Patterns for Managing Source Code Branches

Modern source-control systems provide powerful tools that make it easy to create branches in source code. But eventually these branches have to be merged back together, and many teams spend an inordinate amount of time coping with their tangled thicket of branches. There are several patterns that can allow teams to use branching effectively, concentrating around integrating the work of multiple developers and organizing the path to production releases. The over-arching theme is that branches should be integrated frequently and efforts focused on a healthy mainline that can be deployed into production with minimal effort.

28 May 2020

CONTENTS

SIDEBARS

Source code is a vital asset to any software development team, and over the decades a set of source code management tools have been developed to keep code in shape. These tools allow changes to be tracked, so we recreate previous versions of the software and see how it develops over time. These tools are also central to the coordination of a team of multiple programmers, all working on a common codebase. By recording the changes each developer makes, these systems can keep track of many lines of work at once, and help developers work out how to merge these lines of work together.

This division of development into lines of work that split and merge is central to the workflow of software development teams, and several patterns have evolved to help us keep a handle on all this activity. Like most software patterns, few of them are gold standards that all teams should follow. Software development workflow is very dependent on context, in particular the social structure of the team and the other practices that the team follows.

My task in this article is to discuss these patterns, and I'm doing so in the context of a single article where I describe the patterns but intersperse the pattern explanations with narrative sections that better explain context and the interrelationships between them. To help make it easier to distinguish them, I've identified the pattern sections with the "✢" dingbat.

Base Patterns

In thinking about these patterns, I find it useful to develop two main categories. One group looks at integration, how multiple developers combine their work into a coherent whole. The other looks at the path to production, using branching to help manage the route from an integrated code base to a product running in production. Some patterns underpin both of these, and I'll tackle these now as the base patterns. That leaves a couple of patterns that are neither fundamental, nor fit into the two main groups - so I'll leave those till the end.

✢ Source Branching ✢

> *Create a copy and record all changes to that copy.*

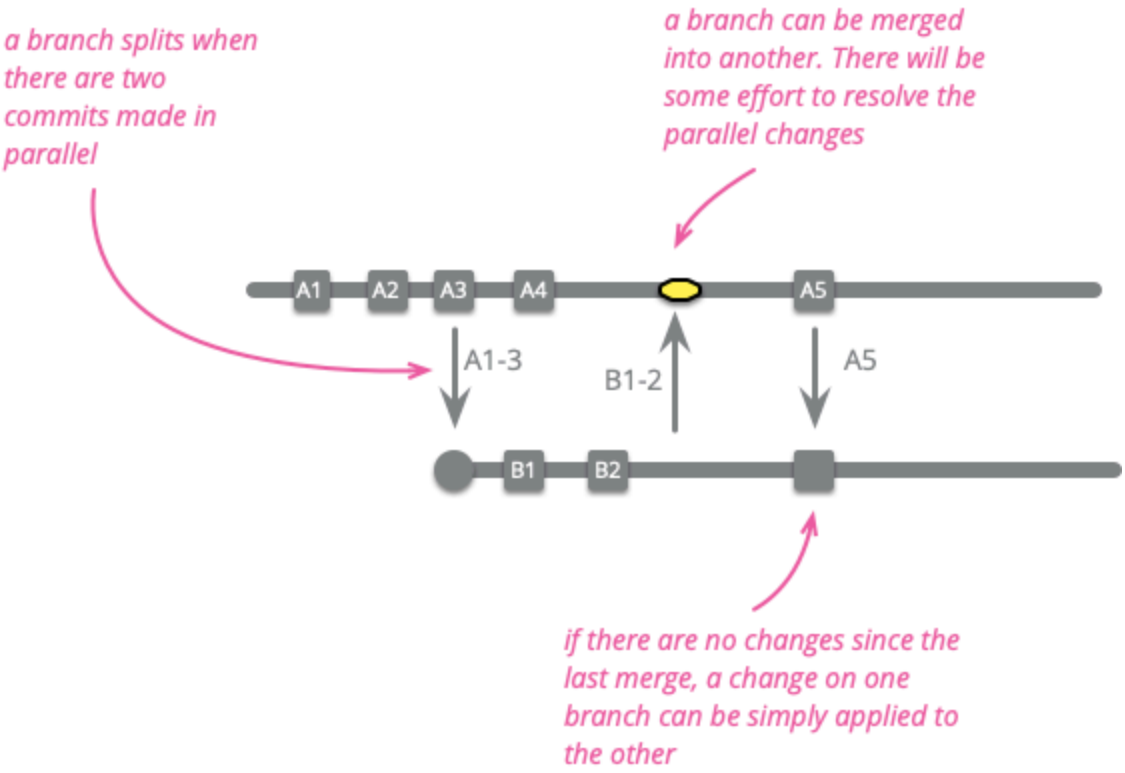
If several people work on the same code base, it quickly becomes impossible for them to work on the same files. If I want to run a compile, and my colleague is the middle of typing an expression, then the compile will fail. We would have to holler at each other: “I'm compiling, don't change anything”. Even with two this would be difficult to sustain, with a larger team it would be incomprehensible.

