

## **RELEASING**

Software that is pushed into production **gains value**.

In order to meet commitments and take advantage of opportunities, push the software into production within minutes.

Six practices that reduce a big release push to a 10-minute process:

- **Done done** - ensures that completed work is ready to release
- **No bugs** – allows to release software without separate testing phase
- **Version control** – allows team member to work together
- **Ten-minute build** – builds a tested release package in under 10 minutes
- **Continuous Integration** – prevents long risky integration phase
- **Collective Code Ownership** – allows team members to solve problems no matter where they may lie
- **Post-hoc documentation** – decreases the cost of documentation

### **Done Done**

Production ready software is said to be **Done Done**.

### **Production Ready Software**

A completed story isn't a lump of unintegrated, untested code. It's ready to deploy. Partially finished stories result in hidden costs to the 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.

Checklist that shows the story completion criteria:

- Designed (code refactored to the team's satisfaction)
- Coded (all code written)
- Builds (the build script includes any new modules)

- Integrated (the story works from end to end—typically, UI to database—and fits into the rest of the software)
- Tested (all unit, integration, and customer tests finished)
- Fixed (all known bugs have been fixed or scheduled as their own stories)
- 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)
- Accepted (customers agree that the story is finished)

Some teams include Documentation, Scalability and Performance also in their “Done Done” list.

### **How to be “Done Done”**

XP works best by making small daily progress on all tasks instead of rushing at the end. Use TDD, continuous integration, and keep builds fast and updated. Integrate code early, update installers in parallel, and involve customers throughout. Run the software often to catch issues early, fix problems immediately, and improve practices. Final customer acceptance should take only 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. Make sure stories are small enough that can be completely finished all in a single week.

If there is trouble getting the stories “done done”, don’t count those stories toward the velocity.

Even if the story only has a few minor UI bugs, count it as a zero when calculating velocity. This will lower velocity, which will help to choose a more manageable amount of work in next iteration.

### **No Bugs**

With no bugs, software can be released without a dedicated testing phase.

### **How is this possible?**

Having “no bugs” may sound unrealistic, especially for teams with many defects. It is an ideal to aim for, not a guaranteed outcome. However, XP teams have shown that bug rates can be reduced significantly, sometimes to just a few per month, without high cost or loss of productivity. While large-scale scientific proof is limited, real project experiences suggest strong benefits. The only way to know if it will work for your team is to try XP seriously, with commitment to its practices and support from the organization even novice teams can succeed with proper guidance.

### **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.

#### **Techniques:**

1. **Write fewer bugs** – Start with TDD to reduce the number of defects that generated. Pair programming improves brainpower, which helps to make fewer mistakes and allows to see mistakes more quickly. To prevent requirement-oriented defects, work closely with stakeholders. Demonstrate software to stakeholders every week and act on their feedback.
2. **Eliminate bug breeding grounds** - Even with test-driven development, software will accumulate technical debt over time. 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 the debt. Keep new code clean by creating simple designs. Use the slack in each iteration to pay down debt in old code.
3. **Fix bugs now** - Programmers know that the longer you wait to fix a bug, the more it costs to fix. To fix the bug, start by writing an automated test that demonstrates the bug. It could be an unit test, integration test, or customer test, depending on what kind of defect you’ve found. Once failed test is found, fix it. Fixing bugs quickly requires the whole team to participate. Programmers, use collective code ownership so that any pair can fix a buggy module. Some bugs are too big to fix while you’re in the middle of another task. 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. 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.

4. **Test your process** - Good practices reduce bugs, but only for problems you expect. **Exploratory testing** helps uncover unexpected issues by using tester intuition and creative scenarios. It should **not replace TDD**—instead, use it to reveal weaknesses in your process. When exploratory testing finds bugs, fix both the bug and the process. Run exploratory tests only on “done-done” stories, treat findings as real bugs, and improve practices to keep defect rates very low.
5. **Fix your process** - Some bugs are human errors, but many reveal flaws in the process. When bug counts are low, teams can perform **root-cause analysis** on each bug. Start by writing a test, fixing the design, and asking *why* the bug occurred using techniques like the **five whys**. As a team, improve work habits to prevent similar issues and add tests to uncover related hidden bugs.

## **Version Control**

A **version control system** provides a central repository that helps coordinate changes to files and also provides a history of changes. 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.

**Terminology** - Different version control systems use different terminology:  
**Repository** - The repository is the master storage for all your files 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. The sandbox contains a copy of all the files in the repository from a particular point in time.

**Checkout** - 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.

**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.

**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 void this, locking model of version control can be used. If one programmer works on a file it is locked for other programmers and the file will be read-only until locked.

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.

Concurrent model of version control system 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. They would be risky if it weren’t for continuous integration and the automated build.

### **Time travel**

Ability to go back in time.

Sandbox can be updated with all files from a particular point in the past. This allows to use diff debugging. **Diff debugging** is a debugging technique where you **compare (diff)** two versions of a program—one that **works correctly** and one that **has a bug**—to identify what change caused the problem. 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**

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. Keep all your tools, libraries, documentation, and everything else related to the project in version control.

Store the entire project in a single repository.

Always perform update and commit actions from the top-level directory to ensure the entire project remains in a single consistent version. Committing only a subdirectory can split your working copy across different versions.

The only project-related artifact that don't need to keep in version control is generated code. Automated build should re-create generated code automatically.

## **Customers and Version Control**

Customer data should go in the repository

- Documentation
- Notes on requirements
- Manuals
- Customer tests

## **Keep It Clean**

Never check in code that breaks the build.

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.

Levels of code 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**

Duplicating a codebase to deliver customized software doubles maintenance effort and severely slows development. Such copy-paste customization creates long-term technical debt that is difficult to undo. Using version control branches for multiple custom versions has similar risks, as keeping branches synchronized becomes increasingly complex. Instead, design the code to support multiple configurations using plug-ins, configuration files, shared libraries, and an automated build and delivery process.

## **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. Branches can also be useful for continuous integration and other code management tasks. These private branches live for less than a day.