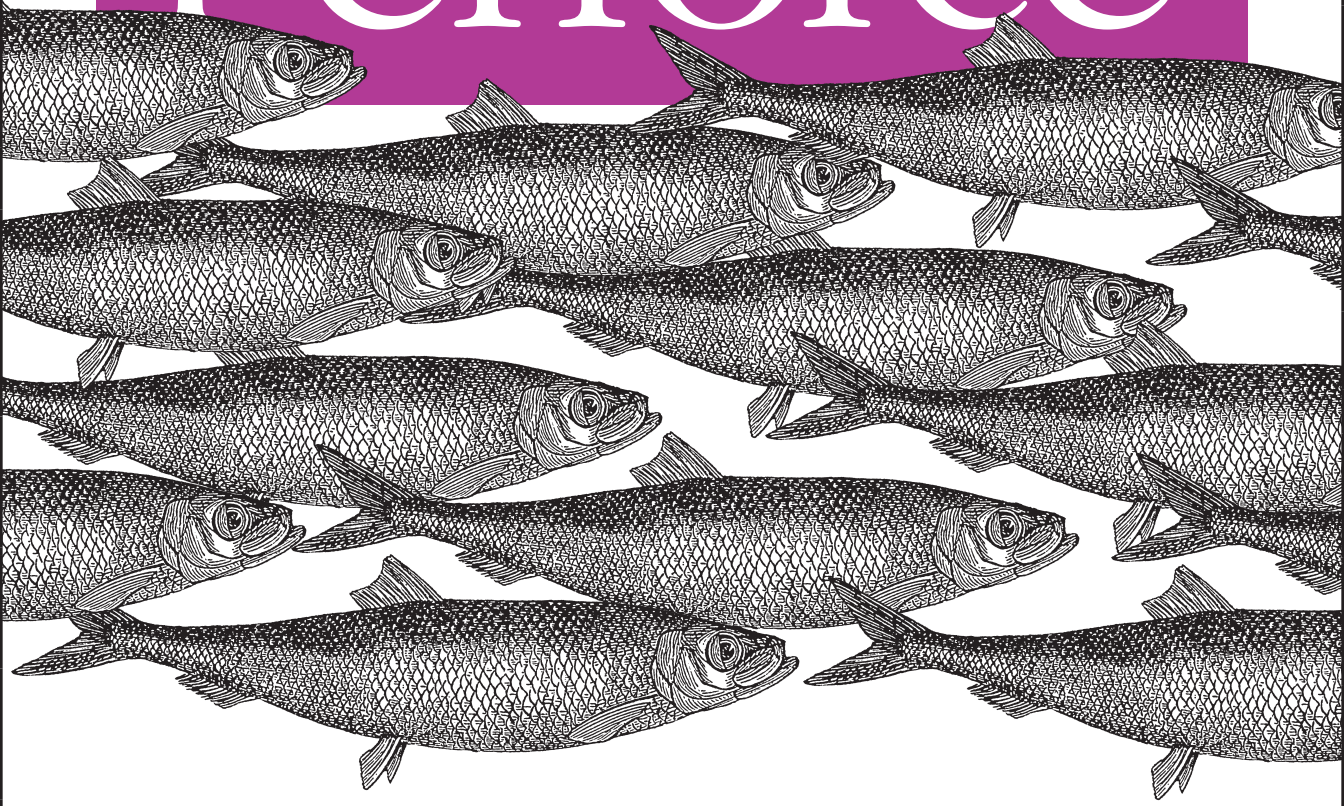*Channelling the Flow of Change
in Software Development Collaboration*

# Practical Perforce

*Laura Wingerd*

# How Software Evolves

Just as there's more to driving than knowing how to operate a car, there's more to SCM than knowing how to use an SCM tool. Mastering SCM starts with understanding how software evolves and recognizing how team collaboration, defect management, parallel releases, and distributed development affect the software life cycle. For just as road maps and rules of the road are the bigger part of driving, the software life cycle is the bigger part of SCM.

In this chapter we take a step back from Perforce to look at the roadmap of the software life cycle, the mainline model. We'll identify the codelines that form the mainline model and describe the rules of the road for change flowing between them. This chapter sets the stage for the chapters that follow, each of which demonstrates using Perforce to manage codelines of a particular type.

## The Story of Ace Engineering

Consider the story of Ace Engineering, a fictitious software company. After a year of intensive start-up development, the company introduced a new product, AcePack1. 0. Sales were successful, the customer base grew. Alas, so did the bug report database. Within six months Ace had produced a point release—essentially the same product but with many bug fixes and small enhancements—as AcePack1.1. For a while, the company supported customers on either version, but at the end of the second year it announced that AcePack 1.0 was being discontinued.

During this time Ace developers had started working on two new features, code-named Saturn and Pluto. The plan had been to include both features in the AcePack2.0 release. Midway through the third year Pluto was done, but Saturn turned out to be much more work than anticipated. (In fact, the company ended up doubling the size of the Saturn development team, with half the team working on an unforseen adjunct now code-named Saturn Plus.) AcePack2.0 was ultimately released without the Saturn feature.

The 2.0 release did well, although it, too, had its share of problems. It was hard to get customers to upgrade, and Ace ended up having to fix show-stopping bugs in both available releases, 1.1 and 2.0. However, it was able to produce a very stable release, AcePack2.1, about six months later. Shortly after that, support for AcePack1.1 was discontinued.

Finally the Saturn and Saturn Plus development projects were completed. The Saturn feature will be in AcePack3.0. Meanwhile, customers have upgraded to AcePack 2.1, AcePack 2.0 has been retired, and Ace developers have started working on yet another new feature, codenamed Rocket. (Figure 7-1.)
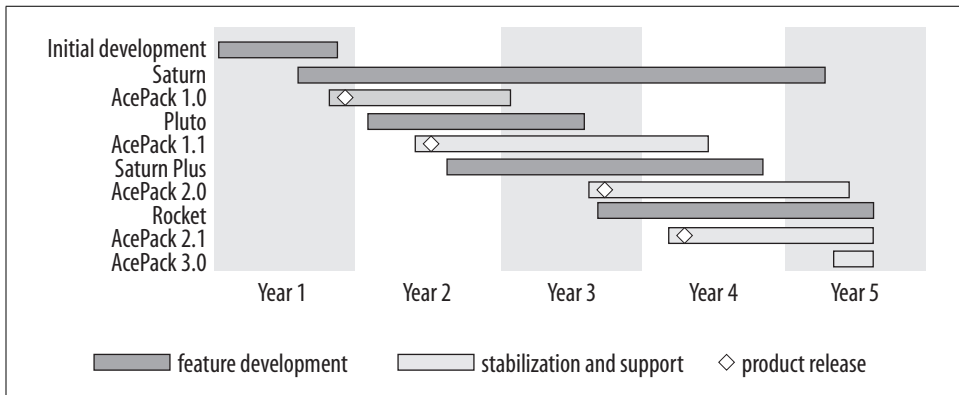


*Figure 7-1. Ace Engineering's first five years*

The Ace Engineering story illustrates typical problems of the software life cycle:

- First, at any point in time there is likely to be more than one supported product version available to customers. Ace must be prepared to field customer calls, diagnose problems, and fix critical bugs in all of their currently supported versions.

- Second, not all development tasks have the same urgency. Some are expected to yield results immediately while others are targeted for distantly future releases.

- Third, software development is not entirely predictable; some projects go according to plan, others get mired in unforeseen difficulties.