The simple answer to this is for each developer to take a copy of the code base. Now we can easily work on our own features, but a new problem arises: how do we merge our two copies back together again when we're done?

A source code control system makes this process much easier. The key is that it records every change made to each branch as commit. Not just does this ensure nobody forgets the little change they made to `utils.java`, recording changes makes it easier to perform the merge, particularly when several people have changed the same file.

This leads me to the definition of branch that I'll use for this article. I define a **branch** as a particular sequence of commits to the code base. The **head**, or **tip**, of a branch is the latest commit in that sequence.

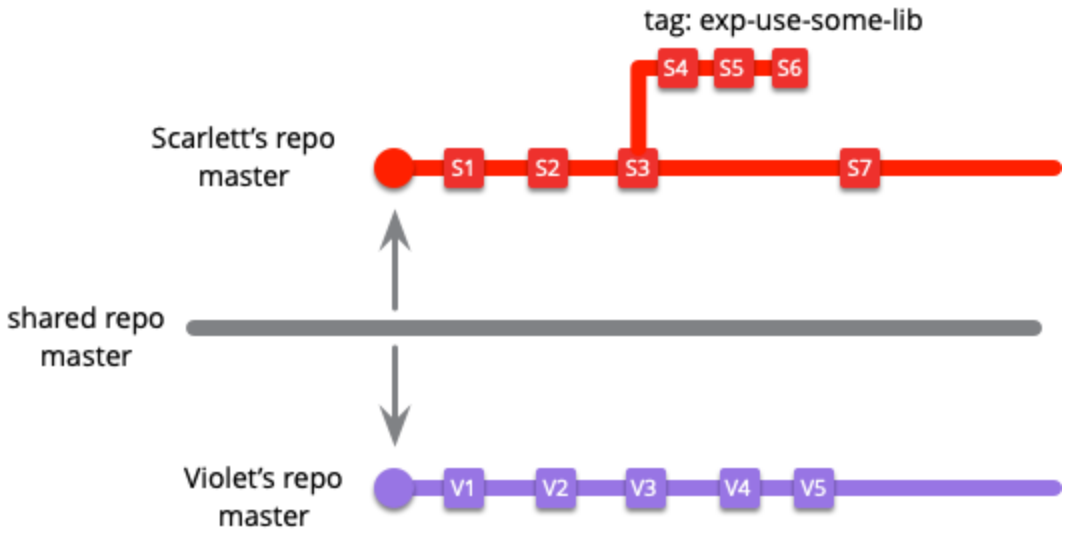
That's the noun, but there's also the verb, “to branch”. By this I mean creating a new branch, which we can also think of as splitting the original branch into two. Branches merge when commits from one branch are applied to another.



The definitions I'm using for “branch” correspond to how I observe most developers talking about them. But source code control systems tend to use “branch” in a more particular way.

I can illustrate this with a common situation in a modern development team that's holding their source code in a shared git repository. One developer, Scarlett, needs to make a few changes so she clones that git repository and checks out the master branch. She makes a couple of changes committing back into her master. Meanwhile, another developer, let's call her Violet, clones the repository onto the her desktop and

checks out the master branch. Are Scarlett and Violet working on the same branch or a different one? They are both working on “master”, but their commits are independent of each other and will need to be merged when they push their changes back to the shared repository. What happens if Scarlett decides she's not sure about the changes that she's made, so she tags the last commit and resets her master branch to origin/master (the last commit she cloned from the shared repository).



According to the definition of branch I gave earlier, Scarlett and Violet are working on separate branches, both separate from each other, and separate from the master branch on the shared repository. When Scarlett puts aside her work with a tag, it's still a branch according to my definition (and she may well think of it as a branch), but in git's parlance it's a tagged line of code.

With distributed version control systems like git, this means we also get additional branches whenever we further clone a repository. If Scarlett clones her local repository to put on her laptop for her train home, she's created a third master branch. The same effect occurs with forking in GitHub - each forked repository has its own extra set of branches.

This terminological confusion gets worse when we run into different version control systems as they all have their own definitions of what constitutes a branch. A branch in Mercurial is quite different to a branch in git, which is closer to Mercurial's bookmark. Mercurial can also branch with unnamed heads and Mercurial folks often branch by cloning repositories.

All of this terminological confusion leads some to avoid the term. A more generic term that's useful here is codeline. I define a **codeline** as a particular sequence of versions of

the code base. It can end in a tag, be a branch, or be lost in git's reflog. You'll notice an intense similarity between my definitions of branch and codeline. Codeline is in many ways the more useful term, and I do use it, but it's not as widely used in practice. So for this article, unless I'm in the particular context of git (or another tool's) terminology, I'll use branch and codeline interchangeably.

