Agile version control with multiple teams

Taking the pain out of branching & merging

henrik.kniberg @crisp.se
Version 1.2, 2008-04-04

Introduction

Purpose of this paper

If we have several agile development teams working on the same codebase, how do we minimize the risk of stumbling over each other? How do we ensure that there always is a clean, releasable version at the end of each iteration?

This paper describes an example of how to handle version control in an agile environment with multiple teams. I'm going to assume that you are familiar with the basic elements of Scrum and XP and taskboards.

I didn't invent this scheme – it is based on the "mainline model" or "stable trunk pattern". See the references section for more info.

I wrote this paper because I keep bumping into teams that really need something like this. Most teams seem to like the model quite a lot once they understand it. This is also the scheme that we migrated to at the company described in "Scrum and XP from the Trenches". It really helped us develop and release software in a more agile manner.

By describing the model in an easy-to-read fashion maybe I won't have to explain it as often over a whiteboard :o)

Who is this paper for?

Anyone directly involved in agile software development, regardless of role. Branching and merging is *everybody's* business, not just the configuration manager.

This paper is not primarily targeted for version control experts, in fact such experts probably won't find anything new here. This paper is targeted for the rest of us, those of us that just want to learn simple and useful ways to collaborate.

Disclaimer

This is only one pattern among many, not a silver bullet. If you do choose to use this pattern you will probably need to adapt it to match your particular context.

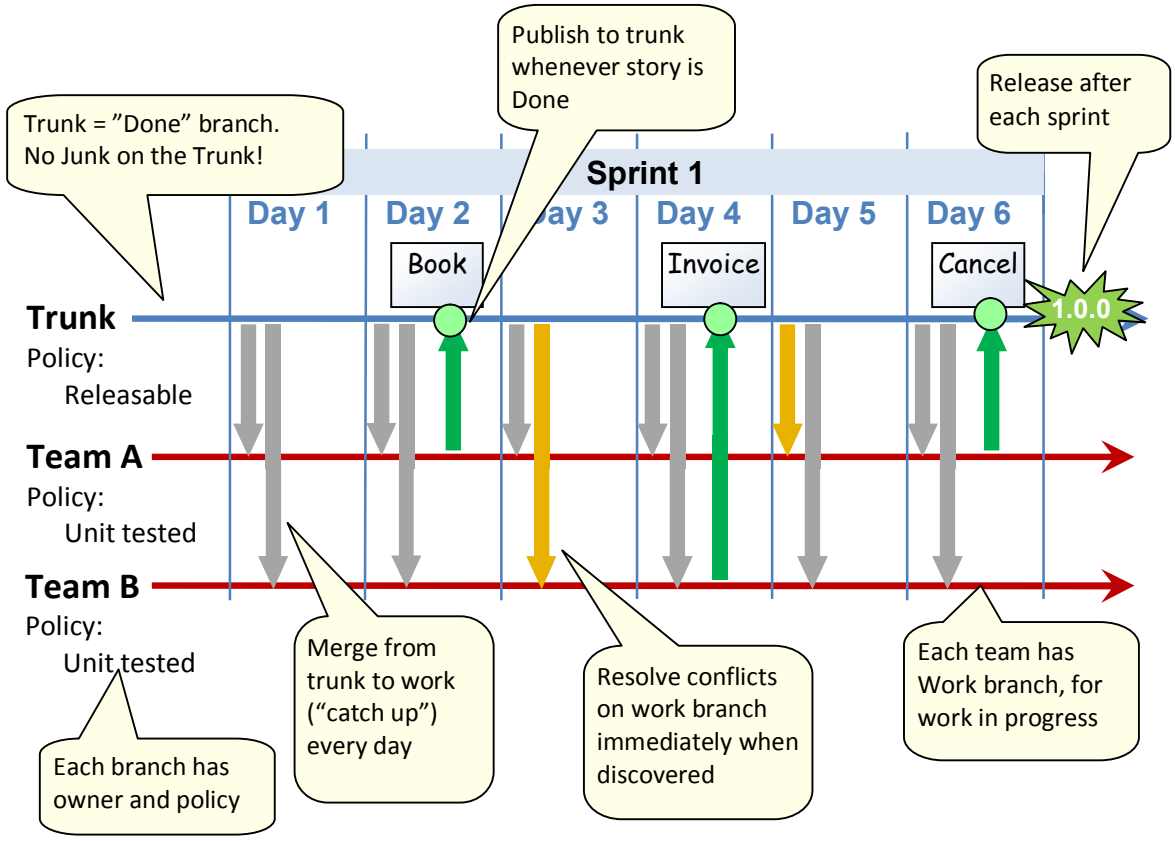
Goals

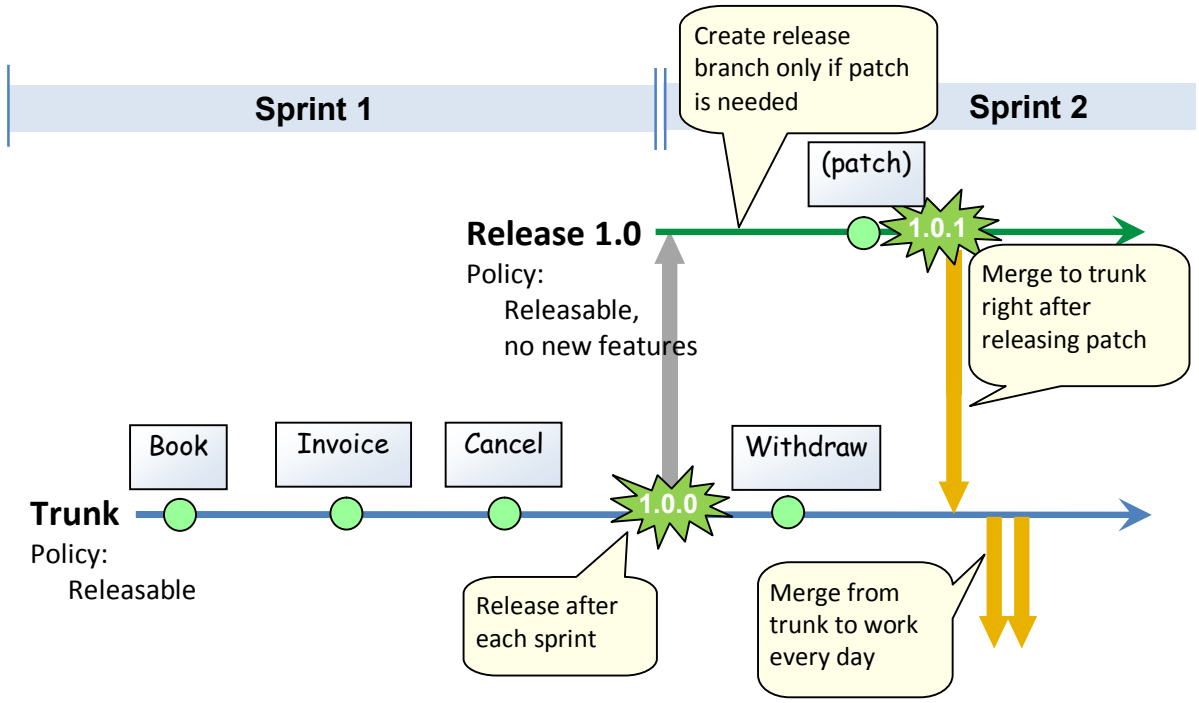
In an agile environment with multiple teams the version control model needs to achieve the following goals:

- **Fail fast**
  - Code conflicts and integration problems should be discovered as soon as possible.
  - Better to fix small problems often than to fix large problems seldom.
- **Always releasable**
  - Even after a really bad sprint there should be at least *something* that is releasable.
- **Simple**
  - All team members will be using this scheme every day, so the rules and routines must be clear and simple.

Single page summary (for wall mounting)

If this picture is confusing to you, don't worry, just read the paper.
If this picture is obvious to you, don't bother reading the paper.





The version control pattern

Branch owner & policy

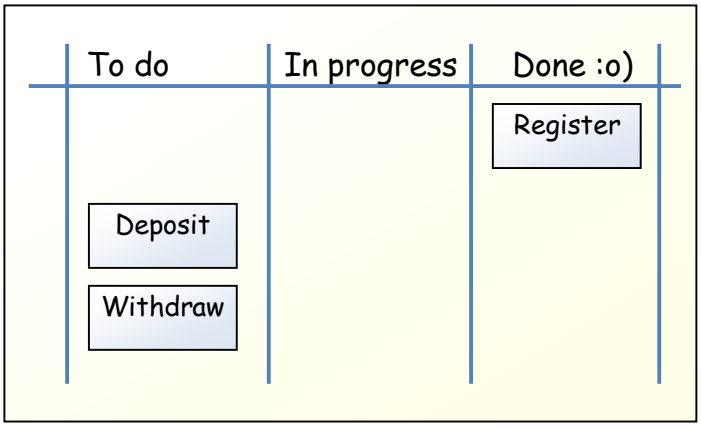
Here's a simple rule.

> **Rule:** Each branch (even the trunk) has an owner and a policy.

Otherwise we get a mess. The policy describes rules for what kind of stuff is allowed to be checked into this branch. The owner is the person responsible for defining and following up the policy.

The "done" concept

When is a user story "done"? More specifically, when your team moves one particular user story into the "done" column on their taskboard, what does that actually *mean*?



I'll assume the following.

> **Assumption:** Definition of Done = "releasable".

So when a team member says that a story is Done and moves the story card into the Done column, a customer could run into the room at that moment and say "Great! Let's go live *now*!", and nobody in the team will say "no but wait".

You can use whatever definition of Done you like. But remember – if the definition is anything less than "releasable" you need to consider: what is *not* included in Done? And who will do that other stuff? When? And what happens if something goes wrong after Done?

The Done branch

When a story is Done, it needs a home. With my definition of Done ("releasable") that means there must be some branch in the system that you could release from in order to get that story into production. That's the Done branch.

Any branch could be the Done branch. I'm going to use the trunk as the Done branch (a good starting point). Sometimes this is called the "mainline".

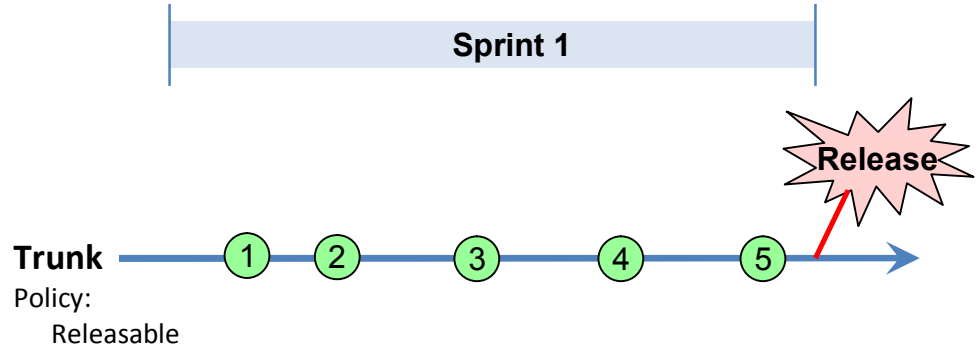
> **Assumption:** Trunk is the Done branch.

Trunk policy:

- Can release at any time
- Want to release ASAP

"Can release at any time" means just that. At any moment, the product owner can decide that we should make a new production release directly from the tip of the trunk.

Here's an example.



The blue line represents the trunk. Each green ball represents one checkin. So 5 things were checked in during this sprint. We could release from the trunk at any time, although we will typically do it at the end of each sprint. If half the team gets sick near the end of the sprint and they don't have time to finish story #5, we could still make a release. And, of course, we could choose *not* to make a release and instead wait another sprint to get story #5 in.

"Want to release ASAP" means that we shouldn't check in a story unless we *want* it to go live (or at least *wouldn't mind* if it goes live). If story #3 above is something that I don't want to release, I've essentially killed the branch. Since the branch is no longer releasable, I've violated the branch policy.

> **Rule:** Don't combine different release cycles on a single branch

When do we create additional branches?

As seldom as possible. Here's a good rule of thumb.

> **Recommendation:** Only create a new branch when you have something you want to check in, and there is no existing branch that you can use without violating its branch policy.

Work branches

OK, so let's say we have our nice and clean trunk, releasable at any moment.

Wait a sec. Releasable means *integration tested*. And that means we need to *run integration tests*. So we need to run integration tests *before* checking in to the trunk.

So where do I check in code that I *believe* is done, but need to verify before checking in to the trunk? Of course, I could test it locally on my machine, and then check it directly into trunk. But that's a bit scary, I'm sure you've all run into the "hey but it works on *my* machine" issue.

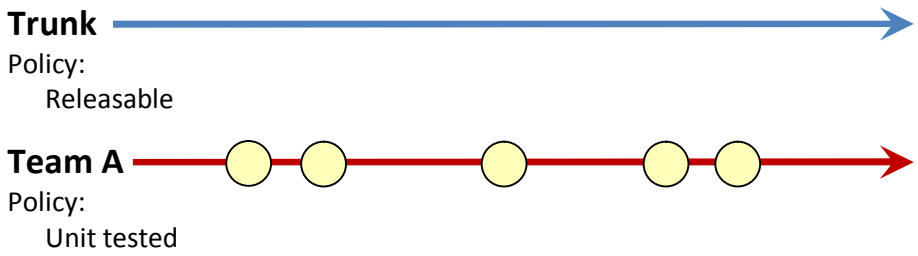
Another issue is "OK, I'm finished coding for today, I'm going home now. Where do I check in my code? It's not tested yet, so I can't check in to the trunk. I want to check it in somewhere so that other team members can continue working on it" (agile team = collective code ownership right?)