What Ace Engineering makes is shrink-wrapped* software. Other kinds of software—web-hosted, embedded, open source—evolve differently and have life cycle problems of their own. What all of them have in common is that their software life cycle problems can be solved by parallel development. Ace, for example, solved its

---

* Shrink-wrapped software is software that is distributed in periodic releases. The provider decides when to make new releases available; users decide when to upgrade. As a consequence, there can be several releases in use at the same time and the provider may have to support many or all of them concurrently.

problem of having to support customers on two releases by putting some developers to work on the old release while others developers worked in parallel on the new relese.

Software development is complicated enough; *parallel* software development can be even more complicated. But it doesn't have to be. In the next section, we'll look at how the mainline model can be used to keep the complexities of parallel development in check.

# The Mainline Model

A *codeline* is, for the most part, the same as a branch. But while the term *branch* can mean any set of files created by branching, *codeline* is imbued with slightly more significance. Codelines have a purpose, a strategic role in the development of software. Together, codelines form a model of software evolution.

In the Perforce view of software configuration management, one model, the *mainline model,* is most effective. This chapter discusses codelines and software evolution in the context of the mainline model. It's not a Perforce-specific discussion, by the way—the mainline model is a concept, not a Perforce feature. But it is the concept on which much of the design of Perforce is based.

## From Ideal World to Real World

In the ideal world, there are no bugs, no schedule crunches, no personnel changes, no market shifts, and no technology revolutions. Software in the ideal world is simply developed and released—that is, new features are developed, and when they're ready, a new version of the software is released. Each release contains features that work perfectly.

If there were such an ideal world, we probably wouldn't need an SCM system. Even so, we'd have a collection of files evolving together in a codeline. This codeline embodies the evolution of our software; it is our *mainline*. In the ideal world it would be the only codeline we'd ever need. (See Figure 7-2.)
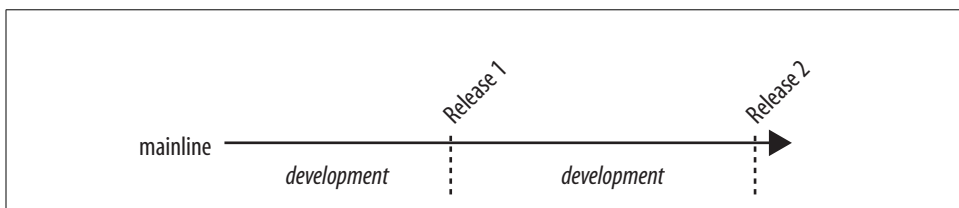


*Figure 7-2. The mainline in the ideal world*

A sad fact of the real world is that the software we develop isn't perfect. Because of that we subject software to a testing phase before release, during which bugs are

invariably found. We could do all this in the mainline if development could halt for the testing phase, and if all bugs could be found and fixed during testing. But all bugs are *not* found during testing; many are found after software is released. And we can't hold off on new development during the testing phase because we have deadlines and market pressures to face. So we branch completed software from the main codeline into a release codeline.

Branching release codelines allows us to do two different kinds of software development at once. One kind is bug fixing—euphemistically known as *stabilization*—and the other is new feature development. In the release codeline, we stabilize a version of our software—both before and after release—while in the mainline we get on with developing new features. As we stabilize the release version we can make point-releases—that is, we can re-release the software—without the risk of releasing untested new development.

Another problem with the real world is that our customers expect us to fix bugs in old versions of our software even as we are developing and stabilizing new versions. To deal with this, we branch a new codeline for each release, leaving our old release codelines intact for more bug-fixing. Now the typical shrink-wrapped software evolution model begins to take shape. A mainline charts the course of the overall development, while release codelines sprout as new versions are ready. When releases are no longer supported, the codelines designated for them cease to evolve, but the mainline persists. (Figure 7-3.)
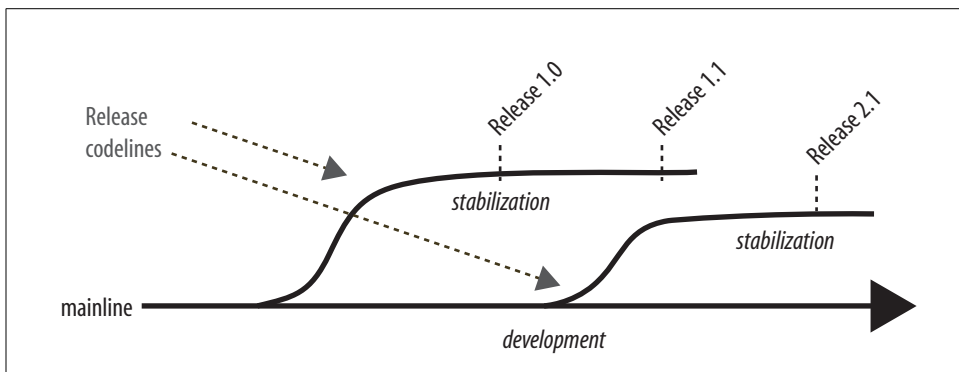


*Figure 7-3. Release codelines*

In the ideal world, all development projects are completed on schedule. No matter how many new features are slated for the next release, developers in the ideal world get them all done on time. But this is not the ideal world, and in the real world a single incomplete project can hold up an entire release if it's in the same codeline as completed projects. To decouple development projects from one another in the real world, we can branch the mainline into one or more development codelines. Development projects are delivered to the mainline as they're completed, and when

enough development is completed to warrant it, a new version is branched for release stabilization. (Figure 7-4.)
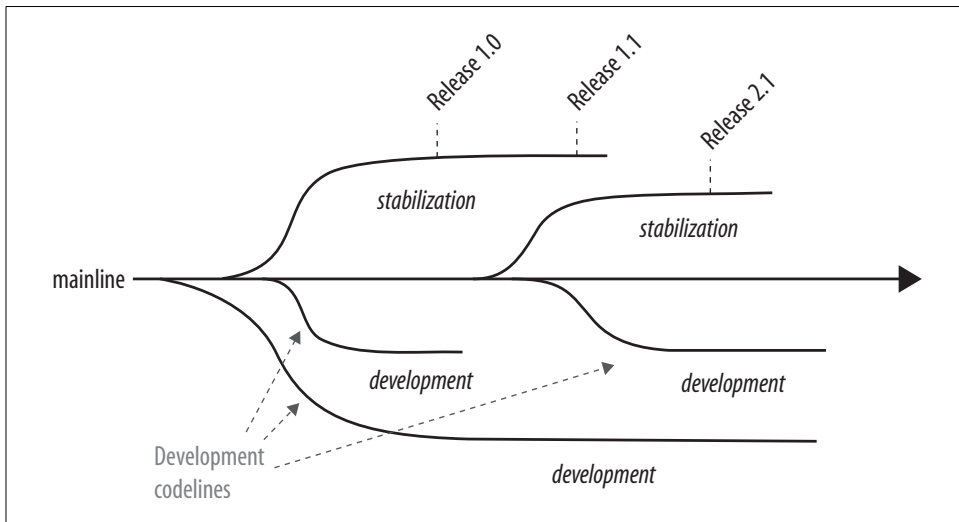


*Figure 7-4. Development codelines*

Thus our mainline evolves as new development is completed, although development does not necessarily take place in the mainline.

And there you have it. The mainline model is a necessary deviation from the ideal world. It seeks to preserve the intent of the ideal world while accommodating the constraints of the real world.

# Why We Don't Drive Through Hedges

Why not just branch a development codeline into a release codeline? Or merge a bug fix straight from a release codeline into a development codeline? Well, just as there is no law of physics that keeps you from driving through hedges to get on and off the freeway, there's nothing in Perforce that keeps you from integrating changes any which way you please.

One has only to look at traffic on a freeway to see why entrances and exits are controlled. Clearly driving would be inefficient and unpredictable if they weren't. Because we can't see the flow of change, it's not so easy for us to understand that we must control change for the same reason: Parallel development would be inefficient and unpredictable if we didn't.