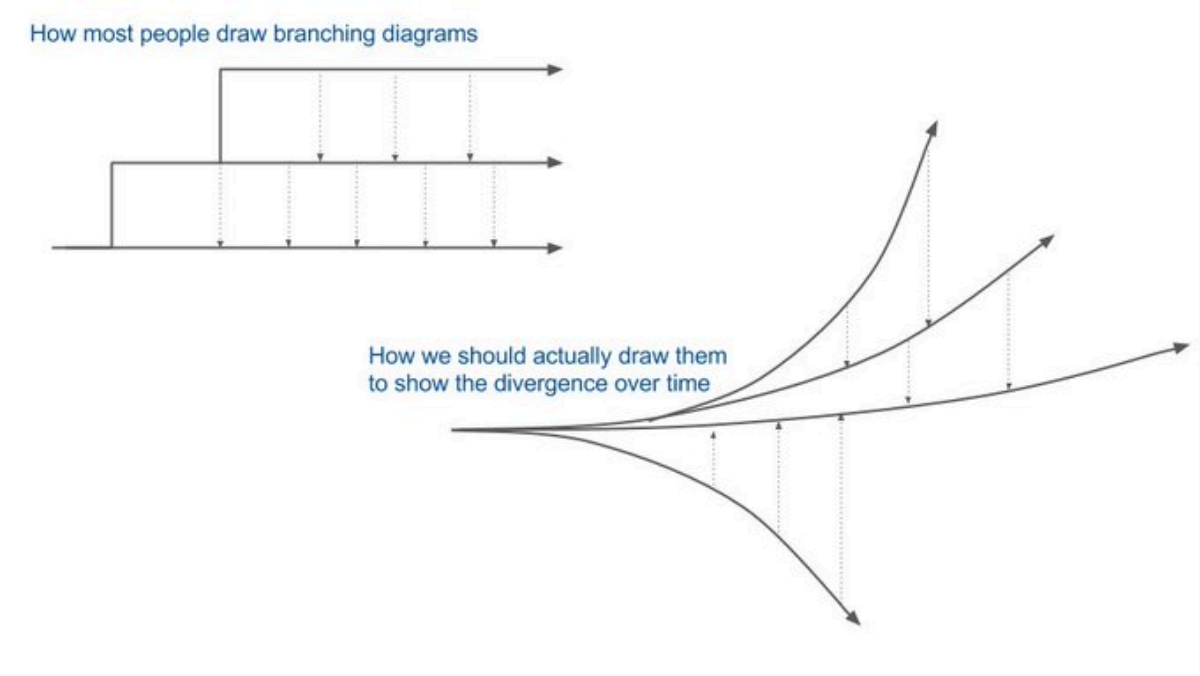
A consequence of this definition is that, whatever version control system you're using, every developer has at least one personal codeline on the working copy on their own machine as soon as they make local changes. If I clone a project's git repo, checkout master, and update some files - that's a new codeline even before I commit anything. Similarly if I make my own working copy of the trunk of a subversion repository, that working copy is its own codeline, even if there's no subversion branch involved.

When to use it

An old joke says that if you fall off a tall building, the falling isn't going to hurt you, but the landing will. So with source code: branching is easy, merging is harder.

Source control systems that record every change on the commit do make the process of merging easier, but they don't make it trivial. If Scarlett and Violet both change the name of a variable, but to different names, then there's a conflict that the source management system cannot resolve without human intervention. To make it more awkward this kind of **textual conflict** is at least something the source code control system can spot and alert the humans to take a look. But often conflicts appear where the text merges without a problem, but the system still doesn't work. Imagine Scarlett changes the name of a function, and Violet adds some code to her branch that calls this function under its old name. This is what I call a **Semantic Conflict**. When these kinds of conflicts happen the system may fail to build, or it may build but fail at run-time.

Jonny LeRoy likes to point out this flaw in how people (including me) draw branching diagrams



The problem is familiar to anyone who has worked with concurrent or distributed computing. We have some shared state (the code base) with developers making updates in parallel. We need to somehow combine these by serializing the changes into some consensus update. Our task is made more complicated by the fact that getting a system to execute and run correctly implies very complex validity criteria for that shared state. There's no way of creating a deterministic algorithm to find consensus. Humans need to find the consensus, and that consensus may involve mixing choice parts of different updates. Often consensus can only be reached with original updates to resolve the conflicts.

> *I start with: "what if there was no branching". Everybody would be editing the live code, half-baked changes would bork the system, people would be stepping all over each other. And so we give individuals the illusion of frozen time, that they are the only ones changing the system and those changes can wait until they are fully baked before risking the system. But this is an illusion and eventually the price for it comes due. Who pays? When? How much? That's what these patterns are discussing: alternatives for paying the piper.*
>
> *-- Kent Beck*

Hence the rest of this article, where I lay out various patterns that support the pleasant isolation and the rush of wind through your hair as you fall, but minimizing the consequences of the inevitable contact with the hard ground.