There we go. We have stuff we'd like to check in, and there's no place we can check in without violating a branch policy. That's a valid reason to create a new branch.

Let's call this a *work* branch, shared by all members of the team. Some people call it a *development* branch.

Team A work branch policy:

- Code compiles & builds, all unit tests pass.



Great, so now we have two branches! One stable branch (the trunk) and one slightly less stable branch (the team branch). The team branch is red to highlight that it is less stable, i.e. the stuff on that branch passes the unit tests but it might not be integration tested and might not be stable enough for release. OK, so the now team has a place to check in work in progress!

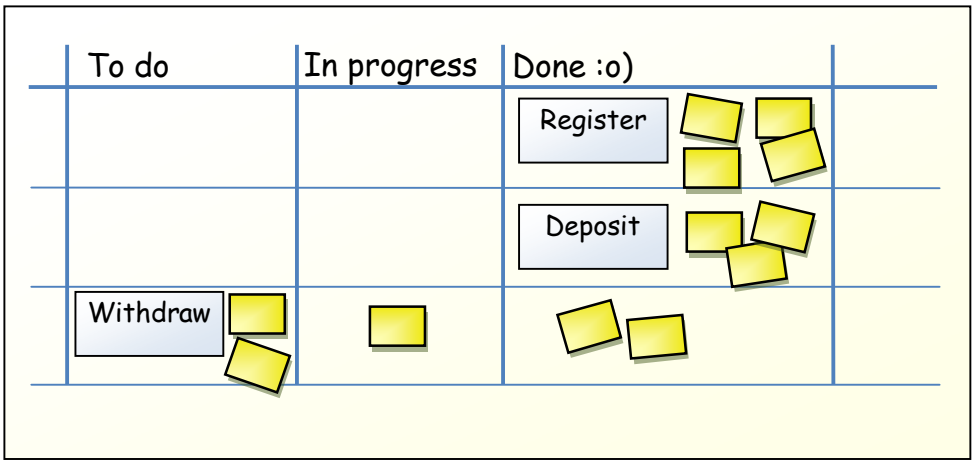
Hmmmm. What about synchronizing the branches then? Read on.

Publishing from work to trunk

At some point (hopefully) a story will get Done. More specifically, at some point the work branch will get to a point where it is *releasable*. At that point we can (and should) publish to the trunk, i.e. take all new code on the work branch and copy over to the trunk. After that the trunk and the work branch are identical.

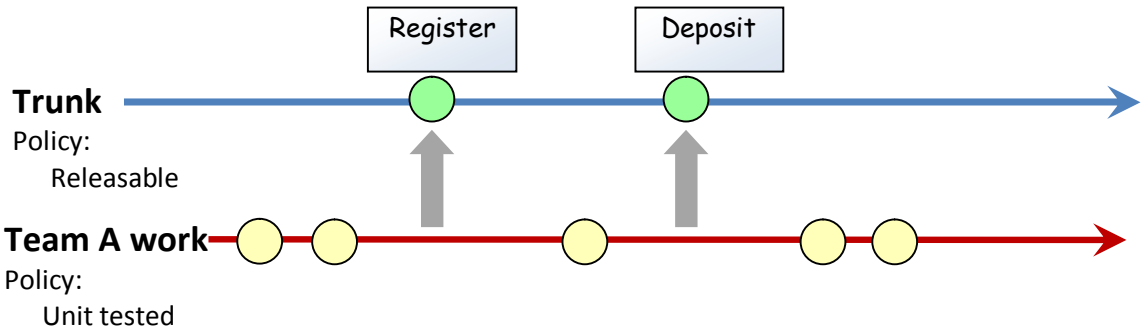
We can call this "publishing" since we have been doing some work and are now ready to "publish" it back to the trunk for release. Just a helpful metaphor.

Here's an example. Let's say we've implemented two stories: Register and Deposit. They are Done, i.e. unit tested, integration tested, and releasable. We have started working on Withdraw, but it is not Done yet. The taskboard would look something like this:



Each yellow stickynote on the board represents a task, i.e. one piece of work needed to complete the story. For example edit class X, update the build script, etc. Each task typically represents about 1 man-day of work, each story typically represents about 3 – 8 man-days of work.

The branch history would look something like this:



So first our team implements Register. Checkin to the work branch, run the integration tests, fix some problems, checkin again, run the tests again, works! Register is Done! Publish it to the trunk.

Next implement Deposit. Only one checkin was needed. Integration tests pass, so we release to trunk again.

Now the team is in the middle of implementing Withdraw. They have done two checkins so far, but aren't done.

Note that "release to trunk" doesn't mean that we are copying the code for one specific story to trunk. It means we are copying everything from work to trunk, i.e. doing a complete synchronization.

So two interesting questions arise:

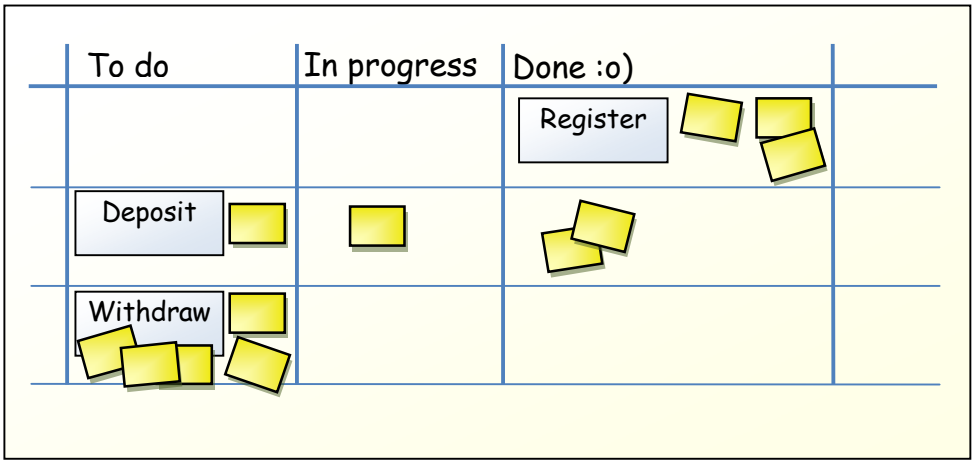
1. What if our team implements *multiple stories in parallell*?
2. What if *other* teams are publishing to the trunk as well?

Let's do one at a time.

