No Bugs

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.

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.

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.

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.

How to Achieve Nearly Zero Bugs

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.

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, writing 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 removing obstacles. If you do them as part of XP, you don'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.

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.

Ingredient #3: Fix Bugs Now

Programmers have long known that the longer you wait to fix a bug, the more it costs to fix .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.

Ingredient #4: Test Your Process

These practices will dramatically cut down on 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.

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.

Ten-Minute Build

We eliminate build and configuration hassles.

Here's an ideal to strive for. Imagine that 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.

You sit down together. "Okay, 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 of our build tools and libraries in version control, too," you say. "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."

"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.

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.

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.

adly, 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.

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.

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.

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 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 on because you can never let the build break in XP. If the build breaks, you have to roll back your changes.

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

The easiest way to keep the build under five minutes (with a ten-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.

Continuous Integration

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 work 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.

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.

Never Break the Build

When was the last time you spent hours chasing down a bug, 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.

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, "It worked on my machine!"?) Second, you need to make sure that 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.

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.

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.

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.

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 ten-minute build with the first story, and grow your release infrastructure along with the rest of your application

Dealing with Slow Builds

The most common problem facing teams practicing continuous integration is slow builds. Whenever possible, keep your build under ten minutes. On new projects, you should be able to keep your build under ten 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.

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.

Multistage Integration Builds

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

A *multistage integration* consists of two separate builds. The normal ten-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

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.

Collective code Ownership:

Collective code ownership, as the name suggests, is the explicit convention that "every" team member is not only allowed, but in fact has a positive duty, to make changes to "any" code file as necessary: either to complete a development task, to repair a defect, or even to improve the code's overall structure.

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. 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 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 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.

Working with Unfamiliar Code:

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.

Hidden Benefits:

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 through out the project. If writing a litte UI code sounds interesting, find a programming partner and have at it.

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 so long as they coordinate with the owner.

Documentation:

We communicate necessary information effectively.

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

We communicate necessary information effectively.

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

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:
whole team sit 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. here's no substitute for face-to-face communication.Even a phone call loses important nuances in conversation.

Product Documentation:

projects need to produce specific kinds of documentation to provide business value. Examples include user manuals, comprehensive API reference documentation, and reports.

Handoff Documentation:

If you're setting the code aside or preparing to hand the project off 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.

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.