✢ ✢ ✢

✢ Mainline ✢

A single, shared, branch that acts as the current state of the product

The **mainline** is a special codeline that we consider to be the current state of the team's code. Whenever I wish to start a new piece of work, I'll pull code from mainline into my local repository to begin working on. Whenever I want to share my work with the rest of the team, I'll update that mainline with my work, ideally using the Mainline Integration pattern that I'll discuss shortly.

Different teams use different names for this special branch, often encouraged by the conventions of the version control systems used. git users will often call it “master”, subversion users usually call it “trunk”.

I must stress here that mainline is a *single*, *shared* codeline. When people talk about “master” in git, they can mean several different things, since every repository clone has it's own local master. Usually such teams have a **central repository** - a shared repository that acts as the single point of record for the project and is the origin for most clones. Starting a new piece of work from scratch means cloning that central repository. If I already have a clone, I begin by pulling master from the central repository, so it's up to date with the mainline. In this case mainline is the master branch in the central repository.

While I'm working on my feature, I have my own personal development branch which may be my local master, or I may create a separate local branch. If I'm working on this for a while, I can keep up to date with changes in the mainline by pulling mainline's changes at intervals and merging them into my personal development branch.

Similarly, if I want to create a new version of the product for release, I can start with the current mainline. If I need to fix bugs to make the product stable enough for release, I can use a Release Branch.

When to use it

I remember going to talk to a client's build engineer in the early 2000s. His job was assemble a build of the product the team was working on. He'd send an email to every member of the team, and they would reply by sending over various files from their code base that were ready for integration. He'd then copy those files into his integration tree and try to compile the code base. It would usually take him a couple of weeks to create a build that would compile, and be ready for some form of testing.

In contrast, with a mainline, anyone can quickly start an up-to-date build of the product from the tip of mainline. Furthermore, a mainline doesn't just make it easier to see what the state of the code base is, it's the foundation for many other patterns that I'll be exploring shortly.

One alternative to mainline is Release Train.

✢ ✢ ✢

✢ Healthy Branch ✢

> *On each commit, perform automated checks, usually building and running tests, to ensure there are no defects on the branch*

Since Mainline has this shared, approved status, it's important that it be kept in a stable state. Again in the early 2000s, I remember talking to a team from another organization that was famous for doing daily builds of each of their products. This was considered quite an advanced practice at the time, and this organization was lauded for doing it. What wasn't mentioned in such write ups was that these daily builds didn't always succeed. Indeed it wasn't unusual to find teams whose daily builds hadn't compiled for several months.

To combat this, we can strive to keep a branch healthy - meaning it builds successfully and the software runs with few, if any, bugs. To ensure this, I've found it critical that we write Self Testing Code. This development practice means that as we write the production code, we also write a comprehensive suite of automated tests so that we can be confident that if these tests pass, then the code contains no bugs. If we do this, then we can keep a branch healthy by running a build with every commit, this build includes running this test suite. Should the system fail to compile, or the tests fail, then our number one priority is to fix them before we do anything else on that branch. Often this means we “freeze” the branch - no commits are allowed to it other than fixes to make it healthy again.

There is a tension around the degree of testing to provide sufficient confidence of health. Many more thorough tests require a lot of time to run, delaying feedback on whether the commit is healthy. Teams handle this by separating tests into multiple stages on a Deployment Pipeline. The first stage of these tests should run quickly, usually no more than ten minutes, but still be reasonably comprehensive. I refer to

such a suite as the **commit suite** (although it's often referred to as “the unit tests” since the commit suite usually is mostly Unit Tests).

Ideally the full range of tests should be run on every commit. However if the tests are slow, for example performance tests that need to soak a server for a couple of hours, that isn't practical. These days teams can usually build a commit suite that can run on every commit, and run later stages of the deployment pipeline as often as they can.

That the code runs without bugs is not enough to say that the code is good. In order to maintain a steady pace of delivery, we need to keep the internal quality of the code high. A popular way of doing that is to use Pre-Integration Review, although as we shall see, there are other alternatives.

When to use it

Each team should have clear standards for the health of each branch in their development workflow. There is an immense value in keeping the mainline healthy. If the mainline is healthy then a developer can start a new piece of work by just pulling the current mainline and not be tangled up in defects that get in the way of their work. Too often we hear people spending days trying to fix, or work around, bugs in the code they pull before they can start with a new piece of work.

A healthy mainline also smooths the path to production. A new production candidate can be built at any time from the head of the mainline. The best teams find they need to do little work to stabilize such a code-base, often able to release directly from mainline to production.

Critical to having a healthy mainline is Self Testing Code with a commit suite that runs in a few minutes. It can be a significant investment to build this capability, but once we can ensure within a few minutes that my commit hasn't broken anything, that completely changes our whole development process. We can make changes much more quickly, confidently refactor our code to keep it easy to work with, and drastically reduce the cycle time from a desired capability to code running in production.