What if our team implements multiple stories in parallell?

If the team implements one story at a time, publishing to trunk is trivial. As soon as a story is implemented and tested on the work branch we copy everything from work to trunk. Done.

But wait. What if we are doing multiple stories simultaneously within the team? What if Register is done and Deposit is in progress?

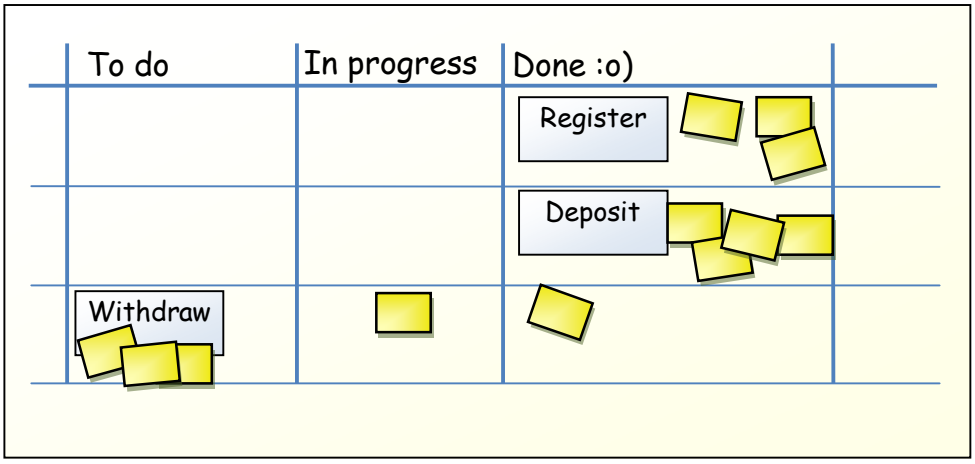


If we synchronize to trunk at this point, we will be including the *partially complete* Deposit story, which isn't releasable! <loud siren> Violation of trunk policy!</loud siren>

Of course, we could wait for Deposit to complete.

(waiting...)

OK Deposit is complete now! Great! Wait a sec... now somebody has started working on Withdraw! Ack! Same problem!



If one of the Deposit tests fail it will be hard to know if that is due to the Deposit code or the partially complete Withdraw code that is checked in to the same branch.

Waiting doesn't help. We are rolling a snowball, saving up for a big bang release at some hypothetical point in the future when all stories are complete (if that ever happens).

This is an extremely common problem. So what do we do?

Here are some strategies:

- Don't do so much parallell development. Try to focus the team on one story at a time.
- If someone is going to start work on Deposit before Register is complete, wait to check in the Deposit code until Register is complete. Or maybe even check in Deposit on a separate temporary branch if you enjoy juggling branches.
- If someone is going to start work on Deposit before Register is complete, start with the safe and invisible bits, code changes that aren't going to affect the releasability of the branch. For example if Deposit requires some new code and some modification of existing code, implement the new code now (new methods, new classes, new tests, etc) but not the modifications. If Deposit needs new GUI elements then make them invisible for now. Once Register is complete and released to the trunk we can start implementing the rest of Deposit.

Here's a convenient rule set:

- Anyone working on the top priority story is King.
- Everyone else on the team is a Servant.
- You want to be King. Try to find ways to help out with the top priority story.
- Whenever a King needs help, Servants immediately offer their services.
- A Servant may not disrupt a King.
- A Servant may never check in unreleasable code on the work branch. A King may check in whatever he pleases (as long as he doesn't violate a branch policy of course).
- As soon as the top priority story is Done, anyone working on the next story is now King.

You might even get a few crowns for the team :o)

On the whole, many teams seem to over-rate the benefits of implementing many stories concurrently. It gives a nice feeling of speed but can be an illusion since it pushes the risky and time consuming bits to the end – merging and integrating and testing.

That's why Scrum teams are supposed to be small (< 9 people) – so that they can collaborate closely and focus their efforts. If everyone does their own story concurrently there probably isn't much collaboration going on. Sure you can have people looking ahead, preparing the next story and doing some of the implementation. But at any given moment the bulk of the team's efforts should be focused on the top priority story.

Multiple teams are a different story. Multiple teams are created when you specifically want to implement multiple stories in parallell. We'll look at that in a moment. First I want to talk a bit about regression testing and diverging code within the team.

Done includes regression testing!

When a story is Done we move it to the Done column and copy from work to trunk. The trunk must always be releasable. This has an important implication.

> **Rule:** Whoever touches the trunk is responsible for ensuring that the whole trunk stays releasable – including all previous functionality!

This rule means, in effect, that testing story A also includes running all relevant regression tests for previously implemented stories. It is not enough that story A works if that code has broken previous stories.

> Wait a sec. Isn't that unreasonable? Running all regression tests every time a story is complete?
>
> Well, first of all I didn't say *all* regressions tests. I said all *relevant* regression tests. Remember that we had a nice clean releasable trunk as a basis, and now we are just adding one story! That is a fairly small incremental change. If our regression tests are automated we could run them all. But if there are manual regression tests we will have to be selective.
>
> It all boils down to a risk vs cost tradeoff. For each manual regression test we should evaluate the cost of the test (i.e. how much work it is to do the test) vs the likelihood that any important defects will be discovered. And of course weigh in the cost of automating that test :o)

Diverging code (merging conflicts)

What if I'm happily writing code that calls the Widget class. What I don't know is that my team member Jim removed the Widget class one hour ago, in conjunction with a refactoring. So now we have *diverging code*. I want to discover this *as soon as possible*, before wasting time writing even more code that calls the Widget class.

Rule: Synchronize your code with the work branch continuously (= as often as possible)

Synchronize in this case means both directions. Merge down the latest from the work branch, and check in your code. The first part can be called "catching up" (= I want to find out what other people have checked in). The second part can be called "publishing" (= I want my changes to be made available to the rest of the team).

