RELEASING

Done Done

Audience: Whole Team

“We’re done when we’re production-ready.”

Teams that use “done-done” often use the term to mean “we did as much work as we were prepared to do!”

Production-Ready Software

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

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.

How to Be “Done Done”

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

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.

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.

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.

Alternatives

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.

No Bugs

Audience: Whole Team

“We confidently release without a dedicated testing phase.”

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

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.

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

- Write fewer bugs by using a wide variety of technical and organizational practices.
- Eliminate bug breeding grounds by refactoring poorly designed code.
- 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.
- Test your process by using exploratory testing to expose systemic problems and hidden assumptions.
- Fix your process by uncovering categories of mistakes and making those mistakes impossible.

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.
- All the "Collaborating" practices except "Reporting" are necessary.
- All the "Releasing" practices except "Documentation" are necessary.
- All the "Planning" practices except "Risk Management" are necessary.
- All the "Developing" practices except "Spike Solutions" are necessary.

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.

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. This is also known as source control.

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.

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.

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.

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 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.

Customers and Version Control

Customer data should go in the repository, too. That includes documentation, notes on requirements, technical writing such as manuals, and customer tests

Keep It Clean

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.

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.

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.