For personal development branches, it's wise to keep them healthy since that way it enables Diff Debugging. But that desire runs counter to making frequent commits to checkpoint your current state. I might make a checkpoint even with a failing compile if I'm about to try a different path. The way I resolve this tension is to squash out any unhealthy commits once I'm done with my immediate work. That way only healthy commits remain on my branch beyond a few hours.

If I keep my personal branch healthy, this also makes it much easier to commit to the mainline - I know that any errors that crop up with Mainline Integration are purely due to integration issues, not errors within my codebase alone. This will make it much quicker and easier to find and fix them.

✢ ✢ ✢

Integration Patterns

Branching is about managing the interplay of isolation and integration. Having everyone work on a single shared codebase all the time, doesn't work because I can't compile the program if you're in the middle of typing a variable name. So at least to some degree, we need a notion of a private workspace that I can work on for a while. Modern source code controls tools make it easy to branch and monitor changes to those branches. At some point however we need to integrate. Thinking about branching strategies is really all about deciding how and when we integrate.

✢ Mainline Integration ✢

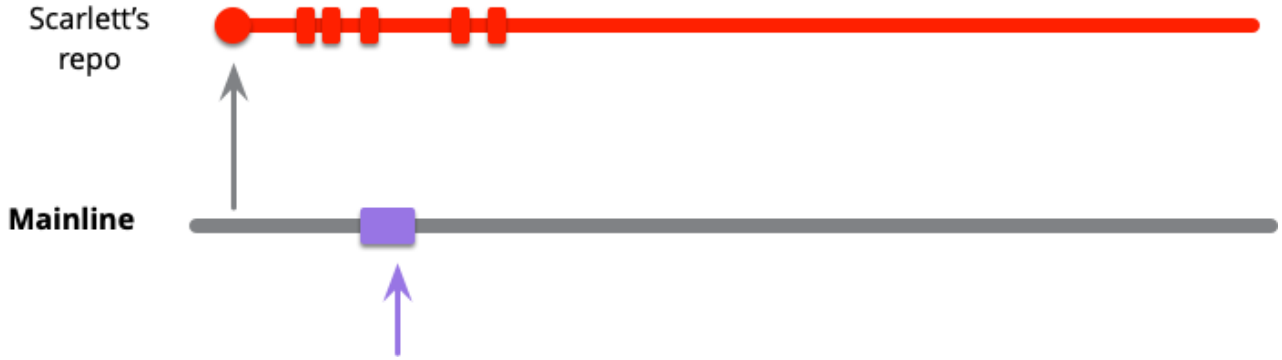
> *Developers integrate their work by pulling from mainline, merging, and - if healthy - pushing back into mainline*

A mainline gives a clear definition of what the current state of the teams' software looks like. One of the biggest benefits of using a mainline is that it simplifies integration. Without mainline, it's the complicated task of coordinating with everyone in the team that I described above. With a mainline however, each developer can integrate on their own.

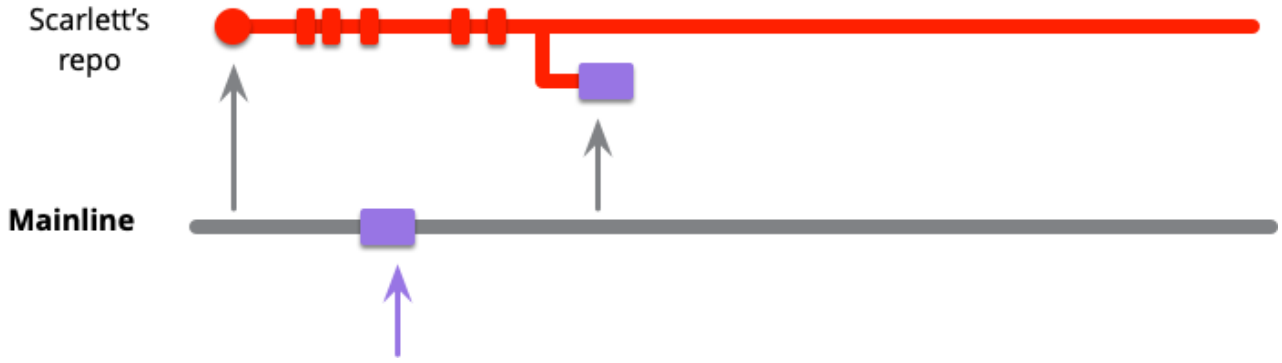
I'll walk through an example of how this works. A developer, who I'll call Scarlett, starts some work by cloning the mainline into her own repository. With git, if she doesn't already have a clone of the central repository, she would clone it and checkout the

master branch. If she already has the clone, she would pull from mainline into her local master. She can then work locally, making commits into her local master.