Think of the mainline model as the freeway system of parallel development. It's a fast and reliable way to get somewhere, but only if we resist the temptation to drive through the hedges.

# The Flow of Change

The closer we are to the ideal world, the simpler our SCM is. When our ideal-world intent to work together is thwarted by real-world constraints, we branch one codeline into another and make our changes there. But we don't lose sight of the fact that our changes in the second codeline are really meant for the first. We pull changes from the second codeline into the first as soon as we can.

It is this flow of change between codelines that brings us closer to the ideal world. Each codeline type—mainline, release, and development—has a role in the flow of change:

*The Mainline*

> The mainline is the clearing-house for all changes to the software we develop. Whether we submit changes directly to the mainline or integrate changes to it from other codelines, all change eventually reaches the mainline. (Figure 7-5.)
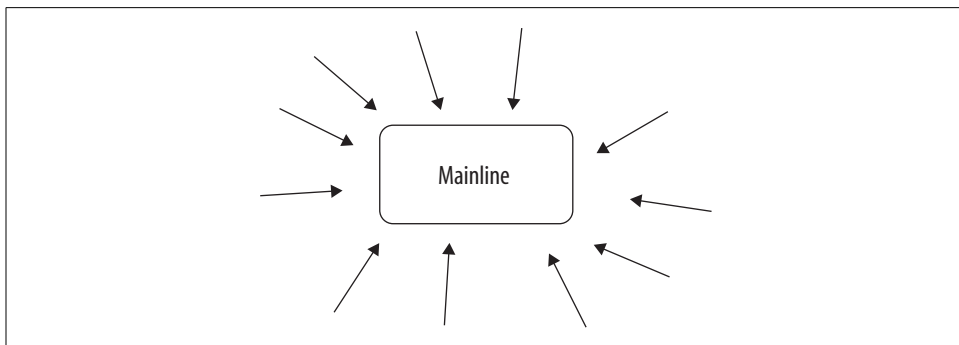


*Figure 7-5. All change flows to the mainline*

> However, the mainline isn't a free-for-all. It holds the software that's complete enough to enter the release stabilization cycle. So the flow of change to the mainline is tempered by the state of the codelines from which it flows.

*Release Codelines*

> Change flows continually from release codelines to the mainline. Every time a bug is fixed in a release codeline, the change that fixed it is integrated into the mainline, as Figure 7-6 shows. This doesn't compromise the mainline, because every change coming from a release codeline has already been reviewed and tested. Moreover, release codeline changes are changes that fix broken things. Thus, merging release codeline changes to the mainline is bound to have a stabilizing effect. It brings the mainline closer to perfection, as is our ideal-world intent. This flow of change continues until the mainline has evolved so much that the bug fixes in a release branch are no longer relevant to it. (In "How much has the mainline changed? we'll take a closer look at this.)
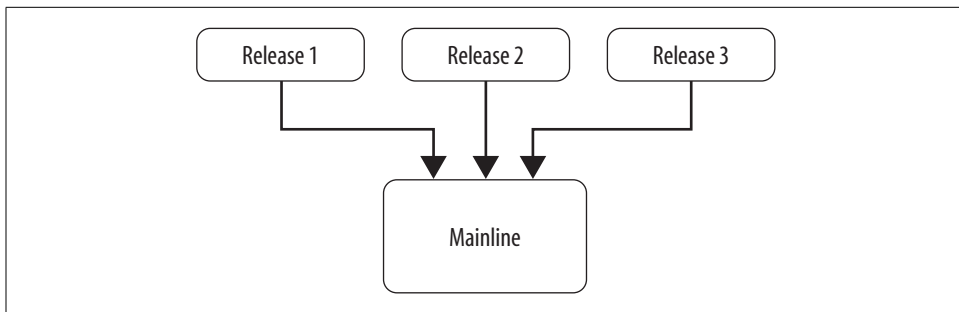
*Figure 7-6. Bug fixes flowing to the mainline*

Release codelines are not normally open to changes from the mainline. For one thing, every change to a release codeline should be a change that stabilizes and finalizes the release. For another, the mainline is changing constantly—it will never be perfect. We don't want the increasing perfection of a release codeline to be sullied by the inherent imperfection of the mainline.

(While release codelines are not normally open to mainline changes, the unexpected can happen. If we're in the unfortunate position of having to support several releases concurrently, a bug fix in one release may have to be applied to another. That is, we'll have to cherry-pick a change from either a release codeline or from the mainline and integrate it into another release codeline. In "Backporting a bug fix we'll cover this in more detail.)

*Development Codelines*

There's also a constant flow of change from the mainline to the development codelines branched from it, as shown in Figure 7-7. In other words, a development codeline is continually updated with changes from its parent codeline. Thus even development codelines benefit from release stabilization. As we fix a bug in a release, we merge the bug fix into the mainline. As the mainline changes, we merge its changes into development codelines.[*]

In some cases, bugs can be fixed right in the mainline. Because the mainline is guaranteed to be stable, development codelines can be continually updated with these bug fixes. This gives project teams the benefit of working with the latest, improved code. It also forces them to integrate sooner, rather than later, with development happening outside of their control.

What about the flow of change from development codelines to the mainline? A development codeline can be a hotbed of untested new development. There may be periods of time when a development codeline doesn't build, or when it builds nothing but a basket case. The valve is closed on the flow of change from the development codeline to the mainline during these periods.

---

[*] Who is this we, kemo sabe? See "The codeline curator" and "Who does the integration?."
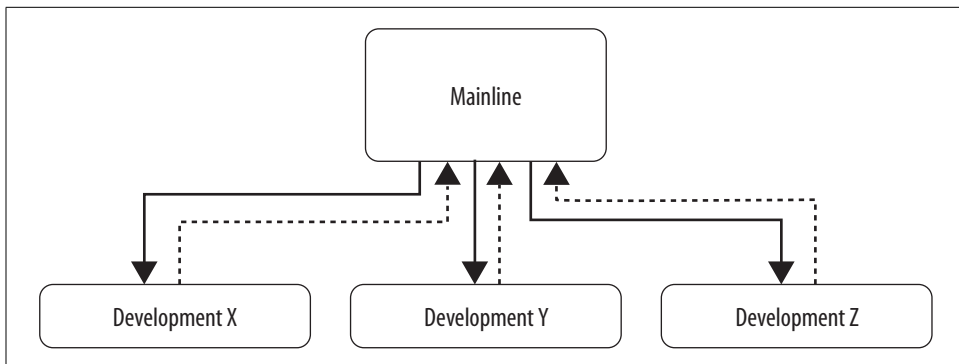
*Figure 7-7. Flow of change between the mainline and development codelines*

> But when the development codeline is stable—when new development is complete or at a deliverable state—the valve opens. At these points the development codeline software is delivered to the mainline. Thus change flows from development codelines to the mainline at points of completion. And, because development codelines are always open to mainline changes, other development codelines will receive the completed new development as well.

Development and release codelines can themselves be branched. Quite often they're branched into short-lived, task-specific sub-branches to accommodate unplanned changes. A release codeline can be branched to make a patched version of a released product, for example, and a development codeline can be branched to isolate work on a specific problem or behavior.

## Branching from Release Codelines

In the ideal world, our customers upgrade to our latest release without complaint. In the real world, customers have reasons they can't do that, and we have reasons to keep our customers happy. This occasionally puts in the position of having to patch a previously release version. We do this by branching a release codeline into a patch branch.

The flow of change between a patch branch and its release codeline parent is exactly the same as the flow of change between a release codeline and its mainline parent. In other words, the release codeline is continually updated with changes from the patch branch. (Not that there's likely to be much change in the patch branch.) This gives the release codeline the benefit of the patch branch's bug fix. No change flows from the release codeline to the patch branch because the whole point of making the branch was to reproduce and patch an earlier version.