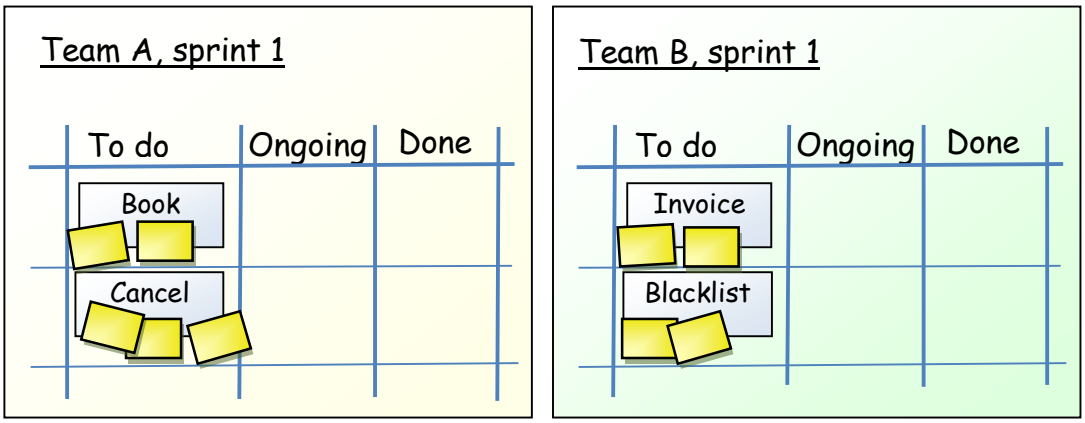
Once per hour or so is a good habit, basically whenever you move from one task to another and aren't in the middle of a flow. This is not just about "I want to find out ASAP when someone else writes code that conflicts with mine", it's also about "I want the other guys to find ASAP if I write code that conflicts with theirs". Just remember to not break the policy of the work branch (unit tests pass and whatnot).

This rule may sound rather obvious but bear with me. I want to make this crystal clear since we will be reusing this mindset further down when dealing with multiple teams.

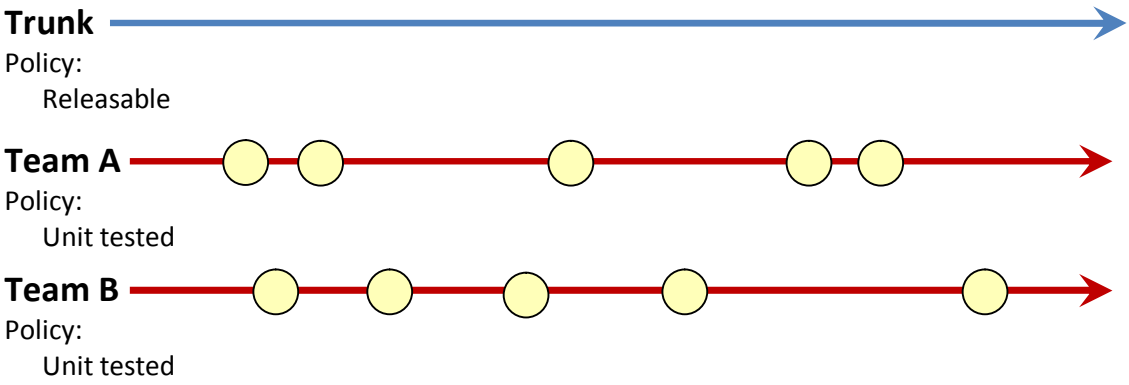
Multiple teams - what if *other* teams are publishing to the trunk as well?

Let's say we have team A and team B. They are cross functional feature teams working on a flight booking system. Team A is focusing on the booking process, team B is focusing on the backoffice stuff.

Let's say they are going to start a sprint now, with two user stories per team (usually there will be more stories than that in a sprint).



Each team has their own work branch, since they each need somewhere to test their code before publishing to the trunk.

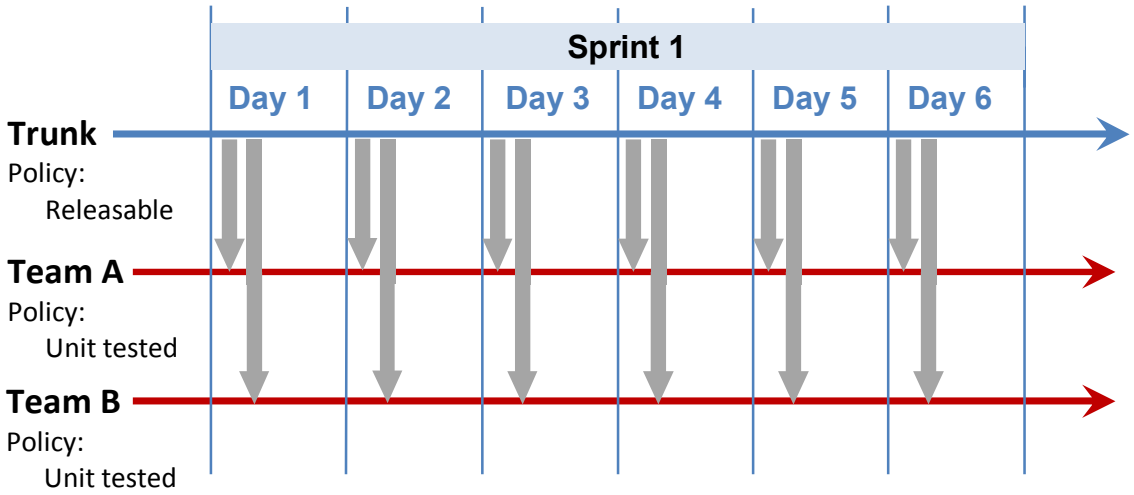


Now we have an interesting problem. Let's say I'm in team A and we have our work branch. Changes might be made in the trunk, without being made in my work branch first! Why? Well because there's another team out there, and they will publish to the trunk whenever they finish a story!

So at any given moment, there may be new code on the trunk that I don't know about. And that code might (god forbid) conflict with my code! Maybe someone in team B renamed the Widget class that I call from my code and... uh... wait a sec. Didn't we just talk about this?

Yep, that's right. It's the same problem. And same solution. But on a slightly different scale.

Rule: Merge from trunk to your work branch every day



Every day when I get to work, someone in my team is responsible for merging the latest version from trunk to our team work branch (= "catching up" with changes made to the trunk).

If my team (team A) discovers a code conflict we resolve it immediately – that's top priority! If we need help from team B (or whoever wrote the code that conflicts with ours) go fetch them and work together to sort it out. The important thing is that my team is responsible for getting the problem sorted out, and that we need to sort it out on our own work branch (not on the trunk).

