CHAPTER 7

Releasing

What is the value of code? Agile developers value "working software over comprehensive documentation."* Does that mean a requirements document has no value? Does it mean unfinished code has no value?

Like a rock at the top of a hill, code has *potential*—potential energy for the rock and potential value for the code. It takes a push to realize that potential. The rock has to be pushed onto a slope in order to gain kinetic energy; the software has to be pushed into production in order to gain value.

It's easy to tell how much you need to push a rock. Big rock? Big push. Little rock? Little push. Software isn't so simple—it often looks ready to ship long before it's actually done. It's my experience that teams underestimate how hard it will be to push their software into production.

To make things more difficult, software's potential value changes. If nothing ever pushes that rock, it will sit on top of its hill forever; its potential energy won't change. Software, alas, sits on a hill that undulates. You can usually tell when your hill is becoming a valley, but if you need weeks or months to get your software rolling, it might be sitting in a ditch by the time you're ready to push.

In order to meet commitments and take advantage of opportunities, you must be able to push your software into production within minutes. This chapter contains 6 practices that give you leverage to turn your big release push into a 10-minute tap:

- *"done done"* ensures that completed work is ready to release.
- *No bugs* allows you to release your software without a separate testing phase.
- *Version control* allows team members to work together without stepping on each other's toes.
- A *ten-minute build* builds a tested release package in under 10 minutes.
- *Continuous integration* prevents a long, risky integration phase.
- *Collective code ownership* allows the team to solve problems no matter where they may lie.
- Post-hoc *documentation* decreases the cost of documentation and increases its accuracy.

---

* The Agile Manifesto, *http://www.agilemanifesto.org/*.

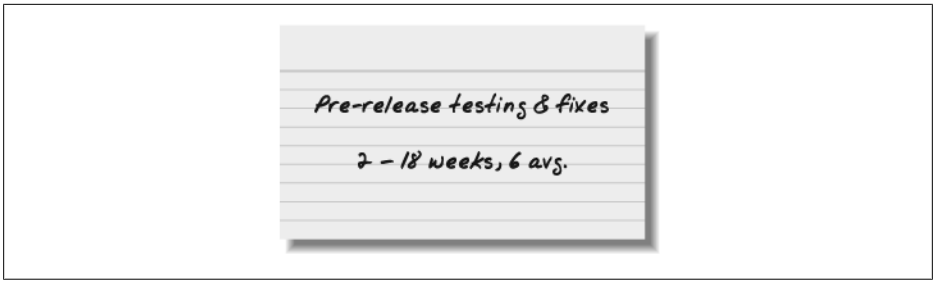

Figure 7-1. A sample card

"RELEASING" MINI-ÉTUDE

The purpose of this étude is to examine pushing software into production. If you're new to agile development, you may use it to create a map of all the steps involved in releasing software, even if you're not currently using XP. If you're an experienced agile practitioner, review Chapter 13 and use this étude to help you modify your process to remove communication bottlenecks.

Conduct this étude for a timeboxed half-hour every day for as long as it is useful. Expect to feel rushed by the deadline at first. If the étude becomes stale, discuss how you can change it to make it interesting again.

You will need red and green index cards, an empty table or magnetic whiteboard for your *value stream map*,* and writing implements for everyone.

Step 1. Start by forming heterogeneous pairs—have a programmer work with a customer, a customer work with a tester, and so forth, rather than pairing by job description. Work with a new partner every day.

Step 2. *(Timebox this step to 10 minutes.)* Within pairs, consider all the activities that have to happen between the time someone has an idea and when you can release it to real users or customers. Count an iteration as one activity, and group together any activities that take less than a day. Consider time spent waiting as an activity, too. If you can't think of anything new, pick an existing card and skip to Step 3.

Choose at least one task, and write it on a *red* card. Reflect on all the times you have performed this activity. If you have released software, use your experience; do not speculate. How long did it take? Think in terms of calendar time, not effort. Write three times down on the card: the *shortest* time you can remember, the *longest* time you can remember, and the *typical* time required. (See Figure 7-1.)

Step 3. *(Timebox this step to 10 minutes.)* Discuss things that your team can do to *reduce* the time required for this activity or to *eliminate* it entirely. Choose just one idea and write it on a *green* card.

Step 4. *(Timebox this step to 15 minutes.)* As a team, discuss your cards and place them on the table or whiteboard in a value stream map. Place activities (red cards) that must happen first before activities that can happen afterward. (See Figure 7-2.) If you're using a whiteboard, draw arrows between the cards to make the flow of work more clear. Place green cards underneath red cards.

Consider these discussion questions:

* The value stream map was inspired by [Poppendieck & Poppendieck].

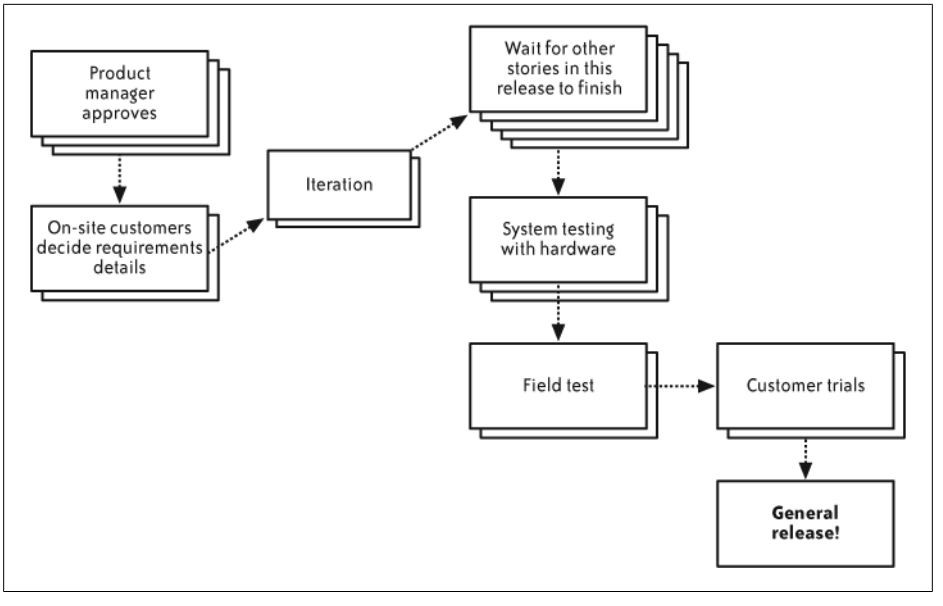

Figure 7-2. A sample value stream map

- At which step does work pile up?
- Which results surprise you?
- Who is the constraint in the overall system? How can you improve the performance of the overall system?
- Are there green cards with ideas you can adopt now?

"Done Done"

Audience
Whole Team

We're done when we're production-ready.

"Hey, Liz!" Rebecca sticks her head into Liz's office. "Did you finish that new feature yet?"

Liz nods. "Hold on a sec," she says, without pausing in her typing. A flurry of keystrokes crescendos and then ends with a flourish. "Done!" She swivels around to look at Rebecca. "It only took me half a day, too."

"Wow, that's impressive," says Rebecca. "We figured it would take at least a day, probably two. Can I look at it now?"

"Well, not quite," says Liz. "I haven't integrated the new code yet."

"OK," Rebecca says. "But once you do that, I can look at it, right? I'm eager to show it to our new clients. They picked us precisely because of this feature. I'm going to install the new build on their test bed so they can play with it."

Liz frowns. "Well, I wouldn't show it to anybody. I haven't tested it yet. And you can't install it anywhere—I haven't updated the installer or the database schema generator."

"I don't understand," Rebecca grumbles. "I thought you said you were done!"

"I am," insists Liz. "I finished coding just as you walked in. Here, I'll show you."

"No, no, I don't need to see the code," Rebecca says. "I need to be able to show this to our customers. I need it to be finished. *Really* finished."

"Well, why didn't you say so?" says Liz. "This feature is done—it's all coded up. It's just not *done* done. Give me a few more days."

Production-Ready Software

Wouldn't it be nice if, once you finished a story, you never had to come back to it? That's the idea behind *"done done."* A completed story isn't a lump of unintegrated, untested code. It's ready to deploy.

> You should able to deploy the software at the end of any iteration.

Partially finished stories result in hidden costs to your project. When it's time to release, you have to complete an unpredictable amount of work. This destabilizes your release planning efforts and prevents you from meeting your commitments.

To avoid this problem, make sure all of your planned stories are "done done" at the end of each iteration. You should be able to deploy the software at the end of any iteration, although normally you'll wait until more features have been developed.

What does it take for software to be "done done"? That depends on your organization. I often explain that a story is only complete when the customers can use it as they intended. Create a checklist that shows the story completion criteria. I write mine on the iteration planning board:

- Tested (all unit, integration, and customer tests finished)
- Coded (all code written)
- Designed (code refactored to the team's satisfaction)
- Integrated (the story works from end to end—typically, UI to database—and fits into the rest of the software)
- Builds (the build script includes any new modules)
- Installs (the build script includes the story in the automated installer)
- Migrates (the build script updates database schema if necessary; the installer migrates data when appropriate)
- Reviewed (customers have reviewed the story and confirmed that it meets their expectations)
- Fixed (all known bugs have been fixed or scheduled as their own stories)
- Accepted (customers agree that the story is finished)

Some teams add "Documented" to this list, meaning that the story has documentation and help text. This is most appropriate when you have a technical writer as part of your team.

Ally

The Whole Team (p. 28)

Other teams include "Performance" and "Scalability" in their "done done" list, but these can lead to premature optimization. I prefer to schedule performance, scalability, and similar issues with dedicated stories (see "Performance Optimization"" in Chapter 9).

How to Be "Done Done"

XP works best when you make a little progress on every aspect of your work every day, rather than reserving the last few days of your iteration for getting stories "done done." This is an easier way to work, once you get used to it, and it reduces the risk of finding unfinished work at the end of the iteration.

> Make a little progress on every aspect of your work every day.

Use test-driven development to combine testing, coding, and designing. When working on an engineering task, make sure it integrates with the existing code. Use continuous integration and keep the 10-minute build up-to-date. Create an engineering task (see "Incremental Requirements"" in Chapter 9 for more discussion of customer reviews) for updating the installer, and have one pair work on it in parallel with the other tasks for the story.

Allies

Test-Driven Development (p. 287)

Continuous Integration (p. 183)

Ten-Minute Build (p. 177)

Just as importantly, include your on-site customers in your work. As you work on a UI task, show an on-site customer what the screen will look like, even if it doesn't work yet (see "Customer review"" in Chapter 9). Customers often want to tweak a UI when they see it for the first time. This can lead to a surprising amount of last-minute work if you delay any demos to the end of the iteration.

Similarly, as you integrate various pieces, run the software to make sure the pieces all work together. While this shouldn't take the place of testing, it's a good check to help prevent you from missing anything. Enlist the help of the testers on occasion, and ask them to show you exploratory testing techniques. (Again, this review doesn't replace real exploratory testing.)

Ally

Exploratory Testing (p. 342)

Throughout this process, you may find mistakes, errors, or outright bugs. When you do, fix them right away—then improve your work habits to prevent that kind of error from occurring again.

Ally

No Bugs (p. 159)

When you believe the story is "done done," show it to your customers for final acceptance review. Because you reviewed your progress with customers throughout the iteration, this should only take a few minutes.

Making Time

This may seem like an impossibly large amount of work to do in just one week. It's easier to do if you work on it throughout the iteration rather than saving it up for the last day or two. The real secret, though, is to make your stories small enough that you can completely finish them all in a single week.

Many teams new to XP create stories that are too large to get "done done." They finish all the coding, but they don't have enough time to completely finish the story—perhaps the UI is a little off, or a bug snuck through the cracks.

Remember, *you* are in control of your schedule. *You* decide how many stories to sign up for and how big they are. Make any story smaller by splitting it into multiple parts (see "Stories"" in Chapter 8) and only working on one of the pieces this iteration.