While she's working, her colleague Violet pushes some changes onto mainline. As she's working in her own codeline, Scarlett can be oblivious to those changes while she works on her own task.



At some point, she reaches a point where she wants to integrate. The first part of this is to fetch the current state of mainline into her local master branch, this will pull in Violet's changes. As she's working on local master, the commits will show on origin/master as a separate codeline.

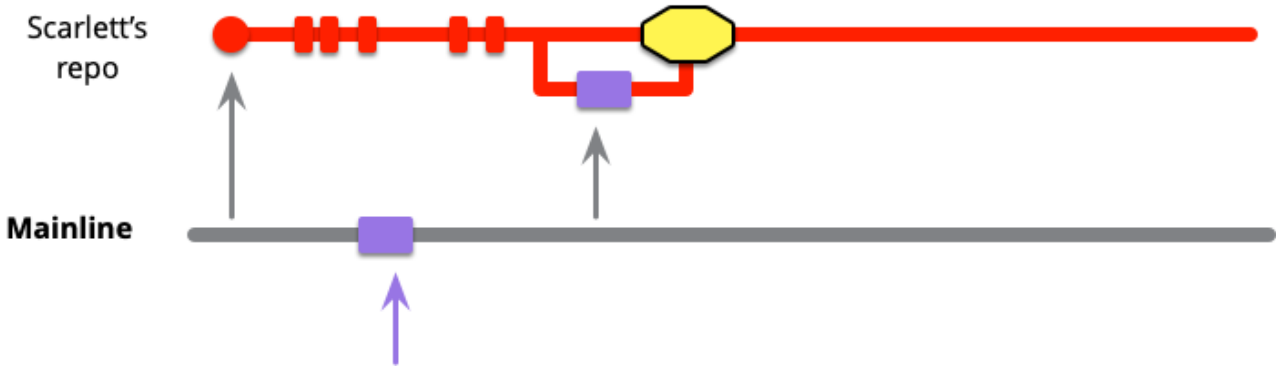


Now she needs to combine her changes with those of Violet. Some teams like to do this by merging, others by rebasing. In general people use the word “merge” whenever they talk about bringing branches together, whether they actually use a git merge or rebase

operation. I'll follow that usage, so unless I'm actually discussing the differences between merging and rebasing consider “merge” to be the logical task that can be implemented with either.

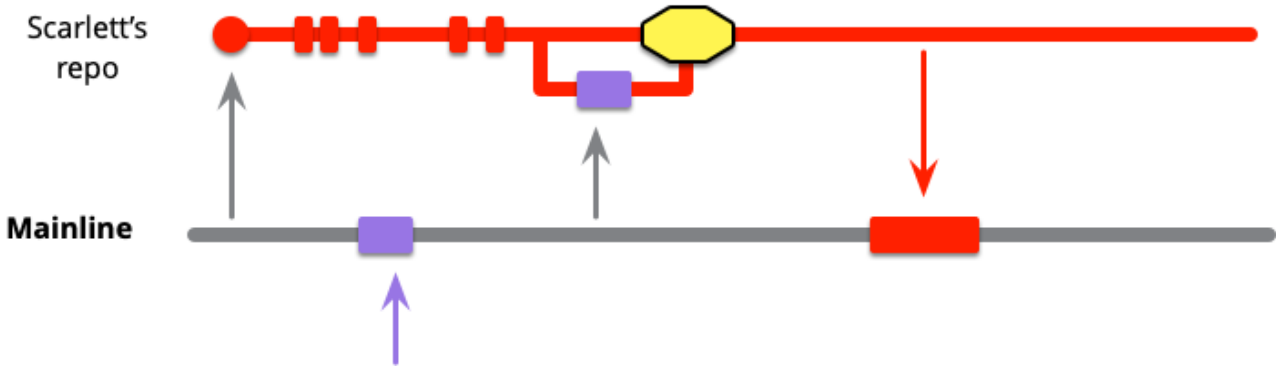
> There's a whole other discussion on whether to use vanilla merges, use or avoid fast-forward merges, or use rebasing. That's outside the scope of this article, although if people send me enough Tripel Karmeliet, I might write an article on that issue. After all, *quid-pro-quos* are all the rage these days.

If Scarlett is fortunate, merging in Violet's code will be a clean merge, if not she'll have some conflicts to deal with. These may be textual conflicts, most of which the source control system can handle automatically. But semantic conflicts are much harder to deal with, and this is where Self Testing Code is very handy. (Since conflicts can generate a considerable amount of work, and always introduce the risk of a lot of work, I mark them with an alarming lump of yellow.)



At this point, Scarlett needs to verify that the merged code satisfies the health standards of the mainline (assuming mainline is a Healthy Branch). This usually means building the code and running whatever tests form the commit suite for mainline. She needs to do this even if it's a clean merge, because even a clean merge can hide semantic conflicts. Any failures in the commit suite should be purely due to the merge, since both merge parents should be green. Knowing this should help her track down the problem as she can look at the diffs for clues.