Rule: Resolve conflicts on the branch that is least stable

Of course, merging from trunk is a waste of time if people don't publish to the trunk on a regular basis. Any divergence between team A and team B is invisible until someone publishes to the trunk. So here's the next rule:

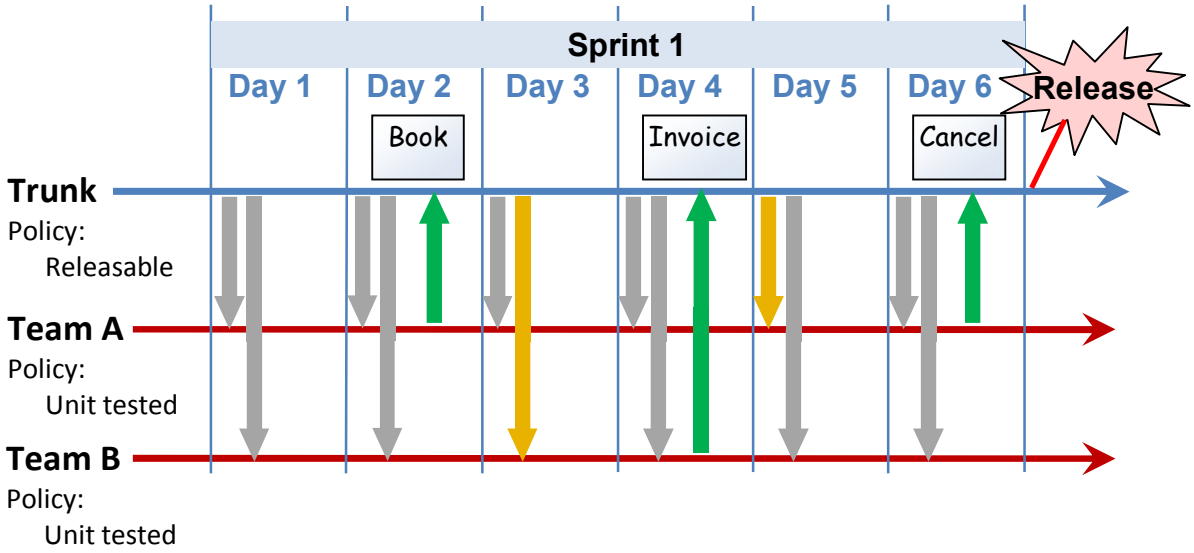
Rule: Merge from your work branch to the trunk on a regular basis, for example whenever a story is done. Don't wait until the end of the sprint!

Note an interesting side effect here:

Side effect: Whoever checks in first wins!

If two teams are writing code that conflicts with each other, *the last team to check in* has to resolve the conflict. That's a nice side effect, since it encourages teams to check in early :o)

Here's an example of a whole sprint.



We are doing a 6 day sprint with two teams. Team A is planning to implement Book and Cancel. Team B is planning to implement Invoice and Blacklist. Let's look at what happened.

Day	**Team A** perspective	**Team B** perspective	**Trunk** perspective
1	Merge from trunk. Nothing new. Working on Book, checking in on our work branch.	Merge from trunk. Nothing new. Working on Invoice, checking in on our work branch.	Nothing happened today.
2	Merged from trunk. Nothing new. Finished implementing Book. Integration-tested it. Done! Copy to trunk. Start working on Cancel.	Same as yesterday.	Book is now done!
3	Merge from trunk. Nothing new. Still working on Cancel.	Merge from trunk. Aha! There have been changes! Book has been added! Merge with our code in Team B branch, resolve any conflicts. Then keep working on Invoice.	Nothing happened today.
4	Same as yesterday.	Merged from trunk. Nothing new. Finished Invoice. Integration test it (including Book!), copy to trunk. Start working on Blacklist.	Invoice is now done!
5	Merge from trunk. Aha! There have been changes! Invoice code has been added! Merge with our code in Team A branch, resolve any conflicts. Then keep working on Cancel.	Merged from trunk. Nothing new. Still working on Blacklist.	Nothing happened today.
6	Merged from trunk. Nothing new. Finished Cancel, copy to trunk.	Same as yesterday.	Cancel is now done!

Sprint is done! All stories except Blacklist got done. But that's OK, we can still release! This is because we merged & integrated incrementally. If we wait until the end of the sprint to do this, any diverging code will be discovered at precisely the wrong moment – when we have the least time to fix it.

Release branches

Let's say we've finished sprint 1 and released version 1.0.0 of the system. Now, while we're in the middle of sprint 2, a serious defect is reported on the released version! Oh no! What do we do?

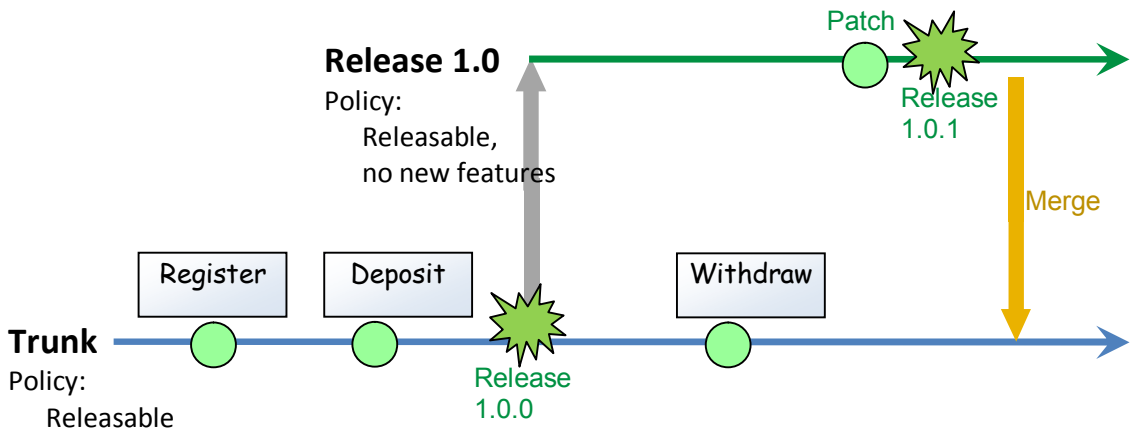
The simplest way is to simply fix it on the trunk and release a version 1.1.0. This means any new stories implemented so far during sprint 2 will be included in the new release. Theoretically this should be fine since the trunk is the Done branch and the definition of Done is "releasable". So whatever is on the trunk at any moment should be something that we want to release.

However there may be reasons *not* to want to release new stories right now. For example:

- The fact that there is a serious defect essentially means that the trunk was broken at the time of the release. That in turn means that the sprint 2 stories are built on top of a broken foundation. We might want to fix the foundation itself without having to deal with the new stories as well.
- Maybe the stakeholders don't want new features to be released in mid-sprint.
- Maybe it takes a while to make a new release from the trunk, with new features and all, so we want a simple "hotfix" mechanism to get bugfixes out the door faster.

So how do we do it?

1. Create a *release branch* called release 1.0, based on the trunk at the time it was released.
2. Patch the defect on the release branch.
3. Merge from release branch to trunk immediately after releasing (so the patch is included in future releases).



Note that we don't have to create the release 1.0 branch when we do release 1.0.0. We can wait until the defect comes up. So we don't need to create a branch until there is something that actually needs to be done on that branch.

The big picture

OK, now I've gone through a fairly detailed example of how to put this pattern to use. Now let's back out a bit and look at the big picture.