Creating large stories is a natural mistake, but some teams compound the problem by thinking, "Well, we really *did* finish the story, except for that one little bug." They give themselves credit for the story, which inflates their velocity and perpetuates the problem.

If you have trouble getting your stories "done done," don't count those stories toward your velocity (see "Velocity"" in Chapter 8). Even if the story only has a few minor UI bugs, count it as a zero when calculating your velocity. This will lower your velocity, which will help you choose a more manageable amount of work in your next iteration. ("Estimating"" in Chapter 8 has more details about using velocity to balance your workload.)

> If a story isn't "done done," don't count it toward your velocity.

You may find this lowers your velocity so much that you can only schedule one or two stories in an iteration. This means that your stories were too large to begin with. Split the stories you have, and work on making future stories smaller.

Questions

How does testers' work fit into "done done"?

In addition to helping customers and programmers, testers are responsible for nonfunctional testing and exploratory testing. Typically, they do these only *after* stories are "done done." They may perform some nonfunctional tests, however, in the context of a specific performance story.

Ally

Exploratory Testing (p. 342)

Regardless, the testers are part of the team, and everyone on the team is responsible for helping the team meet its commitment to deliver "done done" stories at the end of the iteration. Practically speaking, this usually means that testers help customers with customer testing, and they may help programmers and customers review the work in progress.

What if we release a story we think is "done done," but then we find a bug or stakeholders tell us they want changes?

If you can absorb the change with your iteration slack, go ahead and make the change. Turn larger changes into new stories.

Ally

Slack (p. 247)

This sort of feedback is normal—expect it. The product manager should decide whether the changes are appropriate, and if they are, he should modify the release plan. If you are constantly surprised by stakeholder changes, consider whether your on-site customers truly reflect your stakeholder community.

Results

When your stories are "done done," you avoid unexpected batches of work and spread wrap-up and polish work throughout the iteration. Customers and testers have a steady workload through the entire iteration. The final customer acceptance demonstration takes only a few minutes. At the end of each iteration, your software is ready to demonstrate to stakeholders with the scheduled stories working to their satisfaction.

Contraindications

This practice may seem advanced. It's not, but it does require self-discipline. To be "done done" every week, you must also work in iterations and use small, customer-centric stories.

In addition, you need a whole team—one that includes customers and testers (and perhaps a technical writer) in addition to programmers. The whole team must sit together. If they don't, the programmers won't be able to get the feedback they need in order to finish the stories in time.

Finally, you need incremental design and architecture and test-driven development in order to test, code, and design each story in such a short timeframe.

Allies

Iterations (p. 42)
Stories (p. 255)
Sit Together (p. 113)
Incremental Design and Architecture (p. 323)
Test-Driven Development (p. 287)

Alternatives

This practice is the cornerstone of XP planning. If you aren't "done done" at every iteration, your velocity will be unreliable. You won't be able to ship at any time. This will disrupt your release planning and prevent you from meeting your commitments, which will in turn damage stakeholder trust. It will probably lead to increased stress and pressure on the team, hurt team morale, and damage the team's capacity for energized work.

The alternative to being "done done" is to fill the end of your schedule with make-up work. You will end up with an indeterminate amount of work to fix bugs, polish the UI, create an installer, and so forth. Although many teams operate this way, it will damage your credibility and your ability to deliver. I don't recommend it.

No Bugs

Audience

Whole Team

We confidently release without a dedicated testing phase.

Let's cook up a bug pie. First, start with a nice, challenging language. How about C? We'll season it with a dash of assembly.

Next, add extra bugs by mixing in concurrent programming. Our old friends Safety and Liveness are happy to fight each other over who provides the most bugs. They supplied the Java multithreading library with bugs for years!

Now we need a really difficult problem domain. How about real-time embedded systems?

To top off this disaster recipe, we need the right kind of programmers. Let's see... experts... no... senior developers... no... aha! Novices! Just what we need.

Take your ingredients—C, real-time embedded systems, multitasking, and don't forget the novices—add a little assembly for seasoning, shake well, and bake for three years. (I do love me a bug pie.)

Here's how it turns out:

> The GMS team delivered this product after three years of development [60,638 lines of code], having encountered a total of 51 defects during that time. The open bug list never had more than two items at a time. Productivity was measured at almost three times the level for comparable embedded software teams. The first field test units were delivered after approximately six months into development. After that point, the software team supported the other engineering disciplines while continuing to do software enhancements.*

These folks had everything stacked against them—except their coach and her approach to software development. If they can do it, so can you.

How Is This Possible?

If you're on a team with a bug count in the hundreds, the idea of "no bugs" probably sounds ridiculous. I'll admit: "no bugs" is an ideal to strive for, not something your team will necessarily achieve.

However, XP teams *can* achieve dramatically lower bug rates. [Van Schooenderwoert]'s team averaged one and a half bugs per month in a very difficult domain. In an independent analysis of a company practicing a variant of XP, QSM Associates reported an average reduction from 2,270 defects to 381 defects [Mah].

You might think improvements like this are terribly expensive. They're not. In fact, agile teams tend to have above-average productivity.†

Evidence for these results is as yet anecdotal. Scientific studies of complete software development methodologies are rare due to the large number of variables involved. While there are organizations that draw performance conclusions by aggregating hundreds of projects, none that I am aware of have enough data to draw conclusions on agile projects in general, let alone XP specifically. (QSM Associates is a well-regarded example of such an organization; as of June 2006, they only had data from a few agile projects.*)

In the absence of conclusive proof, how can you know if your team will achieve these results? There's only one way to know for sure: try it and see. It doesn't take superpowers. Teams of novices coached by an experienced developer have done it. All you need is *commitment* to follow the XP practices rigorously and *support* from your organization to do so.

* "Embedded Agile Project by the Numbers with Newbies" [Van Schooenderwoert].

† See, for example, [Van Schooenderwoert], [Mah], and [Anderson 2006].

* Personal communication with Michael Mah of QSM Associates.

How to Achieve Nearly Zero Bugs

Many approaches to improving software quality revolve around finding and removing more defects† through traditional testing, inspection, and automated analysis.

The agile approach is to *generate fewer defects*. This isn't a matter of finding defects earlier; it's a question of not generating them at all.

For example, [Van Schooenderwoert] delivered 21 bugs to customers. Working from Capers Jones' data, Van Schooenderwoert says that a "best in class" team building their software would have generated 460 defects, found 95 percent of them, and delivered 23 to their customer.‡ In contrast, Van Schooenderwoert's team generated 51 defects, found 59 percent of them, and delivered 21 to their customer. At 0.22 defects per function point, this result far exceeds Capers Jones' best-in-class expectation of two defects per function point.

To achieve these results, XP uses a potent cocktail of techniques:

1. *Write fewer bugs* by using a wide variety of technical and organizational practices.
2. *Eliminate bug breeding grounds* by refactoring poorly designed code.
3. *Fix bugs quickly* to reduce their impact, write tests to prevent them from reoccurring, then fix the associated design flaws that are likely to breed more bugs.
4. *Test your process* by using exploratory testing to expose systemic problems and hidden assumptions.
5. *Fix your process* by uncovering categories of mistakes and making those mistakes impossible.

This may seem like a lot to do, but most of it comes naturally as part of the XP process. Most of these activities improve productivity by increasing code quality or by removing obstacles. If you do them as part of XP, you won't have to do many of the other more expensive activities that other teams perform, such as an exhaustive upfront requirements gathering phase, disruptive code inspections, or a separate bug-fixing phase.

Ingredient #1: Write Fewer Bugs

Don't worry—I'm not going to wave my hands and say, "Too many bugs? No problem! Just write fewer bugs!" To stop writing bugs, you have to take a rigorous, thoughtful approach to software development.

† I use "defect" synonymously with "bug."

‡ An "average" team would have generated 1,035, found 80 percent, and delivered 207.

Start with test-driven development (TDD), which is a proven technique for reducing the number of defects you generate [Janzen & Saiedian]. It leads to a comprehensive suite of unit and integration tests, and perhaps more importantly, it structures your work into small, easily verifiable steps. Teams using TDD report that they rarely need to use a debugger.

Allies

- Test-Driven Development (p. 287)
- Energized Work (p. 82)
- Pair Programming (p. 74)
- Sit Together (p. 113)
- Customer Tests (p. 280)
- Exploratory Testing (p. 342)
- Iteration Demo (p. 138)
- Coding Standards (p. 133)
- "Done Done" (p. 155)

To enhance the benefits of test-driven development, work sensible hours and program all production code in pairs. This improves your brainpower, which helps you make fewer mistakes and allows you to see mistakes more quickly. Pair programming also provides positive peer pressure, which helps you maintain the self-discipline you need to follow defect-reduction practices.

Test-driven development helps you eliminate coding defects, but code isn't your only source of defects. You can also produce good code that does the wrong thing. To prevent these requirements-oriented defects, work closely with your stakeholders. Enlist on-site customers to sit with your team. Use customer tests to help communicate complicated domain rules. Have testers work with customers to find and fix gaps in their approach to requirements. Demonstrate your software to stakeholders every week, and act on their feedback.

Supplement these practices with good coding standards and a "done done" checklist. These will help you remember and avoid common mistakes.

Ingredient #2: Eliminate Bug Breeding Grounds

Writing fewer bugs is an important first step to reducing the number of defects your team generates. If you accomplish that much, you're well ahead of most teams. Don't stop now, though. You can generate even fewer defects.

Even with test-driven development, your software will accumulate technical debt over time. Most of it will be in your design, making your code defect-prone and difficult to change, and it will tend to congregate in specific parts of the system. According to [Boehm], about 20 percent of the modules in a program are typically responsible for about 80 percent of the errors.

> Technical debt breeds bugs.

These design flaws are unavoidable. Sometimes a design that looks good when you first create it won't hold up over time. Sometimes a shortcut that seems like an acceptable compromise will come back to bite you. Sometimes your requirements change and your design will need to change as well.

Whatever its cause, technical debt leads to complicated, confusing code that's hard to get right. It breeds bugs. To generate fewer defects, pay down your debt.

Although you could dedicate a week or two to fixing these problems, the best way to pay off your debt is to make small improvements every week. Keep new code clean by creating simple designs and refactoring as you go. Use the slack in each iteration to pay down debt in old code.

Allies

- Simple Design (p. 316)
- Refactoring (p. 306)
- Slack (p. 247)

Ingredient #3: Fix Bugs Now

Programmers know that the longer you wait to fix a bug, the more it costs to fix [McConnell 1996] (p. 75). In addition, unfixed bugs probably indicate further problems. Each bug is the

result of a flaw in your system that's likely to breed more mistakes. Fix it now and you'll improve both quality and productivity.

To fix the bug, start by writing an automated test that demonstrates the bug. It could be a unit test, integration test, or customer test, depending on what kind of defect you've found. Once you have a failing test, fix the bug. Get a green bar.

Don't congratulate yourself yet—you've fixed the problem, but you haven't solved the underlying cause. Why did that bug occur? Discuss the code with your pairing partner. Is there a design flaw that made this bug possible? Can you change an API to make such bugs more obvious? Is there some way to refactor the code that would make this kind of bug less likely? Improve your design. If you've identified a systemic problem, discuss it with the rest of your team in your next stand-up meeting or iteration retrospective. Tell people what went wrong so they can avoid that mistake in the future.

Fixing bugs quickly requires the whole team to participate. Programmers, use collective code ownership so that any pair can fix a buggy module. Customers and testers, personally bring new bugs to the attention of a programmer and help him reproduce it. These actions are easiest when the whole team sits together.

Allies

Collective Code Ownership (p. 191)
Sit Together (p. 113)

In practice, it's not possible to fix every bug right away. You may be in the middle of working on something else when you find a bug. When this happens to me, I ask my navigator to write the problem on our to-do list. We come back to it 10 or 20 minutes later, when we come to a good stopping point.

Some bugs are too big to fix while you're in the middle of another task. For these, I write the bug on a story card and announce it to the team. (Some teams use red story cards for this purpose.) We collectively decide if there's enough slack in the iteration to fix the bug and still meet our other commitments. If there is, we create tasks for the newly created story and pairs volunteer for them as normal. Sometimes your only task will be "fix the bug." I use the story card as its own task card when this happens.