For example, one extremely important customer, still using Release 1.0, finds a critical bug and demands a patch that doesn't require an upgrade to our latest point release, 1.1. To accommodate this customer, we take the 1.0 version of the Release 1

codeline and branch it into a patch branch. We fix the fussy customer's bug in the patch branch and build a new version from a snapshot of the branch. This is the version we give to the customer. (Figure 7-8.)
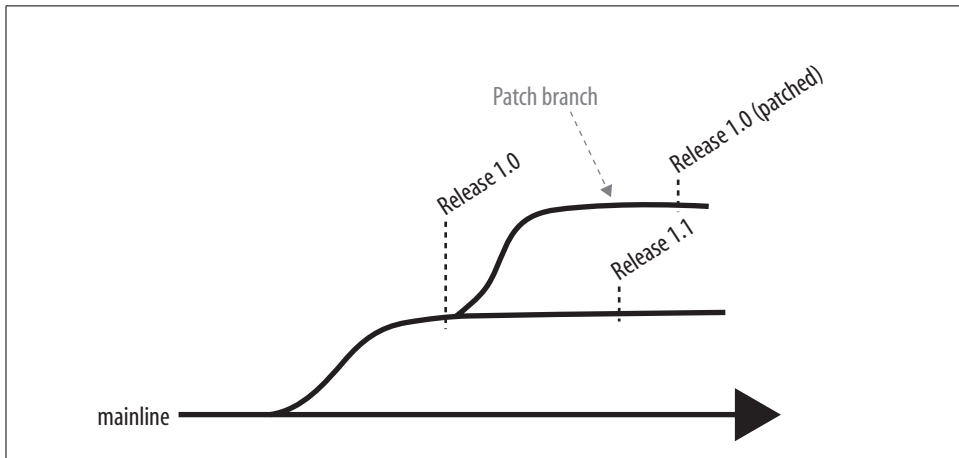


*Figure 7-8. A patch branch*

The changes we made in patch branch are merged into the Release 1 codeline. This gives the Release 1 codeline the benefit of the patch. Release 1 changes flow into mainline of course, bringing the patch with them. (Figure 7-9.)
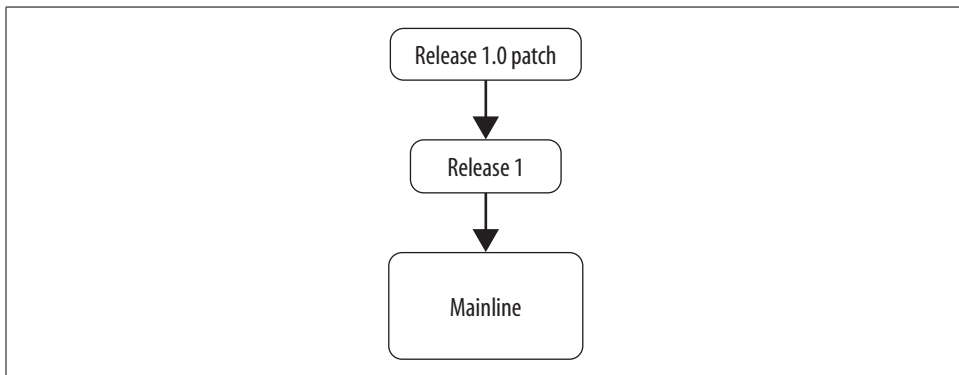


*Figure 7-9. Patch branches and the flow of change*

## Branching from Development Codelines

Development codelines branched into sub-branches also inherit the flow-of-change roles of their parents: Change flows continually from development codelines to their sub-branches so that work in a sub-branch is always up to date with the parent codeline. Change flows in the other direction only at points of completion. When work in

a sub-branch is completed, it's delivered to the development codeline. Thus, each development codeline acts as a mainline for its sub-branches, and each sub-branch behaves like a development codeline.

Consider a team of developers working on an application. They're using a codeline they've named DEVX to develop a major new feature. Two developers plan to help out by overhauling a part of the new feature known as the the Z-widget. It's going to be a ground-up rewrite; the Z-widget won't be working right again for weeks. But a broken Z-widget will make it impossible for other developers to work in the DEVX codeline. (They could simply relax and play ping-pong for two weeks but that doesn't go over very well in the real world.)

To satisfy constraints of the real world, the DEVX codeline is branched into a codeline named DEVZ. (Figure 7-10.) The two Z-widget developers complete their overhaul in the DEVZ codeline. The rest of the developers continue their work in its parent, DEVX, with an old but stable Z-widget. (And they promise not to touch any of the Z-widget files in DEVX.)
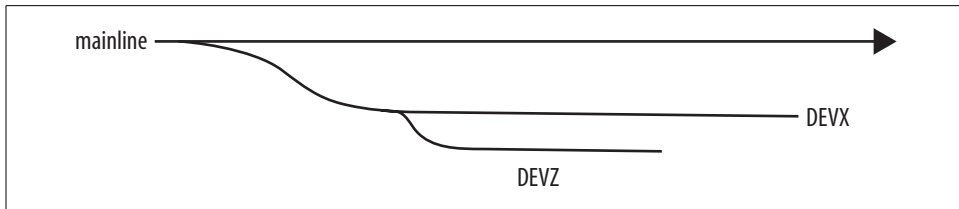


*Figure 7-10. A development sub-branch*

As changes are made in DEVX, the Z-widget developers pull them immediately into DEVZ. They don't put their changes back into DEVX, however, until their overhaul is done and the new Z-widget is stable.

To the Z-widget developers, this satisfies the ideal-world intent to build upon other developers' changes, as if they were working right in DEVX. To the developers working in DEVX, this satisfies the ideal-world expectation that they won't lose project time waiting for broken components to work again. (Figure 7-11.)

## Soft, Medium, and Firm: the Tofu Scale

Every codeline has a characteristic ranking on the tofu scale. And what is this tofu scale? It's an informal assessment of stability and quality that takes into account:

- How close software is to being released
- How rigorously changes must be reviewed and tested
- How much impact a change has on schedules
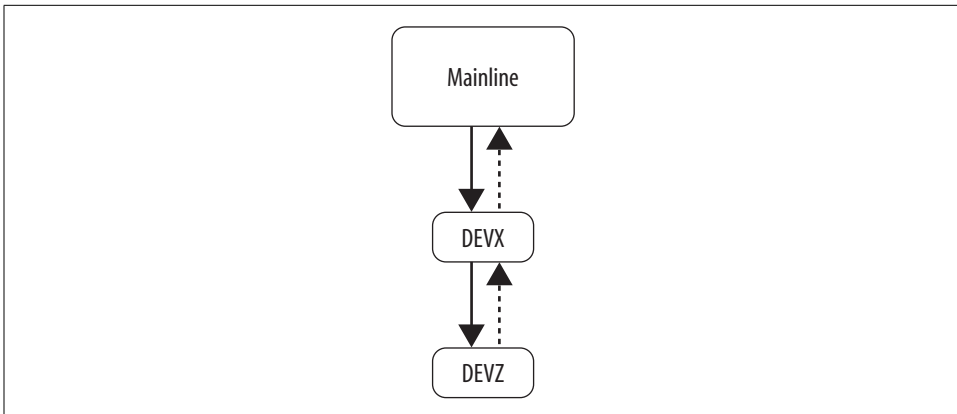- How much a codeline is changing