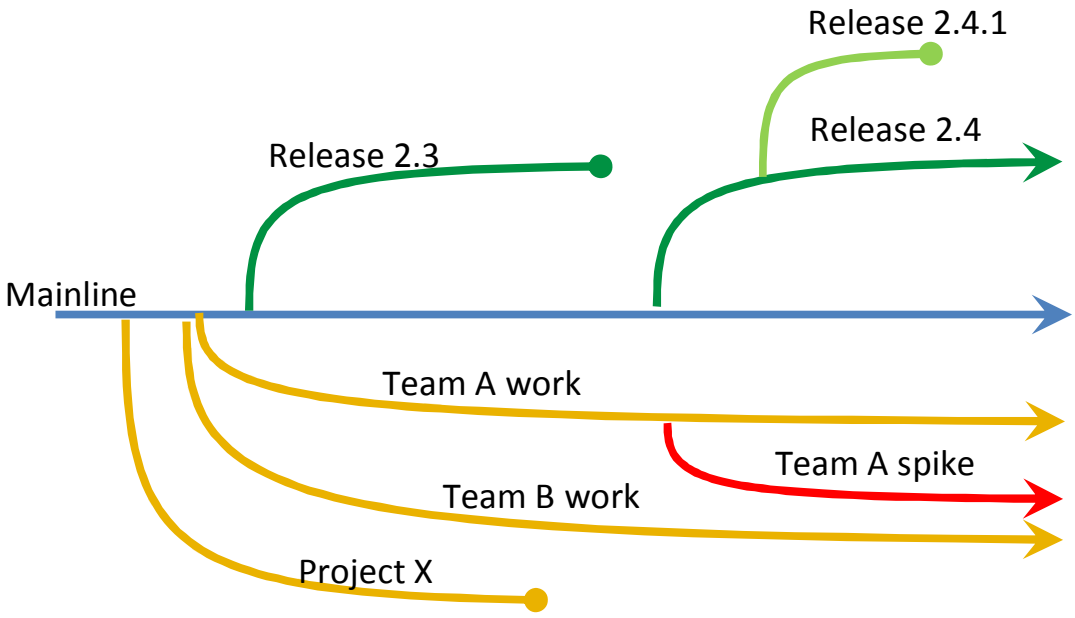
In the mainline model, a branch is called a *codeline* (in fact, branch is considered to be an implementation of a codeline). Sometimes these are called *streams*.

A codeline's parent (i.e. the codeline that it originated from) is called its *baseline*. Mainline is the codeline that has no baseline.

So in our examples above we could conclude that:

- The trunk is our mainline. It has no parent right?
- All other codelines (release 1.0, team A work, team B work) have the trunk as baseline.

Here's a more complex example:



This picture tells us that:

- The project X codeline was spawned from the mainline. The project is now complete, so the branch is closed.
- Team A has an active work branch that was spawned from the mainline.
- Team A also has an ongoing spike that was spawned from the work branch.
- The release 2.3 branch is closed, since 2.3 is no longer in production and won't be maintained.

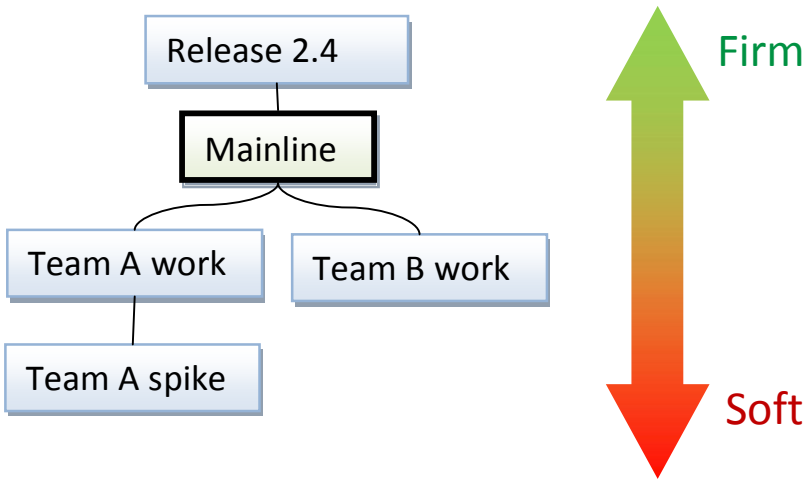
Each codeline has a relative *firmness level* with respect to its baseline, i.e. each codeline is either *more firm* or *less firm* (softer) than its baseline.

- A *firm* codeline is stable, thoroughly tested, changes seldom, and is close to release.
- A *soft* codeline is unstable, barely tested, changes often, and is far from release.

When drawing codelines, firm codelines branch upwards and soft codelines branch downwards. So looking at the picture above, we can conclude that:

- Release 2.3 is firmer than mainline.
- Team A work is softer than mainline.
- Team A spike is softer than team A work.

The drawing format used above is useful for showing branch histories, but it can get a bit messy when there are lots of branches. Here's a cleaner format, showing only the currently existing codelines and where they spawned from.



I suggest drawing a picture of your branches in this format and putting up on the wall in the team room! Really helpful to look at when discussing integration issues.

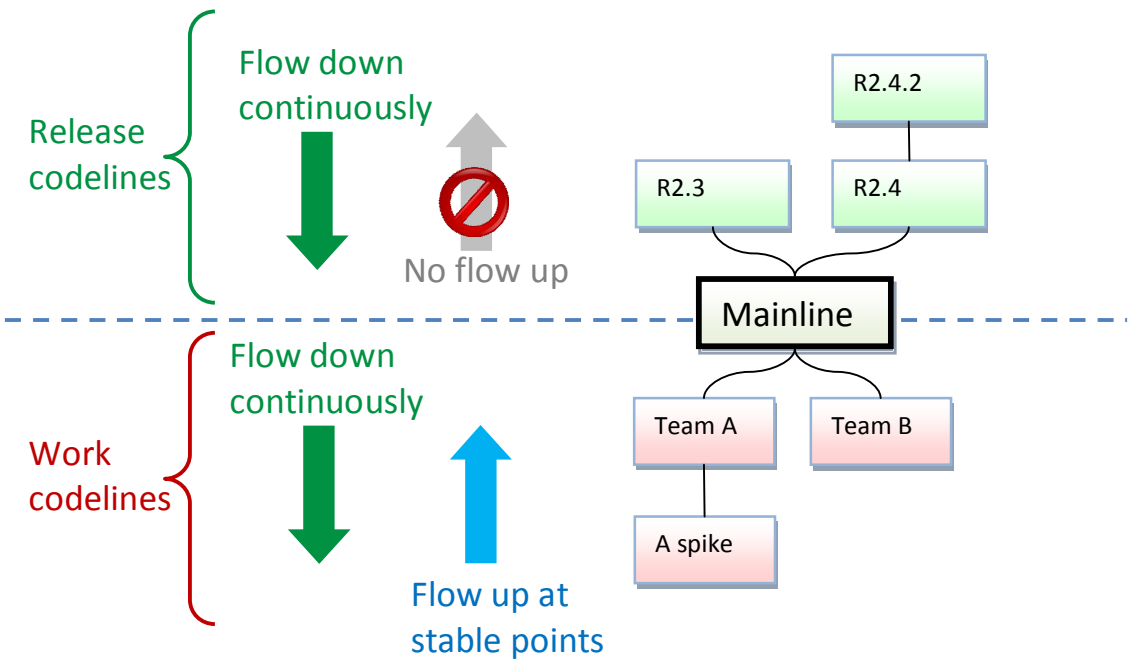
An important rule is that all change flows along the lines! So no code should be merged directly from team A work to team B work, this will cause all kinds of confusion. Instead, changes in team A work should flow to the mainline and then down to team B work.

