31

Small Cycles

4. I will make frequent, small releases so that I do not impede the progress of others.

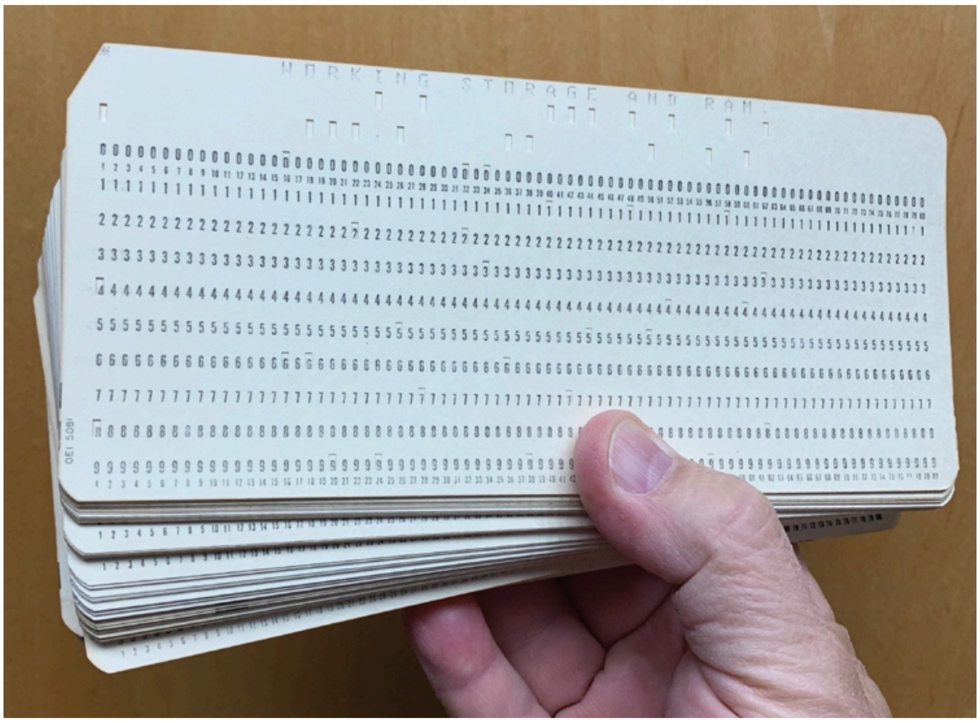
Making small releases just means changing a small amount of code for each release. The system may be large, but the change to that system is small.

The History of Source Code Control

The following is a look back at the history of source code control.

Punch Cards

Let's go back to the 1960s for a moment. What is your source code control system, when your source code is punched on a deck of cards?



The source code is not stored on disk. It's not "in the computer." Your source code is, literally, in your hand.

What is the source code control system? It is your desk drawer.

When you literally possess the source code, there is no need to "control" it. Nobody else can touch it.

And this was the situation throughout much of the '50s and '60s. Nobody even dreamed of something like a source code control system. You simply kept the source code under control by putting it in a drawer or a cabinet.

If anyone wanted to "check out" the source code, they simply went to the cabinet and took it. When they were done, they put it back.

We certainly didn't have merge problems. It was physically impossible for two programmers to be making changes to the same modules at the same time.

But things started to change in the '70s. The idea of storing your source code on magnetic tape, or even on disk, was becoming attractive.

We wrote line editing programs that allowed us to add, replace, and delete lines in source files on tape.

These weren't screen editors. We punched our add, change, and delete directives on cards. The editor would read the source tape, apply the changes from the edit deck, and write the new source tape.

You may think this sounds awful. Looking back on it, it was. But it was better than trying to manage programs on cards! I mean, 6,000 lines of code on cards weighs 30 pounds. If you dropped it…

If you drop a tape, you can just pick it up again.

Anyway, notice what happened. We start with one source tape, and in the editing process, we wind up with a second, new source tape. But the old tape still exists. If you put that old tape back on the rack, someone else might inadvertently apply their own changes to it, leaving you with a merge problem.