*Figure 7-11. The flow of change to and from development codelines*

Release codelines are highest on the tofu scale; they are firm. They don't change much, and even the slightest changes to them can impact release schedules because of their rigorous review and testing requirements. The mainline is medium—changes do require testing, but release is further out and schedules are more accommodating of them. Development codelines are soft—they're changing rapidly, the software in them is farthest from release, and there may not even be tests yet for their newest development. (Figure 7-12.)
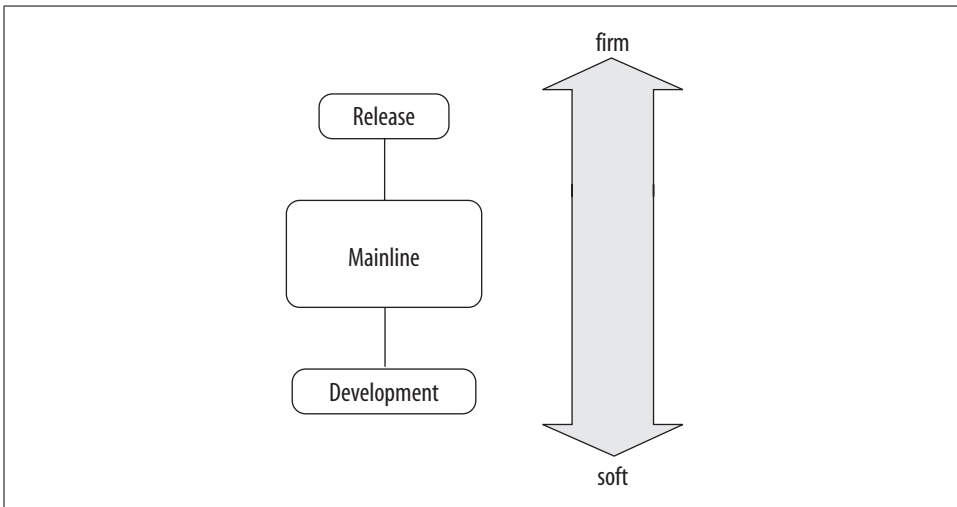


*Figure 7-12. The tofo scale*

The flow-of-change rules tell us *when* file content should be propagated from one codeline to another. The tofu scale tells us *how*:

- In the firm-to-soft direction, file content can be merged. The target, being softer, is more able to absorb the risk and effort of merging than is the donor.
- In the soft-to-firm direction, file content should be copied. The target is more at risk than the donor in this case. Files should be merged from the firmer codeline to the softer one first, then copied from the softer codeline to the firmer one.

The unwritten contract of collaborative development says that we don't impose unstable changes and we always accept stable changes. The tofu scale gives us a way to tell unstable from stable changes before we impose or accept them.

There's a uniformity to the mainline model that can be described in terms of flow of change and the tofu scale. Aside from the mainline, which is in a category of its own, there are essentially only two codeline types, release codelines and development codelines. No matter how many codelines you have, if you know a codeline's type, you know exactly how and when file content should be propagated between them. This is summarized in Table 7-1; you'll see how this plays out in the chapters that follow.

*Table 7-1. How change is propagated*

|  | Release codeline | Development codeline |
|---|---|---|
| Tofu rank: | Firmer than parent | Softer than parent |
| Change flows *to* parent: | Continually | At points of completion |
| Change flows *from* parent: | Never | Continually |
| File content is propagated: | By merging | By copying |

## A Codeline by Any Other Name...

...is still a codeline. If you're saying to yourself that a mainline and a handful of development and release codelines are not going to satisfy your SCM needs, you're right. In fast-paced, large-scale development environments, the fundamental codeline types are adapted and extended to a variety of uses:

*Active development streams*

Sometimes development projects aren't all that clear cut. One use of development codelines is to support long-lived, ongoing development work on components. This gives component developers a common, persistent codeline to work in without requiring them to create a new codeline for each task or feature. In Chapter 10, for example, you'll see how a development codeline is used as an active development stream for a GUI component.

*Task branches*

Task branches are very short-lived codelines branched from either development codelines or release codelines. They can be used to protect release codelines from untested interim changes, or to protect development codelines from destabilizing re-engineering. (The DEVX development codeline described in "Branch-

ing from development codelines" is a task branch.) In "Task Branches and Patch Branches" you'll see how a task branch is used to permit a bug fix to be reviewed and tested before it's introduced into a release codeline.

*Staging streams*
Staging streams allow you do make extremely frequent releases without having to branch a new codeline for each release. (They're commonly used to support web development. You'll see an example of this in Chapter 11.) A staging stream is essentially a reusable release codeline. Each staging stream is used for a particular stage of release stabilization. Once the stage is completed for a particular release, the codeline is immediately redeployed for the next release.

*Private branches, ad hoc branches, and sparse branches*
Private branches make it possible for each developer's changes to be reviewed before they are submitted to shared codelines. Private branches can also be used to isolate experimental or proof-of-concept work. Ad hoc branches are created on the fly to give users a place to check in changes they thought they were going to be able to check in elsewhere but found out they couldn't. Sparse branches can be used in any of the aforementioned cases to piggy-back a few changed files onto a full codeline. Examples of all of these will come up in later chapters.

# One-Way Codelines

We also recognize another fundamental codeline type, the one-way codeline. One-way codelines house software, but not software development. The following are examples of one-way codelines:

*Third-party codelines*
Third-party codelines provide a place to store vendor drops—software and source code obtained from external suppliers. Code is typically copied or merged from third-party codelines into development codelines.

*Remote depot codelines*
In Chapter 6 you read about how you can access depots in other Perforce domains as *remote depots*. Codelines in remote depots are always one-way codelines—files can be branched, copied or merged from them, but not to them.

*Packaging and distribution streams*
Packaging streams can be used to assemble customer-specific configurations of software from released components. Distribution streams can be used to offer released products to customers. (They're the vendor's side of the vendor drop.) Software is delivered from release codelines to packaging and distribution streams; nothing is ever copied or merged in the opposite direction.

We'll revisit one-way codelines in "Distributing Releases" and "Working with Third-Party Software."

## Codelines that Aren't Codelines

Finally, while we're cataloguing codeline species, it's worth recognizing that some codelines aren't really codelines at all:

*Porting branches*
> Porting branches contain architecture-specific variants of source code and built objects.

*Custom-code branches*
> Custom-code branches contain source and built objects configured for hardware, customers, locales, and other deployment targets.

Branches like these aren't codelines in their own right. Although it's often difficult to recognize the fact, they're really modules that belong in development codelines, release codelines, and the mainline.[*]

## Musings on Codeline Diagrams

Codeline and flow diagrams help us visualize software evolution. We have all, at one time or another, sat in a room with our colleagues and drawn or pondered diagrams on a whiteboard. Here are some things to keep in mind when you find yourself doing the diagramming:

- When you're drawing a timeline, put release codelines above their parents and development codelines below them. This orders codelines on the tofu scale, with the firmest on the top and the softest on the bottom. (Figure 7-13.)

  The tofu scale shows the impact of a change at a glance. A change made to a codeline at the top of the diagram, for example, will reach customers soonest, at the greatest risk to quality and scheduling. A change made to a codeline at the bottom of the diagram, on the other hand, doesn't pose a great risk, but it will be a while before it is available to customers.

- Remember the time axis when you're drawing timelines. Plot codeline beginnings and endings in time order. That way, any vertical line you draw will tell you how many active codelines you'll have at that point in time. (Figure 7-14.)

- If the timelines in a diagram are getting congested, try using slightly parabolic lines instead of horizontal ones. Parabolic lines also imply divergence—the greater the vertical distance to the mainline, the more the codeline is likely to have diverged from the mainline. (Figure 7-15.)