Everything *above* the mainline is a called a *release codeline*, which means a codeline that is firmer than the mainline.

Everything *below* the mainline is called a *development codeline (*or *work),* which means a codeline that is softer than the mainline.

Golden rule of collaboration:
- Always accept stabilizing changes
- Never impose destabilizing changes

So what does this mean in terms of the different types of codelines?



The picture above is just a colorful way of saying that:

- Whenever a change is made on a release codeline, the change should immediately be merged down to its baseline, all the way down to the mainline.
  - **Example:** A bug is fixed on R2.4.2. That should immediately be merged down to R2.4, and merged from there down to mainline.
- A release codeline should never receive changes from its baseline.
  - **Example:** New code is checked in to the mainline. This should not propagated up to R2.3 and R2.4.
- Change flows continuously from baselines to development codelines.
  - **Example:** Any change to the mainline should quickly flow on down to Team A and Team B, and from Team A on down to A spike.
- Change flows from a development codeline to its baseline only at stable points.
  - **Example:** Team B merges to mainline only when a story is complete and tested.

Whenever changes have been made to a codeline and its baseline, some kind of merge might be needed. Since code merging is a potentially error prone operation we want to do it on the softer of the two codelines. Once the merge is complete and verified, we can copy the merged code back to the firmer codeline.

Using our convention of drawing firm codelines higher up than soft codelines, we could derive a simple rule:

> **Rule:** Merge down, copy up

Example: Team A notices that the mainline has been updated. They merge the changes down to their development branch and fix any conflicts. As soon as Team A development branch reaches a stable point they copy back up to the mainline. They do of course have to check that nothing else has been changed on the mainline in the meantime.

Variations to the model

The chapter “The version control pattern” describes an example of how to implement the mainline model.

The chapter “The big picture” describes the mainline model in more general terms.

In this chapter I’m going to mention some typical variations on how to implement this pattern.

Definition of Done doesn’t have to be “releasable”

Just settle on any definition of Done, and make sure there is a branch to accomodate stories that are Done according to that definition. Be careful about leaving important things out of Done, though. If integration testing is not included in Done, when will you integration test?

Trunk doesn’t have to be mainline

You need a mainline for this pattern to work. It doesn’t necessarily have to be the trunk though (although the trunk would be a pretty natural choice in most cases).

Teams don’t have to have their own branches

You can certainly have multiple teams sharing the same branch, or even working directly off the mainline. As long as you obey the policy of the branch.

Often teams like to have their own branches to avoid having partially complete stories causing interference between teams.

You don’t need to create a new release branch each time

You might decide to use the same release branch instead of creating a new one after each sprint. That release branch could be called “currently in production” or something like that. Certainly a useful model if you never have more than one version in production at a time.

You don’t need to release after each sprint

You can release after each story. Or after every third sprint. Choose your pace.

You don’t have to run regression tests for each story

If regression testing or integration is truly a pain in your environment you may decide to save it to the end of the sprint, so you can batch up a few stories and test/integrate them together. That is your risk to take. If regression testing & integration is included in your definition of done, that simply means you risk getting 0 stories Done if you run into problems with this at the end of the sprint.

FAQ

Where does continuous integration (CI) fit into this?

Which branches should your CI server work on? This is really context dependent, but here is a good starting point.

Assuming that your policy for trunk is "Done & releasable", and your policy for each work branch is "unit tests pass":

- For each work branch, CI server automatically and continuously checks that it builds and passes all unit tests.
  - Issues a red alert if anything fails. Triggers the smoke machine.
- For each work branch, CI server automatically and regularly (if not continuously) runs integration tests and regression tests.
  - Displays a discrete warning if anything fails, since this is not part of the branch policy.
  - This test is triggered manually whenever someone is considering to publish code from work to trunk, as a way to check that the story is Done.
- For the trunk, the CI server automatically and continuously runs integration tests and regression tests.
  - Issues a red alert if anything fails. Triggers the smoke machine, the siren, the USB rocket launcher, and calls in the national guard.

What's the best tool for this version control model?

Not sure. I know it works well with Perforce, I think it works with subversion, but I'm not too sure about CVS. Any input is welcome.

Remember one important thing though – the cost of switching tools is usually quite low compared to the cost of not being able to collaborate effectively! So figure out how you want to work, *then* find the right tool to support that.

What about checkins that aren't related to a user story?

Not all code changes have to be related to a user story, I just did that in all my examples for clarity. The same model applies no matter what type of code you are checking in (or documents for that matter).

Merging is a pain, so I want to do it as seldom as possible!

Congratudolences, you have mergophobia - the irrational fear of merging code!

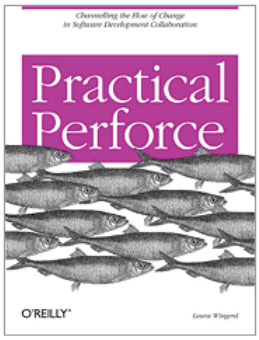
Merging is a pain *because* you do it as seldom as possible. The more often you merge, the less painful it is :o)

I have more questions!

Check out the references on the next page! I'm sure they will answer most of your questions.

References

I can strongly recommend these resources if you want to read more.



Practical Perforce

By Laura Wingerd. Here's a sample chapter that covers most of mainline model:

http://www.oreilly.com/catalog/practicalperforce/chapter/ch07.pdf

It's a Perforce book, but the mainline model is by no means specific to Perforce.

High level best practices in Software Configuration Management

A really useful summary of general best practices for version control by Laura Wingerd & Christopher Seiwald.

http://www.perforce.com/perforce/bestpractices.html

Branching and merging – an agile perspective

An interesting article by Robert Cowham that relates to this.

http://www.cmcrossroads.com/articles/agile-cm-environments/branching-and-merging-%11-an-agile-perspective.html

The Flow of Change

A great summary of the mainline model, again by Laura Wingerd! Most of the "Big Picture" chapter above is based on these slides.

http://www.perforce.com/perforce/conferences/us/2005/presentations/Wingerd.pdf