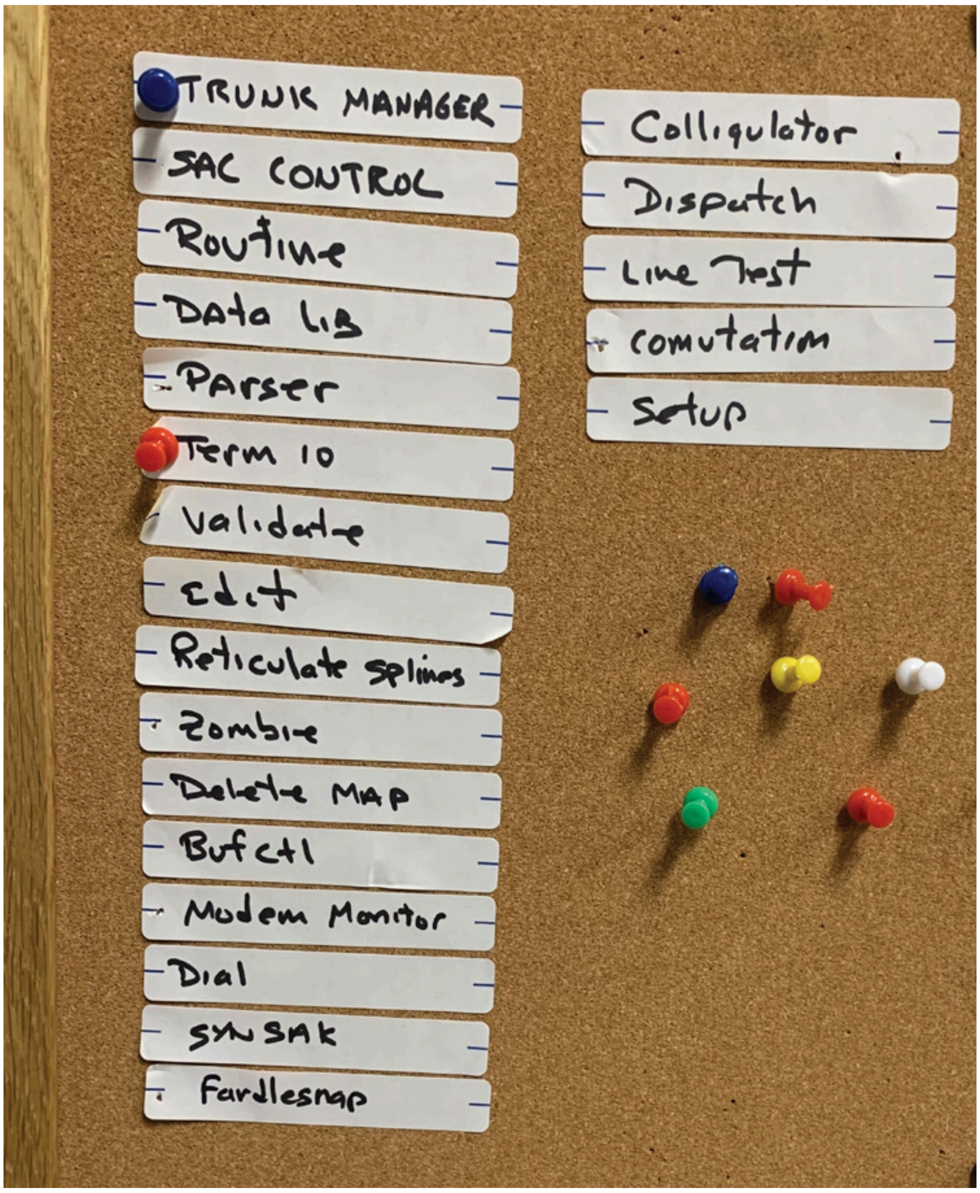
To prevent that, we simply kept the master source tape in our possession until we were done with our edits and our tests. Then, we put a new master source tape back on the rack. We controlled the source code by keeping possession of the tape.

But now this had to be done by process and convention. A true source code control process had to be used. No software yet, just human rules. But still, the concept of source code control had become separate from the source code itself.

As systems got bigger and bigger, they needed more and more programmers working on the same code at the same time. Grabbing the master tape and holding it became a real nuisance for everyone else. I mean, you could keep the master tape out of circulation for a couple of days or more.

So then, we decided to extract modules from the master tape. The whole idea of modular programming was pretty new at the time. The notion that a program could be made up of many different source files was revolutionary.

So we started to use bulletin boards, like this:



The bulletin board had labels on it for each of the modules in the system. Each programmer had their own color thumbtack. I was blue. Ken was red. My buddy CK was yellow, and so on.