With this build and test, she has successfully pulled mainline into her codeline, but - *and this is both important and often overlooked* - she hasn't yet finished integrating with mainline. To finish integrating she must push her changes into the mainline. Unless she does this, everyone else on the team will be isolated from her changes - essentially not integrating. Integration is both a pull and a push - only once Scarlett has pushed is her work integrated with the rest of the project.



Many teams these days require a code review step before commit is added to mainline - a pattern I call Pre-Integration Review and will discuss later.

Occasionally someone else will integrate with mainline before Scarlett can do her push. In which case she has to pull and merge again. Usually this is only an occasional issue and can be sorted out without any further coordination. I have seen teams with long builds use an integration baton, so that only the developer holding the baton could integrate. But I haven't heard so much of that in recent years as build times improve.

When to use it

As the name suggests, I can only use mainline integration if we're also using mainline on our product.

One alternative to using mainline integration is to just pull from mainline, merging those changes into the personal development branch. This can be useful - pulling can at least alert Scarlett to changes other people have integrated, and detect conflicts between her work and mainline. But until Scarlett pushes, Violet won't be able to detect any conflicts between what she's working on and Scarlett's changes.

When people use the word “integrate”, they often miss this important point. It's common to hear someone say they are integrating the mainline into their branch when they are merely pulling. I've learned to be wary of that, and probe further to check to see if they mean just a pull or a proper mainline integration. The consequences of the two are very different, so it's important not to confuse the terms.

Another alternative is when Scarlett is in the middle of doing some work that isn't ready for full integration with the rest of the team, but it overlaps with Violet and she wants to share it with her. In that case they can open a Collaboration Branch.

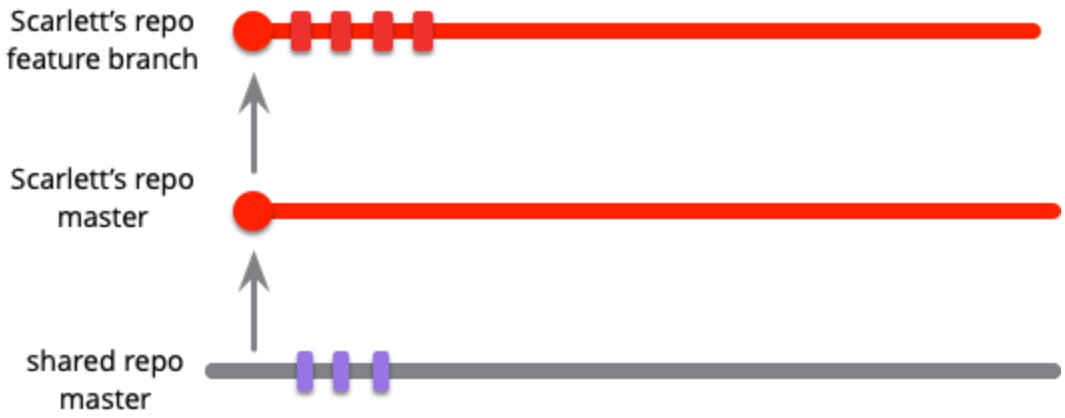
✣ ✣ ✣

✢ Feature Branching ✢

> *Put all work for a feature on its own branch, integrate into mainline when the feature is complete.*

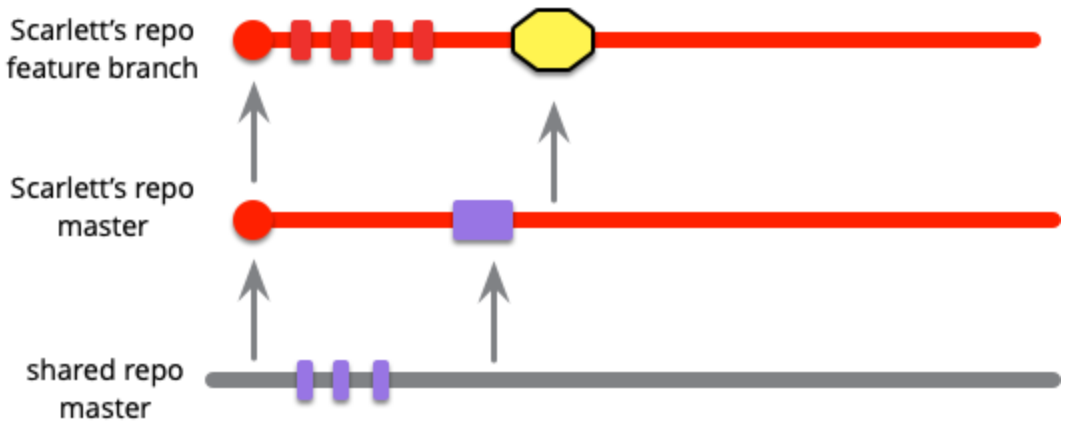
With feature branching, developers open a branch when they begin work on a feature, continue working on that feature until they are done, and then integrate with mainline.