Ally

Slack (p. 247)

If there *isn't* enough slack to fix the bug, estimate the cost to fix it and ask your product manager to decide whether to fix it in this release. If it's important enough to fix, schedule it into the very next iteration.

NOTE

Although you may wish to fix every bug right away, you need to consider its cost and value before doing so. Some bugs are expensive to fix but have limited impact; these are often not worth fixing. However, because bugs generally become more expensive to fix over time, you should typically only choose between "fix" (as soon as possible) or "don't fix" (until a later release).

DON'T PLAY THE BUG BLAME GAME

FEATURE

—License plate seen on a Volkswagen Beetle

If you tell a programmer there's a bug in his software, you're likely to get a prickly response: "That's not a bug, it's a feature!" or "What did you do wrong?"

I suspect this is because "bug" usually means "you screwed up."* Arguing about whether something is a bug often seems to be a fancy approach to finger-pointing. Some folks have taken this to a high art, making elaborate distinctions between bugs, defects, faults, errors, issues, anomalies, and of course, unintentional features. I prefer to avoid formal categorizations. What really matters is whether you will *do* or *not do* something, regardless of whether that something is fixing a bug or implementing a feature.

Mistakes do occur, of course, and we want to prevent those mistakes. Pointing fingers is counterproductive. The *whole team*—on-site customers, testers, and programmers—is responsible for delivering valuable software. Regardless of who made the mistake, the whole team is responsible for preventing it from reaching stakeholders. To that end, when I think about bugs, I focus on what the team delivers, not where the bug came from.

NOTE

A *bug* or *defect* is any behavior of your software that will unpleasantly surprise important stakeholders.

Before you yell "bug!", however, there are a few things to know:

- If the team finds and fixes a problem as part of its normal work—in other words, before a story is "done done"—it's not a bug.
- If the on-site customers have intentionally decided that the software should behave this way, it's not a bug. (Some of your software's behavior will be unpleasant to stakeholders—you can't please all people all the time—but hopefully it won't be a surprise.)

Remembering these things won't help you settle contract disputes, so don't use them for formal bug categorization. Instead, change the way you think about bugs. Stop worrying about who made what mistake, and start focusing on what you can do, as a team, to increase value by delivering fewer bugs.

Ingredient #4: Test Your Process

These practices will dramatically cut down the number of bugs in your system. However, they only prevent bugs you expect. Before you can write a test to prevent a problem, you have to realize the problem can occur.

Exploratory testing, in the hands of a good—some may say diabolical—tester, is an invaluable way to broaden your understanding of the types of problems that can occur. An exploratory tester uses her intuition and experience to tell her what kinds of problems programmers and customers have the most trouble considering. For example, she might unplug her network cable in the middle of an operation or perform a SQL injection attack on your database.

Ally

Exploratory Testing (p. 342)

* Thanks to Ron Jeffries for this insight.

If your team has typical adversarial relationships between programmers, customers, and testers, these sorts of unfair tests might elicit bitter griping from programmers. Remember, though—you're all in this together. The testers expose holes in your thought process and, by doing so, save you from having to make uncomfortable apologies to stakeholders or from dramatic failures in production. Congratulate your testers on their ingenuity—and as you write tests, remember the problems they have exposed.

Exploratory testing is a very effective way of finding unexpected bugs. It's so effective that the rest of the team might start to get a little lazy.

> Think of exploratory testing as testing your process, not your software.

Don't rely on exploratory testing to find bugs in your software. (Really!) Your primary defense against bugs is test-driven development and all the other good practices I've mentioned. Instead, use exploratory testing to test your *process*. When an exploratory test finds a bug, it's a sign that your work habits—your process—have a hole in them. Fix the bug, then fix your process.

Testers, only conduct exploratory testing on stories that the team agrees are "done done." Programmers and customers, if your testers find any problems, think of them as bugs. Take steps to prevent them from occurring, just as you would for any other bug. Aim for a defect rate of under one or two bugs per month *including* bugs found in exploratory testing.

A good exploratory tester will find more bugs than you expect. To make the bug rate go down, fix your process.

Ingredient #5: Fix Your Process

Some bugs occur because we're human. (D'oh!) More often, bugs indicate an unknown flaw in our approach. When the number of bugs you generate gets low enough, you can do something usually associated with NASA's Space Shuttle software: root-cause analysis and process improvement on every bug.

> NOTE
> **Even if your defect count isn't low enough to allow you to do root-cause analysis of every bug, consider randomly picking a bug each iteration for further analysis.**

When you fix a bug, start by writing an automated test and improving your design to make the bug less likely. This is the beginning of root-cause analysis, but you can go even further.

As you write the test and fix the design, ask questions. Why was there no test preventing this bug? Why does the design need fixing? Use the "five whys" technique to consider the root cause. Then, as a team, discuss possible root causes and decide how best to change your work habits to make that kind of bug more difficult.

Ally

Root-Cause Analysis (p. 91)

MAKING MISTAKES IMPOSSIBLE

The best way to fix your process is to make mistakes impossible. For example, if you have a problem with UI field lengths being inconsistent with database field lengths, you could change your design to base one value off the other.

The next best solution is to find mistakes automatically. For example, you could write a test to check all UI and database fields for consistency and run that test as part of every build.

The least effective approach (but better than nothing!) is to add a manual step to your process. For example, you could add "check modified database/UI fields for length consistency" to your "done done" checklist.

Next, ask yourselves if the root cause of *this* bug could also have led to *other* bugs you haven't yet found. Testers, ask yourselves if this type of mistake reveals a blind spot in the team's thinking. Perform additional testing to explore these questions.

Invert Your Expectations

If you follow these practices, bugs should be a rarity. Your next step is to treat them that way. Rather than shrugging your shoulders when a bug occurs—"Oh yeah, another bug, that's what happens in software"—be shocked and dismayed. "That's our fourth bug this month! Why do we have so many bugs?"

I'm not so naïve as to suggest that the power of belief will eliminate bugs, but if you're already close, the shift in attitude will help you make the leap from *reducing* bugs to *nearly eliminating* bugs. You'll spend more energy on discovering and fixing root causes.

For this reason, I recommend that new XP teams not install a bug database. If you're only generating a bug or two per month, you don't need a database to keep track of your bugs; you can just process them as they come in. Explicitly *not* providing a database helps create the attitude that bugs are abnormal. It also removes the temptation to use the database as a substitute for direct, personal communication.

Depending on your industry, you may need a formal way to track defects so a database may be appropriate. However, I never assume a database will be necessary until the requirements of the situation *prove* it. Even then, I look for ways to use our existing process to meet the regulatory or organizational requirements. Although some shudder at the informality, archiving red bug cards in a drawer may suffice.

Even if you are able to eliminate your bug database, you still need to be able to reproduce bugs. You may need screenshots, core dumps, and so forth. If you fix bugs as soon as they come in, you may be able to work directly out of your email inbox. Otherwise, you can keep this information in the same place that you keep other requirements details. (See "Incremental Requirements"" in Chapter 9.)

Questions

If novices can do this, what's stopping management from firing everybody and hiring novices?

All else remaining equal, experienced developers will always produce better results and more quickly than novices. If you have the option to bring high-quality people into your team, do it.

The point is that these practices are within the grasp of average teams—even teams of novices, as long as they have an experienced coach. They won't achieve zero bugs, but they are likely to achieve better results than they would otherwise.

How do we prevent security defects and other challenging bugs?

The approach I've described only prevents bugs you think to prevent. Security, concurrency, and other difficult problem domains may introduce defects you never considered.

In this situation, using exploratory testing to test and fix your process is even more important. You may need to hire a specialist tester, such as a security expert, to probe for problems and teach the team how to prevent such problems in the future.

How long should we spend working on a bug before we convert it into a story?

It depends on how much slack you have left in the iteration. Early in the iteration, when there's still a lot of slack, I might spend as much as four pair-hours on a defect. Later, when there's less slack, I might only spend 10 minutes on it.

Ally
Slack (p. 247)

Bugs are usually harder to find than to fix, so enlist the help of the testers. The fix often takes mere minutes once you've isolated the problem.

If our guideline is to fix bugs as soon as we find them, won't we have the unconscious temptation to overlook bugs?

Perhaps. This is one reason we pair: pairing helps us maintain our team discipline. If you find yourself succumbing to the temptation to ignore a bug, write it on a story card rather than let it slide by. Tell the rest of the team about the bug, and ask somebody to volunteer to fix it.

If we don't find any bugs, how do we know that our testers are doing exploratory testing correctly?

It's a bit of conundrum: the team is supposed to prevent bugs from occurring in "done done" stories, so exploratory testing shouldn't find anything. Yet if exploratory testing doesn't find anything, you could be testing the wrong things.

If bugs are particularly devastating for your project, ask an independent testing group to test a few of your iteration releases. If they don't find anything surprising, then you can have confidence in your exploratory testing approach.

This is probably overkill for most teams. If you're following the practices and your testers haven't found anything, you can comfortably release your software. Reevaluate your approach if your stakeholders or customers find a significant bug.

We have a large amount of legacy code. How can we adopt this policy without going mad?

Most legacy code doesn't have any tests and is chock-full of bugs. You can dig your way out of this hole, but it will take a lot of time and effort. See "Applying XP to an Existing Project"" in Chapter 4 for details.

Results

When you produce nearly zero bugs, you are confident in the quality of your software. You're comfortable releasing your software to production without further testing at the end of any iteration. Stakeholders, customers, and users rarely encounter unpleasant surprises, and you spend your time producing great software instead of fighting fires.

Contraindications

"No Bugs" depends on the support and structure of all of XP. To achieve these results, you need to practice nearly all of the XP practices rigorously:

- All the "Thinking" practices are necessary (Pair Programing, Energized Work, Informative Workspace, Root-Cause Analysis, and Retrospectives); they help you improve your process, and they help programmers notice mistakes as they code.
- All the "Collaborating" practices except "Reporting" are necessary (Trust, Sit Together, Real Customer Involvement, Ubiquitous Language, Stand-Up Meetings, Coding Standards, and Iteration Demo); most help prevent requirements defects, and the rest help programmers coordinate with each other.
- All the "Releasing" practices except "Documentation" are necessary ("Done Done," No Bugs, Version Control, Ten-Minute Build, Continuous Integration, and Collective Code Ownership); most help keep the code organized and clean. "Done Done" helps prevent inadvertent omissions.
- All the "Planning" practices except "Risk Management" are necessary (Vision, Release Planning, The Planning Game, Iteration Planning, Slack, Stories, and Estimating); they provide structure and support for the other practices.
- All the "Developing" practices except "Spike Solutions" are necessary (Test-Driven Development, Refactoring, Simple Design, Incremental Design and Architecture, Performance Optimization, Customer Reviews, Customer Testing, Exploratory Testing); they improve design quality, reduce requirements defects, and provide a way for testers to be involved in defect prevention as well as defect detection.

If you aren't using all these practices, don't expect dramatic reductions in defects. Conversely, if you have a project that's in XP's sweet spot (see "Is XP Right for Us?"" in Chapter 4) and you're using all the XP practices, more than a few bugs per month may indicate a problem with your approach to XP. You need time to learn the practices, of course, but if you don't see improvements in your bug rates within a few months, consider asking for help (see "Find a Mentor"" in Chapter 2).

Alternatives

You can also reduce bugs by using more and higher quality testing (including inspection or automated analysis) to find and fix a higher percentage of bugs. However, testers will need some time to review and test your code, which will prevent you from being "done done" and ready to ship at the end of each iteration.

Further Reading

"Embedded Agile Project by the Numbers with Newbies" [Van Schooenderwoert] expands on the embedded C project described in the introduction.

Version Control

Audience
Programmers

We keep all our project artifacts in a single, authoritative place.

To work as a team, you need some way to coordinate your source code, tests, and other important project artifacts. A *version control system* provides a central repository that helps coordinate changes to files and also provides a history of changes.