If I wanted to edit the “Trunk Manager” module, I’d look on the bulletin board to see if there was a pin in that module. If not, I put a blue pin in it.

Then, I took the master tape from the rack and copied it onto a separate tape.

I would edit the Trunk Manager module, and only the Trunk Manager module, generating a new tape with my changes.

I’d compile, test, wash, and repeat until I got my changes to work.

Then, I’d go get the master tape from the rack, and I would create a new master by copying the current master, but replacing the Trunk Manager module with my changes.

I’d put the new master back in the rack.

Finally, I’d remove my blue pin from the bulletin board.

This worked; but it only worked because we all knew each other; we all worked in the same office together; we all knew what each other was doing. And we *talked* to each other all the time.

I’d holler across the lab: “Ken, I’m going to change the Trunk Manager module.” He’d say, “Put a pin in it.” I’d say, “I already did.”

The pins were just reminders. We all knew the status of the code and who was working on what. And that’s why the system worked.

Indeed, it worked very well. Knowing what each other was doing meant that we could help each other. We could make suggestions. We could warn each other about problems we’d recently encountered. And we could avoid merges.

Back then, merges were *not* fun.

Then, in the ’80s, came the disks. Disks got big, and they got permanent.

By big, I mean hundreds of megabytes. By permanent, I mean that they were permanently mounted—always online.

The other thing that happened is that we got machines like PDP11s and VAXes. We got screen editors, real operating systems, and multiple terminals. More than one person could be editing at exactly the same time.

The era of thumbtacks and bulletin boards had to come to an end.

First of all, by then, we had 20 or 30 programmers. There weren’t enough thumbtack colors. Second of all, we had hundreds and hundreds of modules. We didn’t have enough space on the bulletin board.

Fortunately, there was a solution.

In 1972, Marc Rochkind wrote the first source code control program. It was called SCCS, and he wrote it in SNOBOL.[1]

---

[1]. A lovely little string processing language from the ’60s that had many of the pattern matching facilities of our modern languages.

Later he rewrote it in C, and it became part of the Unix distribution on PDP11s. SCCS only worked on one file at a time, but it allowed you to lock that file so that someone else couldn’t edit it until you were done. It was a lifesaver.

In 1982 RCS, the Revision Control System, was created by Walter Tichy. It too was file based and knew nothing of projects, but it was considered an improvement on SCCS and rapidly became the standard source code control system of the day.

Then, in 1986, CVS, the Concurrent Versions System, came along. It extended RCS to deal with whole projects, not just individual files. It also introduced the concept of optimistic locking.

Up to this time, most source code control systems worked like my thumbtacks. If you had checked out the module, nobody else could edit it. This is called *pessimistic locking*.

CVS used *optimistic locking*. Two programmers could check out and change the same file at the same time. CVS would try to merge any nonconflicting changes, and would alert you if it couldn’t figure out how to do the merge.

After that, source code control systems exploded and even became commercial products. There were literally hundreds of them. Some used optimistic locking, others used pessimistic locking. Indeed, the locking strategy became a kind of religious divide in the industry.

Then, in 2000, Subversion was created. It vastly improved upon CVS and was instrumental in driving the industry away from pessimistic locking once and for all.

Subversion was also the first source code control system to be used in the “cloud.” Does anybody remember Source Forge?

Up to this point, all source code control systems were based on the master tape idea that I used back in my bulletin board days. The source code was maintained in a single central master repository. Source code was checked out from that master repository, and commits were made back into that master repository.

But all that was about to change.

Continuous Integration

The year is 2005. Multigigabyte disks are in our laptops. Network speeds are fast, and getting faster. Processor clock rates have plateaued at 2.6GHz.

We are very, very far away from my old bulletin board control system for source code. But we are still using the master tape concept. We still have a central repository that everybody has to check in and out of. Every commit; every reversion; every merge requires network connectivity to the master.

...and then came git.

Well, actually, git was presaged by BitKeeper and Monotone, but it was git that caught the attention of the programming world and changed everything.

Because, you see, git eliminates the master tape.

Oh, you still need a final authoritative version of the source code. But git does not automatically provide this location for you. Git simply doesn’t care about that. Where you decide to put your authoritative version is entirely up to you. Git has nothing whatsoever to do with it.