- Remember that *historical* flow of change isn't the same as *intended* flow of change. Use flow diagrams in addition to timelines to help people understand

---

[*] For a real-world example of how custom-code branches can become errant codelines, see *Changing How You Change* (*http://www.ravenbrook.com/doc/2003/03/06/changing-how-you-change/*), a white paper presented by Peter Jackson and Richard Brooksby at the 2003 Perforce User Conference.
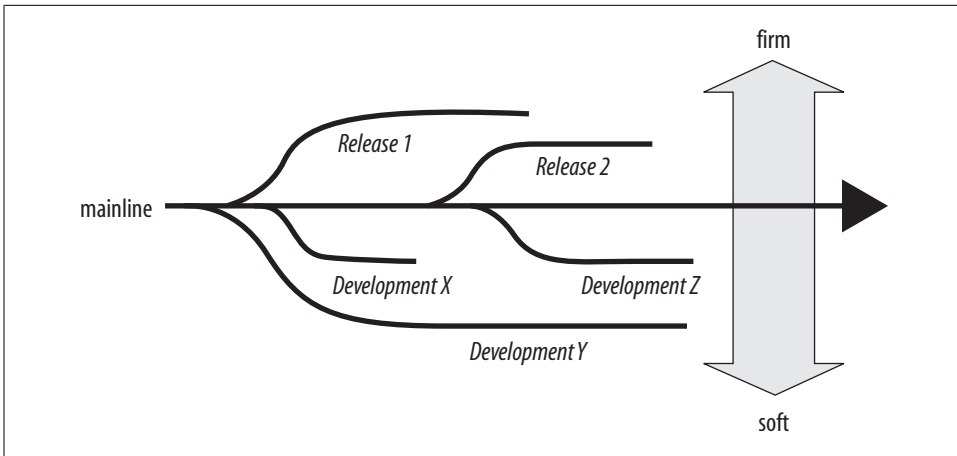
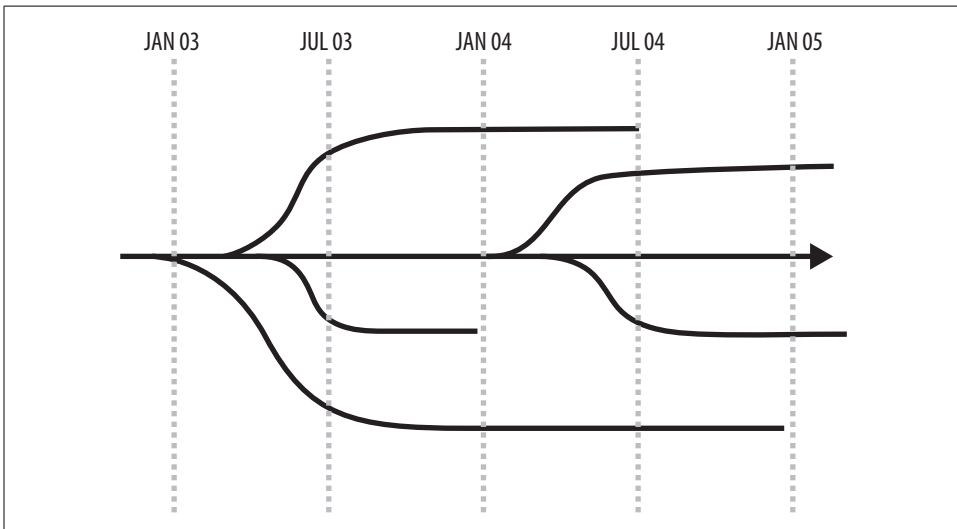*Figure 7-13. Ordering codelines on the tofu scale*



*Figure 7-14. Ordering codeline beginnings and endings on the time axis*

how change is intended to flow between codelines. Show the tofu scale in flow diagrams as well. (Figure 7-16.)

- Finally, recognize that if your codeline diagram is complicated, your SCM process is going to be complicated. Simplifying the diagram may be a good first step in reducing the complexity of your SCM plan.
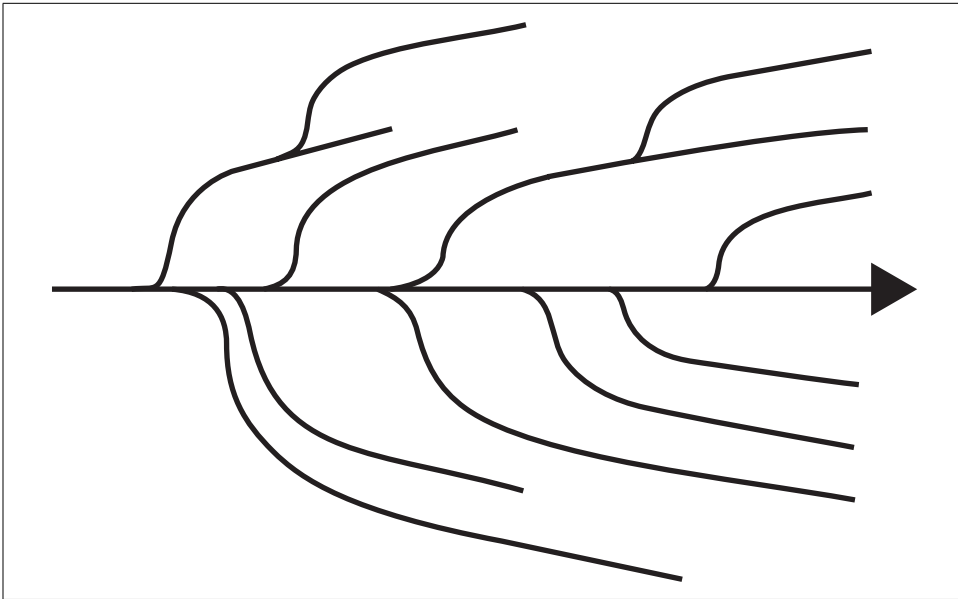
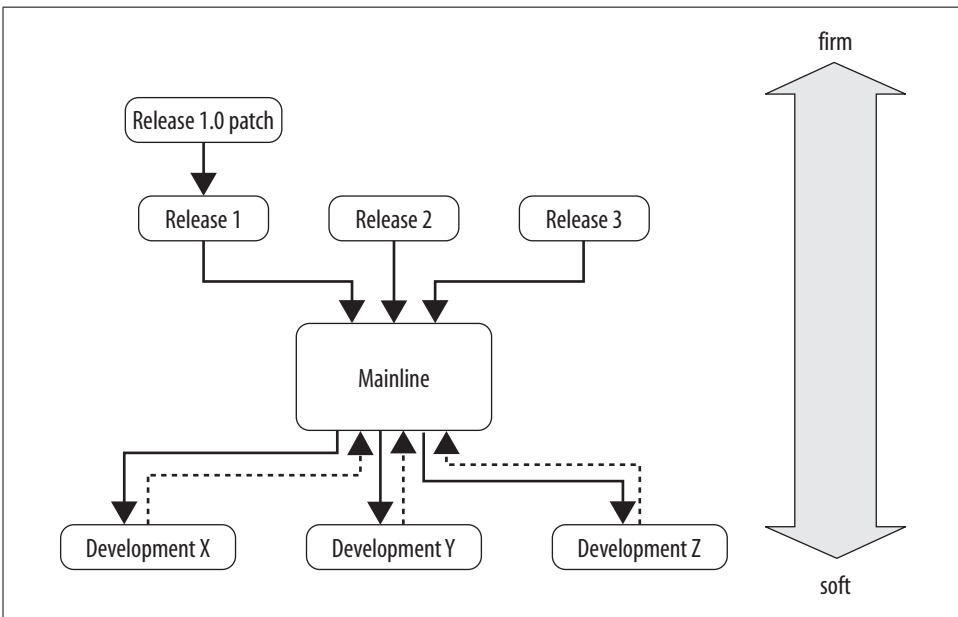*Figure 7-15. Using parabolic lines to reduce congestion in a diagram*