A project without version control may have snippets of code scattered among developer machines, networked drives, and even removable media. The build process may involve one or more people scrambling to find the latest versions of several files, trying to put them in the right places, and only succeeding through the application of copious caffeine, pizza, and stress.

A project *with* version control uses the version control system to mediate changes. It's an orderly process in which developers get the latest code from the server, do their work, run all the tests to confirm their code works, then check in their changes. This process, called *continuous integration*, occurs several times a day for each pair.

Ally
Continuous Integration (p. 183)

NOTE

If you aren't familiar with the basics of version control, start learning now. Learning to use a version control system effectively may take a few days, but the benefits are so great that it is well worth the effort.

VERSION CONTROL TERMINOLOGY

Different version control systems use different terminology. Here are the terms I use throughout this book:

Repository
: The repository is the master storage for all your files and and their history. It's typically stored on the version control server. Each standalone project should have its own repository.

Sandbox
: Also known as a working copy, a sandbox is what team members work out of on their local development machines. (Don't ever put a sandbox on a shared drive. If other people want to develop, they can make their own sandbox.) The sandbox contains a copy of all the files in the repository from a particular point in time.

Check out
: To create a sandbox, check out a copy of the repository. In some version control systems, this term means "update and lock."

Update
: Update your sandbox to get the latest changes from the repository. You can also update to a particular point in the past.

Lock
: A lock prevents anybody from editing a file but you.

Check in or commit
: Check in the files in your sandbox to save them into the repository.

Revert
: Revert your sandbox to throw away your changes and return to the point of your last update. This is handy when you've broken your local build and can't figure out how to get it working again. Sometimes reverting is faster than debugging, especially if you have checked in recently.

Tip or head
: The tip or head of the repository contains the latest changes that have been checked in. When you update your sandbox, you get the files at the tip. (This changes somewhat when you use branches.)

Tag or label
: A tag or label marks a particular time in the history of the repository, allowing you to easily access it again.

Roll back
: Roll back a check-in to remove it from the tip of the repository. The mechanism for doing so varies depending on the version control system you use.

Branch
: A branch occurs when you split the repository into distinct "alternate histories," a process known as *branching*. All the files exist in each branch, and you can edit files in one branch independently of all other branches.

Merge
: A merge is the process of combining multiple changes and resolving any conflicts. If two programmers change a file separately and both check it in, the second programmer will need to merge in the first person's changes.

Concurrent Editing

If multiple developers modify the same file without using version control, they're likely to accidentally overwrite each other's changes. To avoid this pain, some developers turn to a *locking model* of version control: when they work on a file, they lock it to prevent anyone else from making changes. The files in their sandboxes are read-only until locked. If you have to check out a file in order to work on it, then you're using a locking model.

While this approach solves the problem of accidentally overwriting changes, it can cause other, more serious problems. A locking model makes it difficult to make changes. Team members have to carefully coordinate who is working on which file, and that stifles their ability to refactor and make other beneficial changes. To get around this, teams often turn to strong code ownership, which

Ally

Collective Code Ownership (p. 191)

is the worst of the code ownership models because only one person has the authority to modify a particular file. Collective code ownership is a better approach, but it's very hard to do if you use file locking.

Instead of a locking model, use a *concurrent model* of version control. This model allows two people to edit the same file simultaneously. The version control system automatically merges their changes—nothing gets overwritten accidentally. If two people edit the exact same lines of code, the version control system prompts them to merge the two lines manually.

Automatic merges may seem risky. They *would* be risky if it weren't for continuous integration and the automated build. Continuous integration reduces the scope of merges to a manageable level, and the build, with its comprehensive test suite, confirms that merges work properly.

Allies

Continuous Integration (p. 183)
Ten-Minute Build (p. 177)

Time Travel

One of the most powerful uses of a version control system is the ability to go back in time. You can update your sandbox with all the files from a particular point in the past.

This allows you to use *diff debugging*. When you find a challenging bug that you can't debug normally, go back in time to an old version of the code when the bug didn't exist. Then go forward and backward until you isolate the exact check-in that introduced the bug. You can review the changes in that check-in alone to get insight into the cause of the bug. With continuous integration, the number of changes will be small.

NOTE

A powerful technique for diff debugging is the *binary chop*, in which you cut the possible number of changes in half with each test. If you know that version 500 doesn't have the bug and your current version, 700, does, then check version 600. If the bug is present in version 600, it was introduced somewhere between 500 and 599, so now check version 550. If the bug is *not* present in version 600, it was introduced somewhere between version 601 and 700, so check version 650. By applying a bit of logic and halving the search space each time, you can quickly isolate the exact version.

A bit of clever code could even automate this search with an automated test.*

Time travel is also useful for reproducing bugs. If somebody reports a bug and you can't reproduce it, try using the same version of the code that the reporter is using. If you can reproduce the behavior in the old version but not in the current version, especially with a unit test, you can be confident that the bug is and will remain fixed.

Whole Project

It should be obvious that you should store your source code in version control. It's less obvious that you should store everything else in there, too. Although most version control systems allow you to go back in time, it doesn't do you any good unless you can build the exact version

* Thanks to Andreas Kö for demonstrating this.

you had at that time. Storing the whole project in version control—including the build system—gives you the ability to re-create old versions of the project in full.

As much as possible, keep all your tools, libraries, documentation, and everything else related to the project in version control. Tools and libraries are particularly important. If you leave them out, at some point you'll update one of them, and then you'll no longer be able to go back to a time before the update. Or, if you do, you'll have to painstakingly remember which version of the tool you *used* to use and manually replace it.

For similar reasons, store the entire project in a single repository. Although it may seem natural to split the project into multiple repositories—perhaps one for each deliverable, or one for source code and one for documentation—this approach increases the opportunities for things to get out of sync.

Perform your update and commit actions on the whole tree as well. Typically, this means updating or committing from the top-level directory. It may be tempting to commit only the directory you've been working in, but that leaves you vulnerable to the possibility of having your sandbox split across two separate versions.

The only project-related artifact I *don't* keep in version control is generated code. Your automated build should re-create generated code automatically.

> Leave generated code out of the repository.

There is one remaining exception to what belongs in version control: code you plan to throw away. Spike solutions (see "Spike Solutions"" in Chapter 9), experiments, and research projects may remain unintegrated, unless they produce concrete documentation or other artifacts that will be useful for the project. Check in only the useful pieces of the experiment. Discard the rest.

Customers and Version Control

Customer data should go in the repository, too. That includes documentation, notes on requirements (see "Incremental Requirements"" in Chapter 9), technical writing such as manuals, and customer tests (see "Customer Tests"" in Chapter 9).

When I mention this to programmers, they worry that the version control system will be too complex for customers to use. Don't underestimate your customers. While it's true that some version control systems are very complex, most have user-friendly interfaces. For example, the TortoiseSvn Windows client for the open-source Subversion version control system is particularly nice.

Even if your version control system is somewhat arcane, you can always create a pair of simple shell scripts or batch files—one for update and one for commit—and teach your customers how to run them. If you sit together, you can always help your customers when they need to do something more sophisticated, such as time travel or merging.

Keep It Clean

> Always check in code that builds and passes all tests.

One of the most important ideas in XP is that you keep the code clean and ready to ship. It starts with your sandbox. Although you have to break the build in your sandbox in order to make progress, confine it to your sandbox. *Never* check in code that breaks the build. This allows anybody to update at any time without worrying about breaking their build—and that, in turn, allows everyone to work smoothly and share changes easily.

> NOTE
> **You can even minimize broken builds in your sandbox. With good test-driven development, you're never more than five minutes away from a working build.**

Because your build automatically creates a release, any code that builds is theoretically ready to release. In practice, the code may be clean but the software itself won't be ready for the outside world. Stories will be half-done, user interface elements will be missing, and some things won't entirely work.

By the end of each iteration, you will have finished all these loose ends. Each story will be "done done," and you will deploy the software to stakeholders as part of your iteration demo. This software represents a genuine increment of value for your organization. Make sure you can return to it at any time by *tagging* the tip of the repository. I usually name mine "Iteration *X*," where *X* is the number of the iterations we have conducted.

Not every end-of-iteration release to stakeholders gets released to customers. Although it contains completed stories, it may not have enough to warrant a release. When you conduct an actual release, add another tag to the end-of-iteration build to mark the release. I usually name mine "Release *Y*," where *Y* is the number of releases we have conducted.

> NOTE
> **Although your build should theoretically work from any sandbox, save yourself potential headaches by performing release builds from a new, pristine sandbox. The first time you spend an hour discovering that you've broken a release build by accidentally including an old file, you'll resolve never to do it again.**

To summarize, your code goes through four levels of completion:

1. *Broken.* This only happens in your sandbox.
2. *Builds and passes all tests.* All versions in your repository are at least at this level.
3. *Ready to demo to stakeholders.* Any version marked with the "Iteration *X*" tag is ready for stakeholders to try.
4. *Ready to release to real users and customers.* Any version marked with the "Release *Y*" tag is production-ready.

Single Codebase

One of the most devastating mistakes a team can make is to duplicate their codebase. It's easy to do. First, a customer innocently requests a customized version of your software. To deliver this version quickly, it seems simple to duplicate the codebase, make the changes, and ship it. Yet that *copy and paste customization* doubles the number of lines of code that you need to maintain.

I've seen this cripple a team's ability to deliver working software on a timely schedule. It's nearly impossible to recombine a duplicated codebase without heroic and immediate action. That one click doesn't just lead to technical debt; it leads to indentured servitude.

> Duplicating your codebase will cripple your ability to deliver.

Unfortunately, version control systems actually make this mistake easier to make. Most of these systems provide the option to *branch* your code—that is, to split the repository into two separate lines of development. This is essentially the same thing as duplicating your codebase.

Branches have their uses, but using them to provide multiple customized versions of your software is risky. Although version control systems provide mechanisms for keeping multiple branches synchronized, doing so is tedious work that steadily becomes more difficult over time. Instead, design your code to support multiple configurations. Use a plug-in architecture, a configuration file, or factor out a common library or framework. Top it off with a build and delivery process that creates multiple versions.

Appropriate Uses of Branches

Branches work best when they are short-lived or when you use them for small numbers of changes. If you support old versions of your software, a branch for each version is the best place to put bug fixes and minor enhancements for those versions.

Some teams create a branch in preparation for a release. Half the team continues to perform new work, and the other half attempts to stabilize the old version. In XP, your code shouldn't require stabilization, so it's more useful to create such a branch at the point of release, not in preparation for release.

NOTE

To eliminate the need for a branch entirely, automatically migrate your customers and users to the latest version every time you release.

Branches can also be useful for continuous integration and other code management tasks. These *private branches* live for less than a day. You don't need private branches to successfully practice XP, but if you're familiar with this approach, feel free to use it.

Questions

Which version control system should I use?

There are plenty of options. In the open source realm, Subversion is popular and particularly good when combined with the TortoiseSvn frontend. Of the proprietary options, Perforce gets good reviews, although I haven't tried it myself.

Avoid Visual SourceSafe (VSS). VSS is a popular choice for Microsoft teams, but it has numerous flaws and problems with repository corruption—an unacceptable defect in a version control system.

Your organization may already provide a recommended version control system. If it meets your needs, use it. Otherwise, maintaining your own version control system isn't much work and requires little of a server besides disk space.

Should we really keep all our tools and libraries in version control?

Yes, as much as possible. If you install tools and libraries manually, two undesirable things will happen. First, whenever you make an update, everyone will have to manually update their computer. Second, at some point in the future you'll want to build an earlier version, and you'll spend several hours struggling to remember which versions of which tools you need to install.

Some teams address these concerns by creating a "tools and libraries" document and putting it in source control, but it's a pain to keep such a document up-to-date. Keeping your tools and libraries in source control is a simpler, more effective method.

Some tools and libraries require special installation, particularly on Windows, which makes this strategy more difficult. They don't all need installation, though—some just come with an installer because it's a cultural expectation. See if you can use them without installing them, and try to avoid those that you can't easily use without special configuration.

