTC 185
INFTC 185
INFTC 185
INFTC 185
INFTC 185
INFTC 185
INFTC 185
INFTC 185
INFTC 185
INFTC 185
INFTC 185
INFTC 185
INFTC 185
INFTC 186
INFTC 186
INFTC 186
INFTC 186
INFTC 186
INFTC 186
INFTC 186
INFTC 186
INFTC 186
INFTC 186
INFTC 186
INFTC 186
INFTC 186
INFTC 186
INFTC 186

INFTC 186
INFTC 186
INFTC 186
INFTC 186
INFTC 186
INFTC 186
INFTC 186
INFTC 186
INFTC 186
INFTC 186
INFTC 186
INFTC 186
INFTC 187
INFTC 187
INFTC 187
INFTC 187
INFTC 187
INFTC 187
INFTC 187
INFTC 187
INFTC 187
INFTC 187
INFTC 187
INFTC 187
INFTC 187
INFTC 187
INFTC 187
INFTC 187
INFTC 187
INFTC 187
INFTC 187
INFTC 187
INFTC 187
INFTC 187
INFTC 187
INFTC 187
INFTC 187
INFTC 187

INFTC 187
INFTC 189
INFTC 189
INFTC 189
INFTC 189
INFTC 189
INFTC 189
INFTC 189
INFTC 189
INFTC 189
INFTC 189
INFTC 189
INFTC 189
INFTC 189
INFTC 189
INFTC 189
INFTC 189
INFTC 189
INFTC 189
INFTC 189
INFTC 189
INFTC 189
INFTC 189
INFTC 189
INFTC 189
INFTC 189
INFTC 189
INFTC 189
INFTC 190
INFTC 190
INFTC 190
INFTC 190
INFTC 190
INFTC 190
INFTC 190
INFTC 190
INFTC 190
INFTC 190

INFTC 190
INFTC 190
INFTC 190
INFTC 190
INFTC 190
INFTC 190
INFTC 190
INFTC 190
INFTC 190
INFTC 190
INFTC 190
INFTC 190
INFTC 190
INFTC 190
INFTC 190
INFTC 190
INFTC 190
INFTC 191
INFTC 191
INFTC 191
INFTC 191
INFTC 191
INFTC 191
INFTC 191
INFTC 191
INFTC 191
INFTC 191
INFTC 191
INFTC 191
INFTC 191
INFTC 191
INFTC 191
INFTC 191
INFTC 191
INFTC 191
INFTC 191
INFTC 191
INFTC 191

INFTC 191
INFTC 191
INFTC 191
INFTC 7
INFTC 7
INFTC 7
INFTC 7
INFTC 7
INFTC 7
INFTC 7
INFTC 7
INFTC 7
INFTC 7
INFTC 7
INFTC 7
INFTC 7
INFTH 171
INFTH 171
INFTH 171
INFTH 171
INFTH 173
INFTH 173
INFTH 173
INFTH 173
INFTH 175
INFTH 175
INFTH 175
INFTH 175
INFTH 177
INFTH 177
INFTH 177
INFTH 177
INFTH 179
INFTH 179
INFTH 179
INFTH 179
INFTH 181
INFTH 181

SolutionSummary 66
SolutionSummary 76
SolutionSummary 86
SolutionSummary 94
StoryBody 104
StoryBody 109
StoryBody 109
StoryBody 109
StoryBody 121
StoryBody 126
StoryBody 126
StoryBody 13
StoryBody 13
StoryBody 13
StoryBody 130
StoryBody 135
StoryBody 140
StoryBody 146
StoryBody 146
StoryBody 150
StoryBody 17
StoryBody 17
StoryBody 30
StoryBody 58
StoryBody 58
StoryBody 58
StoryBody 58
StoryBody 59
StoryBody 66
StoryBody 76
StoryBody 86
StoryBody 86
StoryBody 86
StoryBody 86
StoryBody 86
StoryBody 93
StoryBody 93
StoryBody 94

StoryBody-last 104
StoryBody-last 110
StoryBody-last 121
StoryBody-last 126
StoryBody-last 13
StoryBody-last 130
StoryBody-last 135
StoryBody-last 140
StoryBody-last 146
StoryBody-last 150
StoryBody-last 17
StoryBody-last 30
StoryBody-last 59
StoryBody-last 66
StoryBody-last 76
StoryBody-last 86
StoryBody-last 94
StoryHeader 104
StoryHeader 109
StoryHeader 121
StoryHeader 126
StoryHeader 13
StoryHeader 130
StoryHeader 135
StoryHeader 140
StoryHeader 146
StoryHeader 150
StoryHeader 17
StoryHeader 30
StoryHeader 58
StoryHeader 66
StoryHeader 76
StoryHeader 86
StoryHeader 93

Images/covera0a-10069569.jpg @ 150 dpi 1
Images/fsa8a22891r.jpg @ 150 dpi 11
Images/fsa8b16069r.jpg @ 150 dpi 25
Images/fsa8a23877r.jpg @ 150 dpi 37
Images/LC-USF34-064602-D.jpg @ 150 dpi 55
Images/4a27519r.jpg @ 150 dpi 63
Images/fsa8a30508r.jpg @ 150 dpi 83
Images/MBDMOTI_EC001_H.jpg @ 400 dpi 91
Images/fsa1a35283r.jpg @ 150 dpi 101
Images/fsa8c18321r.jpg @ 150 dpi 107
Images/fsa8b08975r.jpg @ 150 dpi 115
Images/LC-USF34- 084002-C.jpg @ 150 dpi 119
Images/fsa8a23613r.jpg @ 150 dpi 125
Images/fsa8b03594r.jpg @ 150 dpi 129
Images/fsa8b04999r.jpg @ 150 dpi 133
Images/fsa8d08930r.jpg @ 150 dpi 137
Images/8d24440r.jpg @ 200 dpi 143
Images/fsa8d11291r.jpg @ 150 dpi 149
Images/fsa8d09852r.jpg @ 150 dpi 153

- Adaptation/anticipation note from pp214-215. 14
- Add Workspace to the diagram. Perhaps in the middle. 29
- TTWOB P 183 38
- TTWOB p 261 38
- TTWOB p 253 39
- (Rapid Development p 407) 68
- Rework this paragraph...("You can now easily") 111
- Need more examples of a change tasks (and what is not a change task) 117
- Should these be pattern refs or moved into the body. 118
- add citation for Make, ANT, etc. 123
- McConnel Quote is from page 407. 127
- Elaborate on the criteria. 127
- Look at XP email comments. Add info here. 139
- Show a tree with third party code etc. (new Figure) 147
- To Do: Look at descriptions in The Patterns Almanac! 159