Git keeps the entire history of the source code on your local machine. Your laptop, if that’s what you use. On your machine, you can commit changes, create branches, check out old versions. And generally do anything you could do with a centralized system like Subversion; except you don’t need to be connected to some central server.

Anytime you like, you can connect to another user and push any of the changes you've made to that user. Or you can pull any changes they have made into your local repository. Neither is the master. Both are equal. That's why they call it *peer to peer*.

And the final authoritative location that you use for making production releases is just another peer that people can push to or pull from, anytime they like.

The end result of this is that you are free to make as many small commits as you like before you push your changes somewhere else. You can commit every 30 seconds if you like. You could make a commit every time you get a unit test to pass.

And that brings us to the point of this whole historical discussion.

If you stand back and look at the trajectory of the evolution of source code control systems, you can see that they have been driven, perhaps unconsciously, by a single underlying imperative.

Short Cycles

Consider, again, how we began. How long was the cycle when source code was controlled by physically possessing a deck of cards?

You checked the source code out by taking those cards out of the cabinet. You held on to those cards until you were done with your project. Then, you committed your changes by putting the changed deck of cards back in the cabinet.

The cycle time was the whole project.

Then, when we used thumbtacks on the bulletin board, the same rule applied. You kept your thumbtacks in the modules you were changing until you were done with the project you were working on.

Even in the late '70s and into the '80s, when we were using SCCS and RCS, we continued to use this pessimistic locking strategy, keeping others from touching the modules we were changing until we were done.

But CVS changed things—at least for some of us. Optimistic locking meant that one programmer could not lock others out of a module. We still only committed when we were done with a project; but others could be working concurrently on the same modules. So the average time between commits on the project shrank drastically. The cost of this, of course, was merges.

And how we hated doing merges.

Merges are awful. Especially without unit tests! They are tedious, time-consuming, and dangerous.

Indeed, our distaste for merges drove us to a new strategy.

Continuous Integration

By 2000, even while we were using tools like Subversion, we had begun teaching the discipline of committing every few *minutes*.

The rationale was simple. The more frequently you commit, the less likely you are to face a merge. And if you do have to merge, the merge will be trivial.

We called this discipline *continuous integration*.

Of course, this depends critically upon having a very reliable suite of unit tests. I mean, without good unit tests, it would be easy to make a merge error and break someone else's code. So continuous integration goes hand in hand with a good testing discipline.

And now, with tools like git, there is almost no limit to how small we can shrink the cycle.

And that begs the question: Why are we so concerned about shrinking the cycle?

Because long cycles impede the progress of the team.

The longer the time between commits, the greater the chance that someone else on the team—perhaps the whole team—will have to wait for you. And that violates the promise.

Perhaps you think this is only about production releases. No, it’s actually about every other cycle. It’s about iterations and sprints. It’s about the edit/compile/test cycle. It’s about the time between commits. It’s about everything.

And remember the rationale: “So that you do not impede the progress of others.”

Branches versus Toggles

I used to be a branch dogmatist. Back when I was using CVS and Subversion, I refused to allow members on my teams to branch the code. I wanted all changes returned to the main line as frequently as possible.

My rationale was simple. A branch is simply a long-term checkout. And as we’ve seen, long-term checkouts impede the progress of others by prolonging the time between integrations.

But then I switched to git—and everything changed overnight.

At the time, I was managing the open source FitNesse project, which had a dozen or so people working on it. I had just moved the FitNesse repository from Subversion (Source Forge) to git (GitHub). Suddenly, branches started appearing all over the place.

For the first few days, these crazy branches in git had me confused. Should I abandon my branch dogmatism stance? Should I abandon continuous integration and simply allow everyone to make branches willy-nilly, forgetting about the cycle time issue?

But then, it occurred to me that these “branches” I was seeing were not true named branches. Instead, they were just the stream of commits made by a developer between pushes.

In fact, all git had really done was to record the actions of the developer between continuous integration cycles.

So I resolved to continue my rule restricting branches. It’s just that now it’s not commits that return immediately to the main line. It’s pushes. Continuous integration was preserved.

If we are following continuous integration and pushing to the main line every hour or so, then we'll certainly have a bunch of half-written features on the main line. There are typically two strategies for dealing with that: branches and toggles.

The branching strategy is simple. You create a new branch of the source code in which to develop the feature. You merge it back when the feature is done.