For tools that require installation, I put their install packages in version control, but I don't install them automatically in the build script. The same is true for tools that are useful but not necessary for the build, such as IDEs and diff tools.

How can we store our database in version control?

Rather than storing the database itself in version control, set up your build to initialize your database schema and migrate between versions. Store the scripts to do this in version control.

Ally

Ten-Minute Build (p. 177)

How much of our core platform should we include in version control?

In order for time travel to work, you need to be able to exactly reproduce your build environment for any point in the past. In theory, everything required to build should be in version control, including your compiler, language framework, and even your database management system (DBMS) and operating system (OS). Unfortunately, this isn't always practical. I include as much as I can, but I don't usually include my DBMS or operating system.

Some teams keep an image of their entire OS and installed software in version control. This is an intriguing idea, but I haven't tried it.

With so many things in version control, how can I update as quickly as I need to?

Slow updates may be a sign of a poor-quality version control system. The speed of better systems depends on the number of files that have *changed*, not the total number of files in the system.

One way to make your updates faster is to be selective about what parts of your tools and libraries you include. Rather than including the entire distribution—documentation, source

code, and all—include only the bare minimum needed to build. Many tools only need a handful of files to execute. Include distribution package files in case someone needs more details in the future.

How should we integrate source code from other projects? We have read-only access to their repositories.

If you don't intend to change their code and you plan on updating infrequently, you can manually copy their source code into your repository.

If you have more sophisticated needs, many version control systems will allow you to integrate with other repositories. Your system will automatically fetch their latest changes when you update. It will even merge your changes to their source code with their updates. Check your version control system's documentation for more details.

Be cautious of making local changes to third-party source code; this is essentially a branch, and it incurs the same synchronization challenges and maintenance overhead that any long-lived branch does. If you find yourself making modifications beyond vendor-supplied configuration files, consider pushing those changes upstream, back to the vendor, as soon as possible.

We sometimes share code with other teams and departments. Should we give them access to our repository?

Certainly. You may wish to provide read-only access unless you have well-defined ways of coordinating changes from other teams.

Results

With good version control practices, you are easily able to coordinate changes with other members of the team. You easily reproduce old versions of your software when you need to. Long after your project has finished, your organization can recover your code and rebuild it when they need to.

Contraindications

You should always use some form of version control, even on small one-person projects. Version control will act as a backup and protect you when you make sweeping changes.

Concurrent editing, on the other hand, can be dangerous if an automatic merge fails and goes undetected. Be sure you have a decent build if you allow concurrent edits. Concurrent editing is also safer and easier if you practice continuous integration and have good tests.

Allies

Ten-Minute Build (p. 177)
Continuous Integration (p. 183)
Test-Driven Development (p. 287)

Alternatives

There is no practical alternative to version control.

You may choose to use file locking rather than concurrent editing. Unfortunately, this approach makes refactoring and collective code ownership very difficult, if not impossible. You can alleviate this somewhat by keeping a list of proposed refactorings and scheduling them, but the added overhead is likely to discourage people from suggesting significant refactorings.

Further Reading

[Mason] is a good introduction to the nuts and bolts of version control that specifically focuses on Subversion.

[Sink], at *http://www.ericsink.com/scm/source_control.html*, is a helpful introduction to version control for programmers with a Microsoft background.

[Berczuk & Appleton] goes into much more detail about the ways in which to use version control.

Ten-Minute Build

Audience
Programmers

We eliminate build and configuration hassles.

Here's an ideal to strive for. Imagine you've just hired a new programmer. On the programmer's first day, you walk him over to the shiny new computer you just added to your open workspace.

"We've found that keeping everything in version control and having a really great automated build makes us a lot faster," you say. "Here, I'll show you. This computer is new, so it doesn't have any of our stuff on it yet."

You sit down together. "OK, go ahead and check out the source tree." You walk him through the process and the source tree starts downloading. "This will take a while because we have all our build tools and libraries in version control, too. Don't worry—like any good version control system, it brings down changes, so it's only slow the first time. We keep tools and libraries in version control because it allows us to update them easily. Come on, let me show you around the office while it downloads."

After giving him the tour, you come back. "Good, it's finished," you say. "Now watch this—this is my favorite part. Go to the root of the source tree and type `build`."

The new programmer complies, then watches as build information flies by. "It's not just building the source code," you explain. "We have a complex application that requires a web server, multiple web services, and several databases. In the past, when we hired a new programmer, he would spend his first couple of weeks just configuring his workstation. Test environments were even worse. We used to idle the whole team for days while we wrestled with problems in the test environment. Even when the environment worked, we all had to share one environment and we couldn't run tests at the same time.

"All that has changed. We've automated all of our setup. Anybody can build and run all the tests on their own machine, any time they want. I could even disconnect from the network right now and it would keep building. The build script is doing everything: it's configuring a local web server, initializing a local database... everything.

"Ah! It's almost done. It's built the app and configured the services. Now it's running the tests. This part used to be really slow, but we've made it much faster lately by improving our unit tests so we could get rid of our end-to-end tests."

Suddenly, everything stops. The cursor blinks quietly. At the bottom of the console is a message: `BUILD SUCCESSFUL`.

"That's it," you say proudly. "Everything works. I have so much confidence in our build and tests that I could take this and install it on our production servers today. In fact, we could do that right now, if we wanted to, just by giving our build a different command.

"You're going to enjoy working here." You give the new programmer a friendly smile. "It used to be hell getting even the simplest things done. Now, it's like butter. Everything just works."

Automate Your Build

What if you could build and test your entire product—or create a deployment package—at any time, just by pushing a button? How much easier would that make your life?

Producing a build is often a frustrating and lengthy experience. This frustration can spill over to the rest of your work. "Can we release the software?" "With a few days of work." "Does the software work?" "My piece does, but I can't build everything." "Is the demo ready?" "We ran into a problem with the build—tell everyone to come back in an hour."

Sadly, build automation is easy to overlook in the rush to finish features. If you don't have an automated build, start working on one now. It's one of the easiest things you can do to make your life better.

> Automating your build is one of the easiest ways to improve morale and increase productivity.

How to Automate

There are plenty of useful build tools available, depending on your platform and choice of language. If you're using Java, take a look at Ant. In .NET, NAnt and MSBuild are popular. Make is the old standby for C and C++. Perl, Python, and Ruby each have their preferred build tools as well.

Your build should be comprehensive but not complex. In addition to compiling your source code and running tests, it should configure registry settings, initialize database schemas, set up web servers, launch processes—everything you need to build and test your software from scratch without human intervention. Once you get the basics working, add the ability to create a production release, either by creating an install file or actually deploying to servers.

NOTE

Construct your build so that it provides a single, unambiguous result: `SUCCESS` or `FAILURE`. You will run the build dozens of times per day. Manual checks are slow, error-prone, and—after the umpteenth time—seriously annoying.

A key component of a successful automated build is the *local build*. A local build will allow you to build and test at any time without worrying about other people's activities. You'll do this every few minutes in XP, so independence is important. It will also make your builds run faster.

> You should be able to build even when disconnected from the network.

Be cautious of IDEs and other tools that promise to manage your build automatically. Their capability often begins and ends with compiling source code. Instead, take control of your build

script. Take the time to understand exactly how and why it works and when to change it. Rather than starting with a pre-made template, you might be better off creating a completely new script. You'll learn more, and you'll be able to eliminate the complexity a generic script adds.

The details are up to you. In my build scripts, I prefer to have all autogenerated content go into a single directory called *build/*. The output of each major step (such as compiling source code, running tests, collating files into a release, or building a release package) goes into a separate directory under *build/*. This structure allows me to inspect the results of the build process and —if anything goes wrong—wipe the slate clean and start over.

When to Automate

At the start of the project, in the very first iteration, set up a bare-bones build system. The goal of this first iteration is to produce the simplest possible product that exercises your entire system. That includes delivering a working—if minimal—product to stakeholders.

> Use the build script to configure the integration machine. Don't configure it manually.

Because the product is so small and simple at this stage, creating a high-quality automated build is easy. Don't try to cover all the possible build scenarios you need in the future. Just make sure you can build, test, and deploy this one simple product—even if it does little more than "Hello, world!" At this stage, deployment might be as simple as creating a *.zip* file.

Once you have the seed of your build script, it's easy to improve. Every iteration, as you add features that require more out of your build, improve your script. Use your build script every time you integrate. To make sure it stays up-to-date, never configure the integration machine manually. If you find that something needs configuration, modify the build script to configure it for you.

Ally

Continuous Integration (p. 183)

Automating Legacy Projects

If you want to add a build script to an existing system, I have good news and bad news. The good news is that creating a comprehensive build script is one of the easiest ways to improve your life. The bad news is that you probably have a bunch of technical debt to pay off, so it won't happen overnight.

As with any agile plan, the best approach is to work in small stages that provide value as soon as possible. If you have a particularly complex system with lots of components, work on one component at a time, starting with the one that's most error-prone or frustrating to build manually.

Once you've picked the right component to automate, start by getting it to compile. That's usually an easy step, and it allows you to make progress right away.

Next, add the ability to run unit tests and make sure they pass. You probably compile and run unit tests in your IDE already, so this may not seem like a big improvement. Stick with it; making your build script able to prove itself is an important step. You won't have to check the results manually anymore.

Your next step depends on what's causing you the most grief. What is the most annoying thing about your current build process? What configuration issue springs up to waste a few hours every week? Automate that. Repeat with the next-biggest annoyance until you have automated everything. Once you've finished this, congratulations! You've eliminated all your build annoyances. You're ahead of most teams: you have a good build script.

Now it's time to make a *great* build script. Take a look at how you deploy. Do you create a release package such as an installer, or do you deploy directly to the production servers? Either way, start automating the biggest annoyances in your deployment process, one at a time. As before, repeat with the next-biggest annoyance until you run out of nits to pick.

This won't happen overnight, but try to make progress every week. If you can solve one annoyance every week, no matter how small, you'll see noticeable improvement within a month. As you work on other things, try not to add new debt. Include all new work in the build script from the beginning.

Ten Minutes or Less

A great build script puts your team way ahead of most software teams. After you get over the rush of being able to build the whole system at any time you want, you'll probably notice something new: the build is slow.

With continuous integration, you integrate every few hours. Each integration involves two builds: one on your machine and one on the integration machine. You need to wait for both builds to finish before continuing because you can never let the build break in XP. If the build breaks, you have to roll back your changes.

A 10-minute build leads to a 20-minute integration cycle. That's a pretty long delay. I prefer a 10- or 15-minute integration cycle. That's about the amount of time it takes to stretch my legs, get some coffee, and talk over our work with my pairing partner.

The easiest way to keep the build under 5 minutes (with a 10-minute maximum) is to keep the build times down from the beginning. One team I worked with started to look for ways to speed up the build whenever it exceeded 100 seconds.

Many new XP teams make the mistake of letting their build get too long. If you're in that boat, don't worry. You can fix long build times in the same agile way you fix all technical debt: piece by piece, focusing on making useful progress at each step.

For most teams, their tests are the source of a slow build. Usually it's because their tests aren't focused enough. Look for common problems: are you writing end-to-end tests when you should be writing unit tests and integration tests? Do your unit tests talk to a database, network, or file system?

> Slow tests are the most common cause of slow builds.

You should be able to run about 100 unit tests per second. Unit tests should comprise the majority of your tests. A fraction (less than 10 percent) should be integration tests, which checks that two components synchronize properly. When the rest of your tests provide good coverage, only a handful—if any—need to be end-to-end tests. See "Speed Matters" in Chapter 9 for more information.

Although tests are the most common cause of slow builds, if compilation speed becomes a problem, consider optimizing code layout or using a compilation cache or incremental

compilation. You could also use a distributed compilation system or take the best machine available for use as the build master. Don't forget to take advantage of the dependency evaluation features of your build tool: you don't need to rebuild things that haven't changed.