*Figure 7-16. A flow diagram*

# Ace Engineering Revisited

Let's look at the mainline model as employed by Ace Engineering. Once an initial body of development was begun, it formed the MAIN codeline. As you read, the first AcePack versions released were 1.0 and 1.1. However, Ace engineers did not plan their schedule around two first-generation (1.X) releases. In fact, they didn't know how many 1.X releases they would make. Their strategy was simply to make periodic point releases to fix bugs and tidy up loose ends until the next major release was ready. To support the as-yet undetermined number of 1.X releases, MAIN was branched into REL1. (Figure 7-17.)
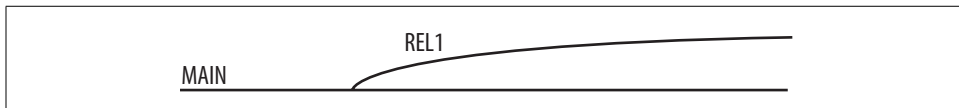
*Figure 7-17. REL1 branched from MAIN*

AcePack 1.0 and AcePack 1.1 were released from the REL1 codeline. REL1 was used for ongoing fixing before and after the releases were made. When AcePack 1.0 customers required patches, the AcePack 1.0 version of the REL1 codeline was branched into R1.0. The only changes allowed in R1.0 were fixes for critical, show-stopping bugs; these were immediately merged into REL1. Thus, when AcePack 1.1 was released, all the R1.0 fixes were already in it. The same strategy was used to patch AcePack 1.1—the REL1 codeline was branched into R1.1. No changes except for critical fixes were allowed in R1.1, and all R1.1 changes were merged immediately into REL1. Meanwhile, changes in REL1 were regularly merged to MAIN; thus the mainline always reflected the sum of improvements made in the 1.X releases. (Figure 7-18.)
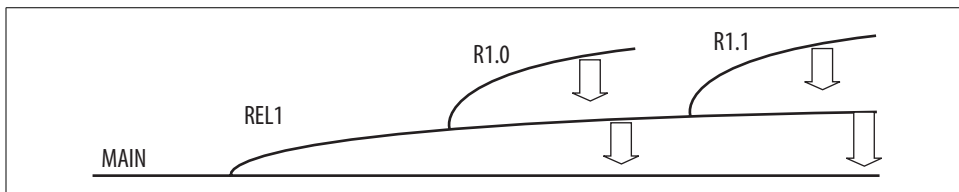
*Figure 7-18. The 1.X release branches*

R1.0 and R1.1 were fairly inactive codelines. Their entire purpose was to provide a place to fix show-stopper bugs in released versions. As a result, neither R1.0 nor R1.1 deviated very much from their parent, REL1, and merging changes from them into REL1 was extremely easy.

You may be wondering why Ace used separate R1.0 and R1.1 codelines to patch released products instead simply building patched releases from REL1. The reason is that while the releases were concurrently supported, developers had to be able to

deliver critical bug fixes to both releases without requiring customers on either release to upgrade. (Ace Engineering is a very accommodating company.) And if you're wondering what REL1 was used for after R1.1 was released, recall that Ace engineers were not sure how many 1.X releases they would make. If all went well, the new features for the next major release would be ready soon and the REL1 codeline could be abandoned. But on the chance that the new-feature schedule would slip, developers continued fixing bugs in REL1 and merging their fixes into MAIN. That way they would have yet another 1.X point release ready to go if no 2.0 release were forthcoming.

To develop the new features slated for the 2.X release, the SATURN and PLUTO codelines were branched from MAIN. Giving each feature development project its own codeline ensured that the two could remain independent. Changes in MAIN were routinely merged into both SATURN and PLUTO, keeping the development codelines up to date with the latest bug fixes from the release codelines. (Figure 7-19.)
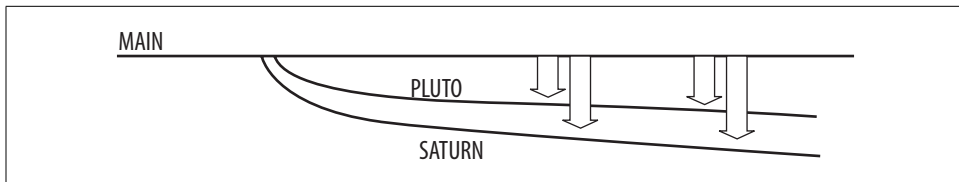


*Figure 7-19. Branching for two development projects*

As you recall, the Saturn feature turned out to be bigger project than anticipated. The SATURN development codeline was eventually branched into SATURNPLUS to keep the now two Saturn development teams out of one another's hair. Feature development continued in SATURN while some much-needed infrastructure work was done in SATURNPLUS. The SATURN changes were continually merged to SATURNPLUS. This made it very easy for SATURNPLUS work to be delivered to SATURN when the infrastructure work was completed. At that point, SATURNPLUS was retired. (Figure 7-20.)
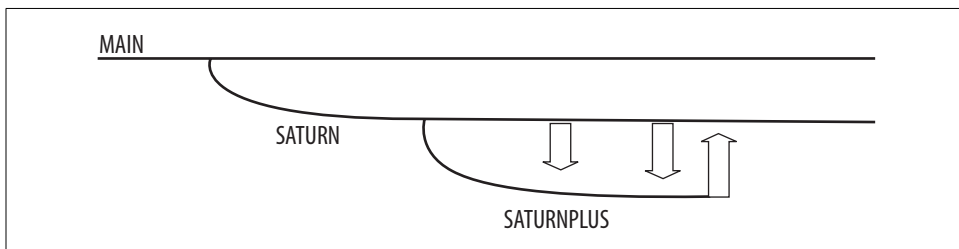


*Figure 7-20. Isolating SATURNPLUS development*

The Pluto team was the first to complete a new feature for the next major release. The PLUTO codeline's work was delivered to MAIN and PLUTO was retired. At that point, the decision was made to defer the Saturn feature to a future release, and MAIN was branched to REL2 for the second major AcePack release. At about this time, development on another new feature was begun in a codeline called ROCKET that was branched from MAIN. All the while development continued in SATURN; MAIN changes were continually merged into SATURN to keep the latter from diverging. (Figure 7-21.)
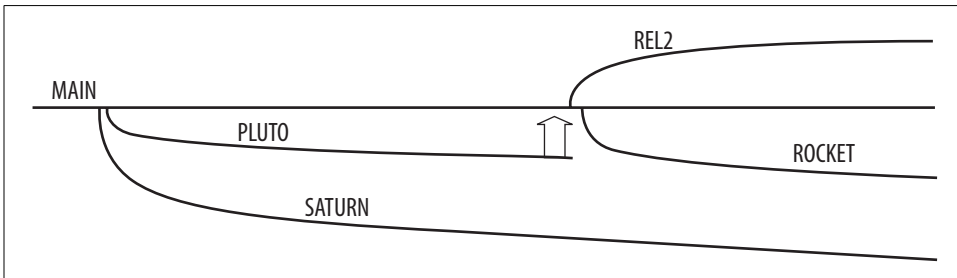


*Figure 7-21. Branching REL2 and ROCKET from MAIN*

Meanwhile, Ace produced two releases, AcePack 2.0 and AcePack 2.1, out of the REL2 codeline. In order that each release could be patched independently, REL2 was branched into R2.0 and R2.1, respectively. Critical, show-stopping bugs were fixed in the R2.0 and R2.1 patch branches, while REL2 formed a trunk for ongoing bug fixing and stabilization. Bug fixes in the patch branches were merged back into REL2, and REL2 changes were merged into MAIN (Figure 7-22).
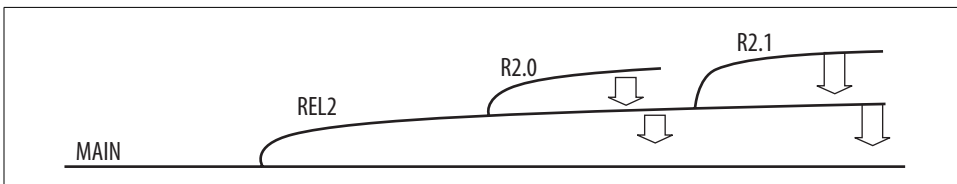


*Figure 7-22. Branches for 2.X point releases*

Once the Saturn development project was completed, work in the SATURN codeline was delivered to the mainline, and the mainline was in turn branched into REL3 in order to stabilize and build what will be AcePack 3.0. (Figure 7-23.)