If you keep the branch out of the main line for days or weeks, then you are likely facing a big merge; and you'll certainly be impeding the team.

However, there are cases where the new feature is so isolated from the rest of the code that branching it is not likely to cause a big merge. In those cases, it might be better to let the developers work in peace on the new feature without continuously integrating.

In fact, we had a case like this with FitNesse a few years ago. We completely rewrote the parser. This was a big project. It took a few person-weeks. And there was no way to do it incrementally. I mean, the parser is the parser.

So we created a branch and kept that branch isolated from the rest of the system until the parser was ready.

There was a merge to do at the end, but it wasn't too bad. The parser was isolated well enough from the rest of the system. And fortunately, we had a very comprehensive set of unit and acceptance tests.

Most of the time, however, I think it's better to keep new feature development on the main line, and use toggles to turn those features off until ready.

Sometimes we use flags for those toggles, but more often we use the Command pattern, the Decorator pattern, and special versions of the Factory pattern to make sure the partially written features cannot be executed in a production environment.

And most of the time, we simply don't give the user the option to use the new feature. I mean, if the button isn't on the Web page, you can't execute that feature….

In many cases, of course, new features will be completed as part of the current iteration—or at least before the next production release. So there's no real need for any kind of toggle.

You only need a toggle if you are going to be releasing to production while some of the features are unfinished. So how often should that be?

Continuous Deployment

What if we could eliminate the delays between production releases? What if we could release to production several times per day? After all, delaying production releases impedes others.

I want you to be able to release your code to production several times per day. I want you to feel comfortable enough with your work that you could release your code to production on every push.

This, of course, depends upon automated testing. Automated tests are written by programmers to cover every line of code. Automated tests are written by business analysts and Q/A testers to cover every desired behavior.

Remember the preceding chapter? Those tests are the scientific proof that everything works as it is supposed to. And if everything works as it is supposed to, the next step is to deploy to production.

And by the way, that's how you know if your tests are good enough. Your tests are good enough if, when they pass, you feel comfortable deploying. If passing tests don't allow you to deploy, your tests are deficient.

Perhaps you think that deploying every day, or even several times per day, would lead to chaos. However, the fact that *you* are ready to deploy does not mean that the business is ready to deploy. As a development team, your standard is to always be ready.

What's more, we want to help the business remove all the impediments to deployment so that the business deployment cycle can be shortened as much as possible.

After all, the more ceremony and ritual the business uses for deployment, the more expensive deployment becomes. Any business would like to shed that expense.

The ultimate goal for any business is continuous, safe, and ceremony-free deployment. Deployment should be as close to a nonevent as possible.

And since deployment is often a lot of work, with servers to configure and databases to load, you need to automate the deployment procedure.

And since deployment scripts are part of the system, you write tests for them.

For many of you, the idea of continuous deployment may be so far away from your current process that it's inconceivable. But that doesn't mean there aren't things you can do to shorten the cycle.

And who knows? If you keep on shortening the cycle, month after month, year after year, perhaps one day you'll find that you are deploying continuously.

Continuous Build

Clearly, if you are going to deploy in short cycles, you have to be able to build in short cycles. If you are going to deploy continuously, you have to be able to build continuously.

Perhaps some of you have slow build times. If you do, speed them up.

Seriously, with the memory and speed of modern systems, there is no excuse for slow builds. None. Speed them up. Consider it a design challenge.

And then, get yourself a continuous build tool and use it. Make sure that you kick off a build at every push. And do what it takes to ensure that the build never fails.

A build failure is a red alert. It's an emergency. If the build fails, I want emails and text messages sent to every team member. I want sirens going off. I want a flashing red light on the CEO's desk. I want everybody to stop whatever they are doing and deal with the emergency.

Keeping the build from failing is not rocket science. You simply run the build in your local environment, along with all the tests, *before you push*. You only push the code when all the tests pass.

If the build fails after that, you've uncovered some environmental issue that needs to be resolved posthaste.

You never allow the build to go on failing, because if you allow the build to fail, you'll get used to it failing. And if you get used to the failures, you'll start ignoring them.

The more you ignore those failures, the more annoying the failure alerts become. And the more tempted you are to turn the failing tests off until you can fix them—later. You know. Later?

And that's when the tests become lies.

With the failing tests removed, the build passes again. And that makes everyone feel good again. But it's a lie.

So build continuously. And never let the build fail.