In the worst-case scenario, you may need to split your build into a "fast" build that you run frequently and a "slow" build that an integration server runs when you check in (see "Continuous Integration"," later in this chapter). Be careful—this approach leads to more build failures than a single, fast build does. Keep working on making your build faster.

Questions

Who's responsible for maintaining the build script?

All the programmers are responsible for maintaining the script. As the codebase evolves, the build script should evolve with it.

At first, one person will probably be more knowledgeable about the script than others. When you need to update the script, pair with this person and learn all you can.

The build script is the center of your project automation universe. The more you know about how to automate your builds, the easier your life will become and the faster you'll be able to get work done.

We have a configuration management (CM) department that's responsible for maintaining our builds. We aren't allowed to modify the script ourselves. What do we do?

You need to be able to update your scripts continuously to meet your specific needs. It's unlikely that anybody can be more responsive to your needs than you are. If the CM department is a bottleneck, ask your project manager for help. He may be able to give you control over the scripts.

Alternatively, you might be able to use a two-stage build in which you run your own scripts privately before handing over control to the CM department.

How do we find time to improve our build?

Improving your build directly improves your productivity and quality of life. It's important enough to include in every iteration as part of your everyday work. The best way to do this is to include enough slack in your iteration for taking care of technical debt such as slow builds. If a particular story will require changes to the build script, include that time in your story estimate.

Ally

Slack (p. 247)

Should we really keep all our tools and libraries in version control?

Yes, as much as possible. See "Version Control"" earlier in this chapter for details.

Does the build have to be under 10 minutes? We're at 11.

Ten minutes is a good rule of thumb. Your build is too long when pairs move on to other tasks before the integration cycle completes.

We use an IDE with an integrated build system. How can we automate our build process?

Many IDEs use an underlying build script that you can control. If not, you may be better off using a different IDE. Your other alternative is to have a separate command line–based build system, such as Ant, NAnt, or make. You risk duplicating information about dependencies, but sometimes that cost is worthwhile.

We have different target and development environments. How do we make this build work?

If possible, use a cross compiler. If that doesn't work, consider using a cross-platform build tool. The benefits of testing the build on your development platform outweigh the initial work in creating a portable system.

How can we build our entire product when we rely on third-party software and hardware?

Even if your product relies on yet-to-be-built custom hardware or unavailable third-party systems, you still need to build and test your part of the product. If you don't, you'll discover a ton of integration and bug-fixing work when the system becomes available.

A common solution for this scenario is to build a simulator for the missing system, which allows you to build integration tests. When the missing system becomes available, the integration tests help you determine if the assumptions you built into the simulator were correct.

Missing components add risk to your project, so try to get your hands on a test system as soon as possible.

How often should we build from scratch?

At least once per iteration. Building from scratch is often much slower than an incremental build, so it depends on how fast the build is and how good your build system is. If you don't trust your build system, build from scratch more often. You can set up a smoke-testing system that builds the project from scratch on every check-in.

My preference is to reduce build times so that incremental builds are unnecessary, or to fix the bugs in the build system so I trust the incremental builds. Even so, I prefer to build from scratch before delivering to customers.

Results

With a good automated build, you can build a release any time you want. When somebody new joins the team, or when you need to wipe a workstation and start fresh, it's a simple matter of downloading the latest code from the repository and running the build.

When your build is fast and well-automated, you build and test the whole system more frequently. You catch bugs earlier and, as a result, spend less time debugging. You integrate your software frequently without relying on complex background build systems, which reduces integration problems.

Contraindications

Every project should have a good automated build. Even if you have a system that's difficult to build, you can start chipping away at the problem today.

Some projects are too large for the 10-minute rule to be effective. Before you assume this is true for your project, take a close look at your build procedures. You can often reduce the build time much more than you realize.

Alternatives

If the project truly is too large to build in 10 minutes, it's probably under development by multiple teams or subteams. Consider splitting the project into independent pieces that you can build and test separately.

If you can't build your system in less than 10 minutes (yet), establish a maximum acceptable threshhold and stick to it. Drawing this line helps identify a point beyond which you will not allow more technical debt to accumulate. Like a sink full of dishes two hours before a dinner party, the time limit is a good impetus to do some cleaning.

Continuous Integration

Audience
Programmers

We keep our code ready to ship.

Most software development efforts have a hidden delay between when the team says "we're done" and when the software is actually ready to ship. Sometimes that delay can stretch on for months. It's the little things: merging everyone's pieces together, creating an installer, prepopulating the database, building the manual, and so forth. Meanwhile, the team gets stressed out because they forgot how long these things take. They rush, leave out helpful build automation, and introduce more bugs and delays.

Continuous integration is a better approach. It keeps everybody's code integrated and builds release infrastructure along with the rest of the application. The ultimate goal of continuous integration is to be able to deploy all but the last few hours of work at any time.

> The ultimate goal is to be able to deploy at any time.

Practically speaking, you won't actually release software in the middle of an iteration. Stories will be half-done and features will be incomplete. The point is to be *technologically* ready to release even if you're not *functionally* ready to release.

Why It Works

If you've ever experienced a painful multiday (or multiweek) integration, integrating every few hours probably seems foolish. Why go through that hell so often?

Actually, short cycles make integration *less* painful. Shorter cycles lead to smaller changes, which means there are fewer chances for your changes to overlap with someone else's.

That's not to say collisions don't happen. They do. They're just not very frequent because everybody's changes are so small.

> NOTE
> **Collisions are most likely when you're making wide-ranging changes. When you do, let the rest of the team know beforehand so they can integrate their changes and be ready to deal with yours.**

How to Practice Continuous Integration

In order to be ready to deploy all but the last few hours of work, your team needs to do two things:

1. Integrate your code every few hours.
2. Keep your build, tests, and other release infrastructure up-to-date.

To integrate, update your sandbox with the latest code from the repository, make sure everything builds, then commit your code back to the repository. You can integrate any time you have a successful build. With test-driven development, that should happen every few minutes. I integrate whenever I make a significant change to the code or create something I think the rest of the team will want right away.

Ally

Test-Driven Development (p. 287)

NOTE

Many teams have a rule that you have to integrate before you go home at the end of the day. If you can't integrate, they say, something has gone wrong and you should throw away your code and start fresh the next day. This rule seems harsh, but it's actually a very good rule. With test-driven development, if you can't integrate within a few minutes, you're likely stuck.

Each integration should get as close to a real release as possible. The goal is to make preparing for a release such an ordinary occurrence that, when you actually do ship, it's a nonevent.* Some teams that use continuous integration automatically burn an installation CD every time they integrate. Others create a disk image or, for network-deployed products, automatically deploy to staging servers.

Toss out your recent changes and start over when you get badly stuck.

Never Break the Build

When was the last time you spent hours chasing down a bug in your code, only to find that it was a problem with your computer's configuration or in somebody else's code? Conversely, when was the last time you spent hours blaming your computer's configuration (or somebody else's code), only to find that the problem was in code you just wrote?

On typical projects, when we integrate, we don't have confidence in the quality of our code *or* in the quality of the code in the repository. The scope of possible errors is wide; if anything goes wrong, we're not sure where to look.

Reducing the scope of possible errors is the key to developing quickly. If you have total confidence that your software worked five minutes ago, then only the actions you've taken in the last five minutes could cause it to fail now. That reduces the scope of the problem so much that you can often figure it out just by looking at the error message—there's no debugging necessary.

* ... except for the release party, of course.

To achieve this, agree as a team never to break the build. This is easier than it sounds: you can actually guarantee that the build will never break (well, almost never) by following a little script.

> Agree as a team never to break the build.

The Continuous Integration Script

To guarantee an always-working build, you have to solve two problems. First, you need to make sure that what works on *your* computer will work on *anybody's* computer. (How often have you heard the phrase, "But it worked on my machine!"?) Second, you need to make sure nobody gets code that hasn't been proven to build successfully.

To do this, you need a spare development machine to act as a central integration machine. You also need some sort of physical object to act as an integration token. (I use a rubber chicken. Stuffed toys work well, too.)

With an integration machine and integration token, you can ensure a working build in several simple steps.

To update from the repository

1. Check that the integration token is available. If it isn't, another pair is checking in unproven code and you need to wait until they finish.
2. Get the latest changes from the repository. Others can get changes at the same time, but don't let anybody take the integration token until you finish.

Run a full build to make sure everything compiles and passes tests after you get the code. If it doesn't, something went wrong. The most common problem is a configuration issue on your machine. Try running a build on the integration machine. If it works, debug the problem on your machine. If it doesn't work, find the previous integrators and beat them about the head and shoulders, if only figuratively.

To integrate

1. Update from the repository (follow the previous script). Resolve any integration conflicts and run the build (including tests) to prove that the update worked.
2. Get the integration token and check in your code.
3. Go over to the integration machine, get the changes, and run the build (including tests).
4. Replace the integration token.

If the build fails on the integration machine, you have to fix the problem before you give up the integration token. The fastest way to do so is to roll back your changes. However, if nobody is waiting for the token, you can just fix the problem on your machine and check in again.

Avoid fixing problems manually on the integration machine. If the build worked on your machine, you probably forgot to add a file or a new configuration to the build script. In either case, if you correct the problem manually, the next people to get the code won't be able to build.

CONTINUOUS INTEGRATION SERVERS

There's a lively community of open-source *continuous integration servers* (also called *CI servers*). The granddaddy of them all is CruiseControl, pioneered by ThoughtWorks employees.

A continuous integration server starts the build automatically after check-in. If the build fails, it notifies the team. Some people try to use a continuous integration server instead of the continuous integration script discussed earlier. This doesn't quite work because without an integration token, team members can accidentally check out code that hasn't yet been proven to work.

Another common mistake is using a continuous integration server to shame team members into improving their build practices. Although the "wow factor" of a CI server can sometimes inspire people to do so, it only works if people are really willing to make an effort to check in good code. I've heard many reports of people who tried to use a CI server to enforce compliance, only to end up fixing all the build failures themselves while the rest of the team ignored their strong-arming.

If your team sits together and has a fast build, you don't need the added complexity of a CI server. Simply walk over to the integration machine and start the build when you check in. It only takes a few seconds—less time than it takes for a CI server to notice your check-in—and gives you an excuse to stretch your legs.

If you do install a CI server, don't let it distract you. Focus on mastering the *practice* of continuous integration, not the *tool*. Integrate frequently, never break the build, and keep your release infrastructure up-to-date.

Introducing Continuous Integration

The most important part of adopting continuous integration is getting people to agree to integrate frequently (every few hours) and never to break the build. Agreement is the key to adopting continuous integration because there's no way to force people not to break the build.

> Get the team to agree to continuous integration rather than imposing it on them.

If you're starting with XP on a brand-new project, continuous integration is easy to do. In the first iteration, install a version control system. Introduce a 10-minute build with the first story, and grow your release infrastructure along with the rest of your application. If you are disciplined about continuing these good habits, you'll have no trouble using continuous integration throughout your project.

If you're introducing XP to an existing project, your tests and build may not yet be good enough for continuous integration. Start by automating your build (see "Ten-Minute Build"" earlier in this chapter), then add tests. Slowly improve your release infrastructure until you can deploy at any time.

Dealing with Slow Builds

The most common problem facing teams practicing continuous integration is slow builds. Whenever possible, keep your build under 10 minutes. On new projects, you should be able to keep your build under 10 minutes all the time. On a legacy project, you may not achieve that goal right away. You can still practice continuous integration, but it comes at a cost.

Ally

Ten-Minute Build (p. 177)

When you use the integration script discussed earlier, you're using *synchronous integration*—you're confirming that the build and tests succeed before moving on to your next task. If the build is too slow, synchronous integration becomes untenable. (For me, 20 or 30 minutes is too slow.) In this case, you can use *asynchronous integration* instead. Rather than waiting for the build to complete, start your next task immediately after starting the build, without waiting for the build and tests to succeed.

The biggest problem with asynchronous integration is that it tends to result in broken builds. If you check in code that doesn't work, you have to interrupt what you're doing when the build breaks half an hour or an hour later. If anyone else checked out that code in the meantime, their build won't work either. If the pair that broke the build has gone home or to lunch, someone else has to clean up the mess. In practice, the desire to keep working on the task at hand often overrides the need to fix the build.