Thus Ace Engineering has produced and supported five releases using a total of 12 codelines. That seems rather a lot of codelines until you consider that only a handful were ever active at the same point in time. Moreover, because the mainline embodies all completed work so far, it serves as the single point of reference for future development. SCM is really no more complicated at Ace Engineering now than it was when the first release was made.
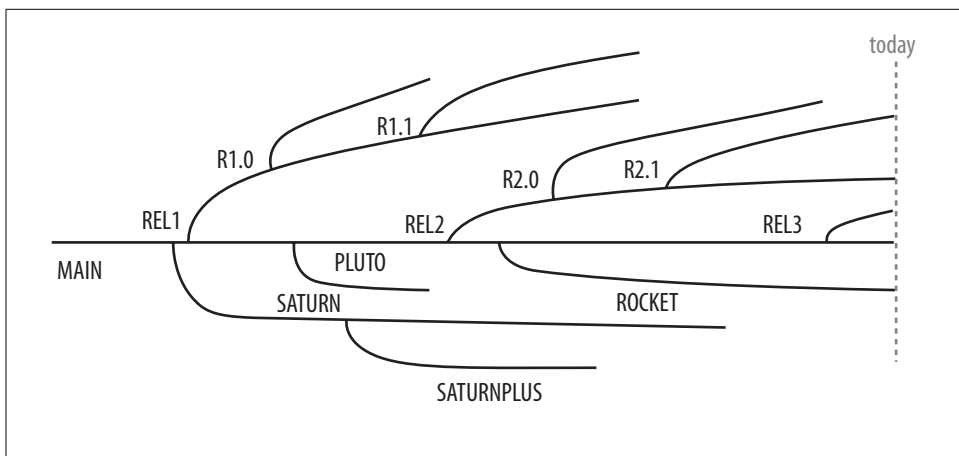
*Figure 7-23. Ace's codelines today*

# Containerizing

In commerce, manufacturers control complexity by containerizing. A truck driver doesn't know he's delivering 19 sofas, 400 chairs, and 80 tables to outfit a hotel in New York. All he knows is that he's delivering one shipping container from the factory to the shipyard. And when he arrives at the shipyard, he doesn't throw open the back of the truck and start counting out sofas. He merely checks that the container is intact.

It's the same with branching and merging. The software we're working on involves far too many files for us to have to branch and merge them individually. We need to containerize so we can branch and merge mere handfuls of containers instead of hundreds of thousands of files.

Although they seem to have different names everywhere they're used, the file containers essential to software development are *modules*, *codelines*, and—for lack of a better term—*bodies of code*.

## Modules

The files we work on are grouped into *modules*. At face value, a module is simply a set files organized in a directory tree. (Other systems, and other writings on the topic, use terms like *source directory*, *component*, and *subsystem* for what we're calling a *module*.) What makes modules important is that they correspond to the file hierarchies needed on disk in order to work on specific parts of the software being developed.

In the development of the AcePack software, for example, the GUI module corresponds to the directory tree of C++ source files and Jamfiles[*] needed to build Ace-

Pack's GUI components. In order to work on the GUIs a developer needs this directory tree on disk. Other modules support other areas of development. The database module, for example, contains the scripts and stored procedures needed on disk to work on AcePack's database component. The documentation module contains Frame files and generated HTML files for the AcePack manuals. The utilities module contains AcePack-specific scripts and configuration files common to all development tasks. (Figure 7-24.)
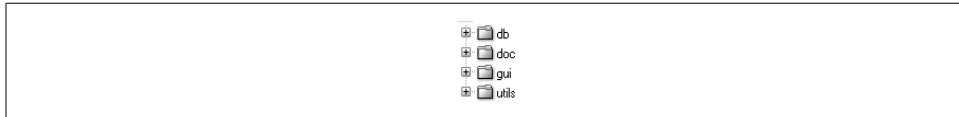


*Figure 7-24. Top-level modules*

Modules can contain other modules as well. For example, the AcePack GUI module is subdivided into a module of common code and a module for each of the AcePack GUI tools. The documentation module is subdivided into modules for each AcePack manual plus a module containing the tools used to build them. (Figure 7-25.)
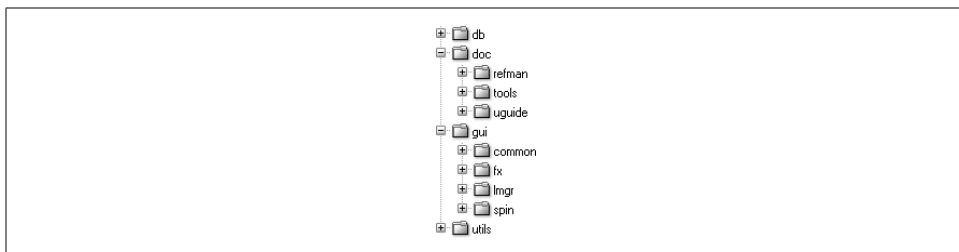


*Figure 7-25. Sub-modules*

Modules are the raw materials of workspaces and builds. When a developer sets up a workspace, she sets it up in order to work with certain modules. When build engineer builds software, his build tools process files in some modules and create files in others.

And although her workspace may be populated with many top-level modules, the changes a developer makes typically affect only one module at a time. A GUI programmer makes changes in the GUI module without affecting the database module or the utilites module, for example. Thus any change submitted by a developer is likely to affect no more than one module.

---

\* Jamfiles are used by Jam, an open-source build tool. Although Jam is available from Perforce Software, it is completely separate from Perforce.

## Codelines

A codeline contains modules evolving together in the same phase of development. (Another good word for codeline is *stream*; it emphasizes the point that a not only is a codeline a container, it is a vessel that channels its contents toward completion.) Modules evolve together in a codeline because they contribute jointly to a specific version of an end product or suite of products. The 2.1 version of AcePack, for example, is made up of the 2.1 version of the GUI tools, the 2.1 version of the database, and the 2.1 version of the documentation.

Branching a codeline is really a matter of branching the modules contained by the codeline. Not all modules need be branched, as we'll see in upcoming chapters. Depending on how closely software components are coupled, release codelines and development codelines may contain only the bare minumum of modules needed to support the work at hand.

Codelines also define the scope of build and test tools. A build script, for example, will only be able to see the files in a single codeline. Thus even if a module won't be changed in the course of a codeline's evolution, its presence in a codeline may be needed in order for build tools to work.

## Bodies of Code

A *body of code* is the complete collection of codelines related to one another. In other words, a mainline and all the codelines related to it by branching form a body of code.

At Ace Engineering, for example, there is only one body of code. It encompasses the codelines that support development and release of the AcePack product suite. An Ace developer, whether working in the MAIN codeline, in the R1.0 release codeline, or in the SATURN development codeline, is always working with the AcePack body of code.

At large companies it's likely that several bodies of code will coexist. Each body of code has its own mainline, its own development goals, its own release schedule, and its own collection of codelines. Products and packages built from one body of code can be imported into another, but each body of code is essentially independent.