If you have a very slow build, asynchronous integration may be your only option. If you must use this, a continuous integration server is the best way to do so. It will keep track of what to build and will automatically notify you when the build has finished.

Switch to synchronous integration when you can.

Over time, continue to improve your build script and tests (see "Ten-Minute Build"" earlier in this chapter). Once the build time gets down to a reasonable number (15 or 20 minutes), switch to synchronous integration. Continue improving the speed of the build and tests until synchronous integration feels like a pleasant break rather than a waste of time.

Multistage Integration Builds

Some teams have sophisticated tests, measuring such qualities as performance, load, or stability, that simply cannot finish in under 10 minutes. For these teams, multistage integration is a good idea.

A *multistage integration* consists of two separate builds. The normal 10-minute build, or *commit build*, contains all the normal items necessary to prove that the software works: unit tests, integration tests, and a handful of end-to-end tests (see "Test-Driven Development"" in Chapter 9 for more about these types of tests). This build runs synchronously as usual.

In addition to the regular build, a slower *secondary build* runs asynchronously. This build contains the additional tests that do not run in a normal build: performance tests, load tests, and stability tests.

Although a multistage build is a good idea for a mature project with sophisticated testing, most teams I encounter use multistage integration as a workaround for a slow test suite. I prefer to improve the test suite instead; it's more valuable to get better feedback more often.

> Prefer improved tests to a multistage integration.

If this is the case for you, a multistage integration might help you transition from asynchronous to synchronous integration. However, although a multistage build is better than completely asynchronous integration, don't let it stop you from continuing to improve your tests. Switch to fully synchronous integration when you can; only synchronous integration guarantees a known-good build.

Questions

I know we're supposed to integrate at least every four hours, but what if our current story or task takes longer than that?

You can integrate at any time, even when the task or story you're working on is only partially done. The only requirement is that the code builds and passes its tests.

What should we do while we're waiting for the integration build to complete?

Take a break. Get a cup of tea. Perform ergonomic stretches. Talk with your partner about design, refactoring opportunities, or next steps. If your build is under 10 minutes, you should have time to clear your head and consider the big picture without feeling like you're wasting time.

Ally
Ten-Minute Build (p. 177)

Isn't asynchronous integration more efficient than synchronous integration?

Although asynchronous integration may seem like a more efficient use of time, in practice it tends to disrupt flow and leads to broken builds. If the build fails, you have to interrupt your new task to roll back and fix the old one. This means you must leave your new task half-done, switch contexts (and sometimes partners) to fix the problem, then switch back. It's wasteful and annoying.

> Synchronous integration reduces integration problems.

Instead of switching gears in the middle of a task, many teams let the build remain broken for a few hours while they finish the new task. If other people integrate during this time, the existing failures hide any new failures in their integration. Problems compound and cause a vicious cycle: painful integrations lead to longer broken builds, which lead to more integration problems, which lead to more painful integrations. I've seen teams that practice asynchronous integration leave the build broken for days at a time.

Remember, too, that the build should run in under 10 minutes. Given a fast build, the supposed inefficiency of synchronous integration is trivial, especially as you can use that time to reflect on your work and talk about the big picture.

Are you saying that asynchronous integration will never work?

You can make asynchronous integration work if you're disciplined about keeping the build running fast, checking in frequently, running the build locally before checking in, and fixing

problems as soon as they're discovered. In other words, do all the good things you're supposed to do with continuous integration.

Synchronous integration makes you confront these issues head on, which is why it's so valuable. Asynchronous integration, unfortunately, makes it all too easy to ignore slow and broken builds. You don't have to ignore them, of course, but my experience is that teams using asynchronous integration have slow and broken builds much more often than teams using synchronous integration.

Ron Jeffries said it best:*

> When I visit clients with asynchronous builds, I see these things happening, I think it's fair to say invariably:
>
> 1. The "overnight" build breaks at least once when I'm there;
> 2. The build lamp goes red at least once when I'm there, and stays that way for more than an hour.
>
> With a synchronous build, once in a while you hear one pair say "Oh, shjt."
>
> I'm all for more automation. But I think an asynch build is like shutting your eyes right when you drive through the intersection.

Our version control system doesn't allow us to roll back quickly. What should we do?

The overriding rule of the known-good build is that you must *know the build works* when you put the integration token back. Usually, that means checking in, running the build on the integration machine, and seeing it pass. Sometimes—we hope not often—it means rolling back your check-in, running the old build, and seeing that pass instead.

Ally
Version Control (p. 169)

If your version control system cannot support this, consider getting one that does. Not being able to revert easily to a known-good point in history is a big danger sign. You need to be able to revert a broken build with as much speed and as little pain as possible so you can get out of the way of other people waiting to integrate. If your version control can't do this for you, create an automated script that will.

One way to script this is to check out the older version to a temporary sandbox. Delete all the files in the regular sandbox except for the version control system's metadata files, then copy all the nonmetadata files over from the older version. This will allow you to check in the old version on top of the new one.

We rolled back our check-in, but the build is still failing on the integration machine. What do we do now?

Oops—you've almost certainly exposed some sort of configuration bug. It's possible the bug was in your just-integrated build script, but it's equally possible there was a latent bug in one of the previous scripts and you accidently exposed it. (Lucky you.)

Either way, the build has to work before you give up the integration token. Now you debug the problem. Enlist the help of the rest of the team if you need to; a broken integration machine is a problem that affects everybody.

Why do we need an integration machine? Can't we just integrate locally and check in?

* Via the art of agile mailing list, *http://tech.groups.yahoo.com/group/art-of-agile/message/365*.

In theory, if the build works on your local machine, it should work on any machine. In practice, don't count on it. The integration machine is a nice, pristine environment that helps prove the build will work anywhere. For example, I occasionally forget to check in a file; watching the build fail on the integration machine when it passed on mine makes my mistake obvious.

Nothing's perfect, but building on the integration machine does eliminate the majority of cross-machine build problems.

I seem to always run into problems when I integrate. What am I doing wrong?

One cause of integration problems is infrequent integration. The less often you integrate, the more changes you have to merge. Try integrating more often.

Another possibility is that your code tends to overlap with someone else's. Try talking more about what you're working on and coordinating more closely with the pairs that are working on related code.

If you're getting a lot of failures on the integration machine, you probably need to do more local builds before checking in. Run a full build (with tests) before you integrate to make sure your code is OK, then another full build (with tests) afterward to make sure the integrated code is OK. If that build succeeds, you shouldn't have any problems on the integration machine.

I'm constantly fixing the build when other people break it. How can I get them to take continuous integration seriously?

It's possible that your teammates haven't all bought into the idea of continuous integration. I often see teams in which only one or two people have any interest in continuous integration. Sometimes they try to force continuous integration on their teammates, usually by installing a continuous integration server without their consent. It's no surprise that the team reacts to this sort of behavior by ignoring broken builds. In fact, it may actually *decrease* their motivation to keep the build running clean.

Talk to the team about continuous integration before trying to adopt it. Discuss the trade-offs as a group, collaboratively, and make a group decision about whether to apply it.

If your team has agreed to use continuous integration but is constantly breaking the build anyway, perhaps you're using asynchronous integration. Try switching to synchronous integration, and follow the integration script exactly.

Results

When you integrate continuously, releases are a painless event. Your team experiences fewer integration conflicts and confusing integration bugs. The on-site customers see progress in the form of working code as the iteration progesses.

Contraindications

Don't try to force continuous integration on a group that hasn't agreed to it. This practice takes everyone's willful cooperation.

Using continuous integration without a version control system and a 10-minute build is *painful*.

Allies

Version Control (p. 169)
Ten-Minute Build (p. 177)

Synchronous integration becomes frustrating if the build is longer than 10 minutes and too wasteful if the build is very slow. My threshhold is 20 minutes. The best solution is to speed up the build.

A physical integration token only works if all the developers sit together. You can use a continuous integration server or an electronic integration token instead, but be careful to find one that's as easy to use and as obvious as a physical token.

Integration tokens don't work at all for very large teams; people spend too much time waiting to integrate. Use private branches in your version control system instead. Check your code into a private branch, build the branch on an integration machine—you can have several—then promote the branch to the mainline if the build succeeds.

Alternatives

If you can't perform synchronous continuous integration, try using a CI server and asynchronous integration. This will likely lead to more problems than synchronous integration, but it's the best of the alternatives.

If you don't have an automated build, you won't be able to practice asynchronous integration. Delaying integration is a very high-risk activity. Instead, create an automated build as soon as possible, and start practicing one of the forms of continuous integration.

Some teams perform a daily build and smoke test. Continuous integration is a more advanced version of the same practice; if you have a daily build and smoke test, you can migrate to continuous integration. Start with asynchronous integration and steadily improve your build and tests until you can use synchronous integration.

Collective Code Ownership

Audience

Programmers

We are all responsible for high-quality code.

There's a metric for the risk imposed by concentrating knowledge in just a few people's heads—it's called the *truck number*. How many people can get hit by a truck before the project suffers irreparable harm?

It's a grim thought, but it addresses a real risk. What happens when a critical person goes on holiday, stays home with a sick child, takes a new job, or suddenly retires? How much time will you spend training a replacement?

Collective code ownership spreads responsibility for maintaining the code to all the programmers. Collective code ownership is exactly what it sounds like: everyone shares reponsibility for the quality of the code. No single person claims ownership over any part of the system, and anyone can make any necessary changes anywhere.

In fact, improved code quality may be the most important part of collective code ownership. Collective ownership allows—no, *expects*—everyone to fix problems they find. If you encounter

> Fix problems no matter where you find them.

duplication, unclear names, or even poorly designed code, it doesn't matter who wrote it. It's *your* code. Fix it!

Making Collective Ownership Work

Collective code ownership requires letting go of a little bit of ego. Rather than taking pride in *your* code, take pride in *your team's* code. Rather than complaining when someone edits your code, enjoy how the code improves when you're not working on it. Rather than pushing your personal design vision, discuss design possibilities with the other programmers and agree on a shared solution.

Collective ownership requires a joint commitment from team members to produce good code. When you see a problem, fix it. When writing new code, don't do a half-hearted job and assume somebody else will fix your mistakes. Write the best code you can.

> Always leave the code a little better than you found it.

On the other hand, collective ownership means you don't have to be perfect. If you've produced code that works, is of reasonable quality, and you're not sure how to make it better, don't hesitate to let it go. Someone else will improve it later, if and when it needs it.

Working with Unfamiliar Code

If you're working on a project that has *knowledge silos*—in other words, little pockets of code that only one or two people understand—then collective code ownership might seem daunting. How can you take ownership of code that you don't understand?

To begin, take advantage of pair programming. When somebody picks a task involving code you don't understand, volunteer to pair with him. When you work on a task, ask the local expert to pair with you. Similarly, if you need to work on some unfamiliar code, take advantage of your shared workspace to ask a question or two.

Allies

Pair Programming (p. 74)
Sit Together (p. 113)

Rely on your inference skills as well. You don't need to know exactly what's happening in every line of code. In a well-designed system, all you need to know is what each package (or namespace) is responsible for. Then you can infer high-level class responsibilities and method behaviors from their names. (See "Refactoring"" in Chapter 9.)

NOTE

Inferring high-level design from a brief review of the code is a black-belt design skill that's well worth learning. Take advantage of every opportunity to practice. Check your inferences by asking an expert to summarize class responsibilities and relationships.

Rely on the unit tests for further documentation and as your safety net. If you're not sure how something works, change it anyway and see what the tests say. An effective test suite will tell you when your assumptions are wrong.

As you work, look for opportunities to refactor the code. I often find that refactoring code helps me understand it. It benefits the next person, too; well-factored code tends toward simplicity, clarity, and appropriate levels of abstraction.

Ally
Refactoring (p. 306)

If you're just getting started with XP, you might not yet have a great set of unit tests and the design might be a little flaky. In this case, you may not be able to infer the design, rely on unit tests, or refactor, so pairing with somebody who knows the code well becomes more important. Be sure to spend time introducing unit tests and refactoring so that the next person can take ownership of the code without extra help.

Hidden Benefits

> "Of course nobody can understand it... it's job security!"
>
> —Old programmer joke

It's not easy to let a great piece of code out of your hands. It can be difficult to subsume the desire to take credit for a particularly clever or elegant solution, but it's necessary so your team can take advantage of all the benefits of collaboration.

It's also good for you as a programmer. Why? The whole codebase is yours—not just to modify, but to support and improve. You get to expand your skills. Even if you're an absolute database guru, you don't have to write only database code throughout the project. If writing a little UI code sounds interesting, find a programming partner and have at it.

You also don't have to carry the maintenance burden for a piece of code someone assigned you to write. Generally, the pair that finds a bug fixes the bug. They don't need your permission. Even better, they don't necessarily need your help; they may know the code now as well as you did when you wrote it.

It's a little scary at first to come into work and not know exactly what you'll work on, but it's also freeing. You no longer have long subprojects lingering overnight or over the weekend. You get variety and challenge and change. Try it—you'll like it.

Questions

We have a really good UI designer/database programmer/scalability guru. Why not take advantage of those skills and specialties?

Please do! Collective code ownership shares knowledge and improves skills, but it won't make everyone an expert at everything.

Don't let specialization prevent you from learning other things, though. If your specialty is databases and the team is working on user interfaces this week, take on a user interface task. It can only improve your skills.

How can everyone learn the entire codebase?

People naturally gravitate to one part of the system or another. They become experts in particular areas. Everybody gains a general understanding of the overall codebase, but each person only knows the details of what he's worked with recently.

The tests and simple design allow this approach to work. Simple design and its focus on code clarity make it easier to understand unfamiliar code. The tests act both as a safety net and as documentation.

Doesn't collective ownership increase the possibility of merge conflicts?

It does, and so it also requires continuous integration. Continuous integration decreases the chances of merge conflicts.

Ally

Continuous Integration (p. 183)

In the first week or two of the project, when there isn't much code, conflicts are more likely. Treat the code gently for the first couple of iterations. Talk together frequently and discuss your plans. As you progress, the codebase will grow, so there will be more room to make changes without conflict.

We have some pretty junior programmers, and I don't trust them with my code. What should we do?

Rather than turning your junior programmers loose on the code, make sure they pair with experienced members of the team. Keep an eye on their work and talk through design decisions and trade-offs. How else will they learn your business domain, learn your codebase, or mature as developers?

Different programmers on our team are responsible for different projects. Should the team collectively own all these projects?

If you have combined programmers working on several projects into a single team (as described in the discussion of team size in "Is XP Right for Us?"" in Chapter 4), then yes, the whole team should take responsibility for all code. If your programmers have formed multiple separate teams, then they usually should not share ownership across teams.

Results

When you practice collective code ownership, you constantly make minor improvements to all parts of the codebase, and you find that the code you've written improves without your help. When a team member leaves or takes a vacation, the rest of the team continues to be productive.

Contraindications

Don't use collective code ownership as an excuse for *no* code ownership. Managers have a saying: "Shared responsibility is no responsibility at all." Don't let that happen to your code. Collective code ownership doesn't mean someone else is responsible for the code; it means *you* are responsible for the code—all of it. (Fortunately, the rest of the team is there to help you.)

Collective code ownership requires good communication. Without it, the team cannot maintain a shared vision, and code quality will suffer. Several XP practices help provide this communication: a team that includes experienced designers, sitting together, and pair programming.

Allies

Sit Together (p. 113)
Pair Programming (p. 74)

Although they are not strictly necessary, good design and tests make collective code ownership easier. Proceed with caution unless you use test-driven development, simple design, and agree on coding standards. To take advantage of collective ownership's ability to improve code quality, the team must practice relentless refactoring.

Allies

Test-Driven Development (p. 287)
Simple Design (p. 316)
Coding Standards (p. 133)
Refactoring (p. 306)

To coordinate changes, you must use continuous integration and a concurrent model of version control.

Allies

Continuous Integration (p. 183)
Version Control (p. 169)

Alternatives

A typical alternative to collective code ownership is *strong code ownership*, in which each module has a specific owner and only that person may make changes. A variant is *weak code ownership*, in which one person owns a module but others can make changes as long as they coordinate with the owner. Neither approach, however, shares knowledge or enables refactoring as well as collective ownership does.

If you cannot use collective code ownership, you need to adopt other techniques to spread knowledge and encourage refactoring. Pair programming may be your best choice. Consider holding weekly design workshops to review the overall design and to brainstorm improvements.

Ally

Pair Programming (p. 74)

I recommend against strong code ownership. It encourages rigid silos of knowledge, which makes you vulnerable to any team member's absence. Weak code ownership is a better choice, although it still doesn't provide the benefits of collective ownership.

Documentation

Audience

Whole Team

We communicate necessary information effectively.

The word *documentation* is full of meaning. It can mean written instructions for end-users, or detailed specifications, or an explanation of APIs and their use. Still, these are all forms of *communication*—that's the commonality.

Communication happens all the time in a project. Sometimes it helps you get your work done; you ask a specific question, get a specific answer, and use that to solve a specific problem. This is the purpose of *work-in-progress documentation*, such as requirements documents and design documents.

Other communication provides business value, as with *product documentation*, such as user manuals and API documentation. A third type—*handoff documentation*—supports the long-term viability of the project by ensuring that important information is communicated to future workers.

Work-In-Progress Documentation

In XP, the whole team sits together to promote the first type of communication. Close contact with domain experts and the use of ubiquitous language create a powerful oral tradition that transmits information when necessary. There's no substitute for face-to-face communication. Even a phone call loses important nuances in conversation.

XP teams also use test-driven development to create a comprehensive test suite. When done well, this captures and communicates details about implementation decisions as unambiguous, executable design specifications that are readable, runnable, and modifiable by other developers. Similarly, the team uses customer testing to communicate information about hard-to-understand domain details. A ubiquitous language helps further reveal the intent and purpose of the code.

The team does document some things, such as the vision statement and story cards, but these act more as reminders than as formal documentation. At any time, the team can and should jot down notes that help them do their work, such as design sketches on a whiteboard, details on a story card, or hard-to-remember requirements in a wiki or spreadsheet.

In other words, XP teams don't need traditional written documentation to do their work. The XP practices support work-in-progress communication in other ways—ways that are actually more effective than written documentation.

Product Documentation

Some projects need to produce specific kinds of documentation to provide business value. Examples include user manuals, comprehensive API reference documentation, and reports. One team I worked with created code coverage metrics—not because they needed them, but because senior management wanted the report to see if XP would increase the amount of unit testing.

Because this documentation carries measurable business value but isn't otherwise necessary for the team to do its work, schedule it in the same way as all customer-valued work: with a story. Create, estimate, and prioritize stories for product documentation just as you would any other story.

Allies

Stories (p. 255)
The Planning Game (p. 221)

Handoff Documentation

If you're setting the code aside or preparing to hand off the project to another team (perhaps as part of final delivery), create a small set of documents recording big decisions and information. Your goal is to summarize the most important information you've learned while creating the software—the kind of information necessary to sustain and maintain the project.

Besides an overview of the project and how it evolved in design and features, your summary should include nonobvious information. Error conditions are important. What can go wrong, when might it occur, and what are the possible remedies? Are there any traps or sections of the code where the most straightforward approach was inappropriate? Do certain situations reoccur and need special treatment?

This is all information you've discovered through development as you've learned from writing the code. In clear written form, this information helps mitigate the risk of handing the code to a fresh group.

As an alternative to handoff documentation, you can gradually migrate ownership from one team to another. Exploit pair programming and collective code ownership to move new developers and other personnel onto the project and to move the previous set off in phases. Instead of a sharp break (or big thud) as one team's involvement ends and the other begins, the same osmotic communication that helps a team grow can help transition, repopulate, or shrink a team.

Allies

Pair Programming (p. 74)
Collective Code Ownership (p. 191)

Questions

Isn't it a risk to reduce the amount of documentation?

It could be. In order to reduce documentation, you have to replace it with some other form of communication. That's what XP does.

Increasing the amount of written communication also increases your risk. What if that information goes out of date? How much time does someone need to spend updating that documentation, and could that person spend that time updating the tests or refactoring the code to communicate that information more clearly?

The real risk is in decreasing the amount and accuracy of appropriate communication for your project, not in favoring one medium of communication. Favoring written communication may decrease your agility, but favoring spoken communication may require more work to disseminate information to the people who need it.

Results

When you communicate in the appropriate ways, you spread necessary information effectively. You reduce the amount of overhead in communication. You mitigate risk by presenting only necessary information.

Contraindications

Alistair Cockburn describes a variant of Extreme Programming called "Pretty Adventuresome Programming":*

> A PrettyAdventuresomeProgrammer says:
>
> "Wow! That ExtremeProgramming stuff is neat! We almost do it, too! Let's see...
>
> "Extreme Programming requires:
>
> - You do pair programming.
> - You deliver an increment every three† weeks.
> - You have a user on the team full time.
> - You have regression unit tests that pass 100% of the time.

* *http://c2.com/cgi/wiki?PrettyAdventuresomeProgramming*.

† Most teams now use one- or two-week iterations. I recommend one-week iterations for new teams; see "Iteration Planning"" in Chapter 8.

- You have automated acceptance tests which define the behavior of the system.

"As a reward for doing those,

- You don't put comments in the code.
- You don't write any requirements or design documentation.

"Now on this project, we're pretty close...

- well, actually a couple of our guys sit in the basement, a couple on the 5th floor, and a couple 2 hours drive from here, so we don't do pair programming,
- and actually, we deliver our increments every 4-6 months,
- we don't have users anywhere in sight,
- and we don't have any unit tests,

"but at least *we don't have any design documentation,*‡ and we don't comment our code much! So in a sense, you're right, we're almost doing ExtremeProgramming!"

Those people aren't doing XP, they are doing PAP [Pretty Adventuresome Programming]. PAP is using XP (in name) to legitimize not doing the things one doesn't want to do, without doing the XP practices that protects one from not doing the other things. E.g., changing the code all the time, but not writing unit tests; not writing documentation, but not writing clear code either. Not...(almost anything)... but not sitting close together. etc.*

In other words, continue to create documentation until you have practices in place to take its place. You have to be rigorous in your practice of XP in order to stop writing work-in-progress documentation. Particularly important is a whole team (with all the team roles filled—see "The XP Team"" in Chapter 3) that sits together.

Ally

Sit Together (p. 113)

Some organizations value written documentation so highly that you can't eliminate work-in-progress documents. In my experience, these organizations usually aren't interested in trying XP. If yours is like that, but it wants to do XP anyway, talk with management about why those documents are important and whether XP can replace them. Perhaps handoff documents are an acceptable compromise. If not, don't eliminate work-in-progress documents. Either schedule the documents with stories or include the cost of creating and updating documents in your estimates.

Alternatives

If you think of documents as *communication mechanisms* rather than simply printed paper, you'll see that there are a wide variety of alternatives for documentation. Different media have different strengths. Face-to-face conversations are very high bandwidth but can't be referenced

‡ qEmphasis in original.

* Alistair later added: "I am interested in having available a sarcasm-filled, derisively delivered phrase to hit people with who use XP as an excuse for sloppy, slap-dash development. I, of all people, think it actually is possible to turn dials to different numbers [Alistair means that you don't have to go as far as XP does to be successful], but I have no patience with people who slap the XP logo on frankly sloppy development."

later, whereas written documents are very low bandwidth (and easily misunderstood) but can be referred to again and again.

Alistair Cockburn suggests an intriguing alternative to written documents for handoff documentation: rather than creating a design overview, use a video camera to record a whiteboard conversation between an eloquent team member and a programmer who doesn't understand the system. Accompany the video with a table of contents that provides timestamps for each portion of the conversation.