For example, let's follow Scarlett. She would pick up the feature to add collection of local sales taxes to their website. She begins with the current stable version of the product, she'll pull mainline into her local repository and then create a new branch starting at the tip of the current mainline. She works on the feature for as long as it takes, making a series of commits to that local branch.



She might push that branch to the project repo so that others may look at her changes.

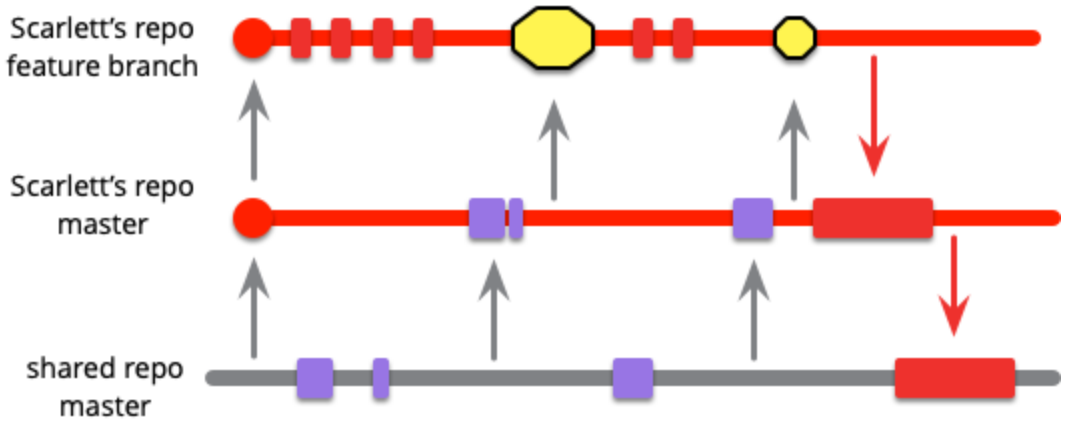
While she's working, other commits are landing on mainline. So from time to time she may pull from mainline so she can tell if any changes there are likely to impact her feature.



Note this isn't integration as I described above, since she didn't push back to mainline. At this point only she is seeing her work, others don't.

> Some teams like to ensure all code, whether integrated or not, is kept in the central repository. In this case Scarlett would push her feature branch into the central repository. This would also allow other team members to see what she's working on, even if it's not integrated into other people's work yet.

When she's done working on the feature, she'll then perform Mainline Integration to incorporate the feature into the product.



If Scarlett works on more than one feature at the same time, she'll open a separate branch for each one.

When to use it

Feature Branching is a popular pattern in the industry today. To talk about when to use it, I need to introduce its principal alternative - Continuous Integration. But first I need to talk about the role of integration frequency.

✢ ✢ ✢

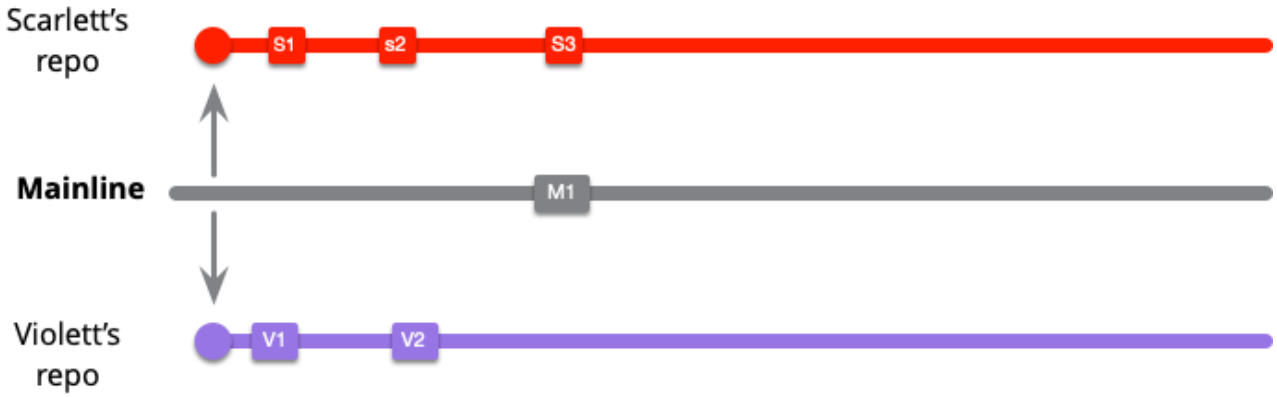
Integration Frequency

How often we do integration has a remarkably powerful effect on how a team operates. Research from the State Of Dev Ops Report indicated that elite development teams integrate notably more often than low performers - an observation that matches my experience and the experiences of so many of my industry peers. I'll illustrate how this plays out by considering two examples of integration frequency starring Scarlett and Violet.

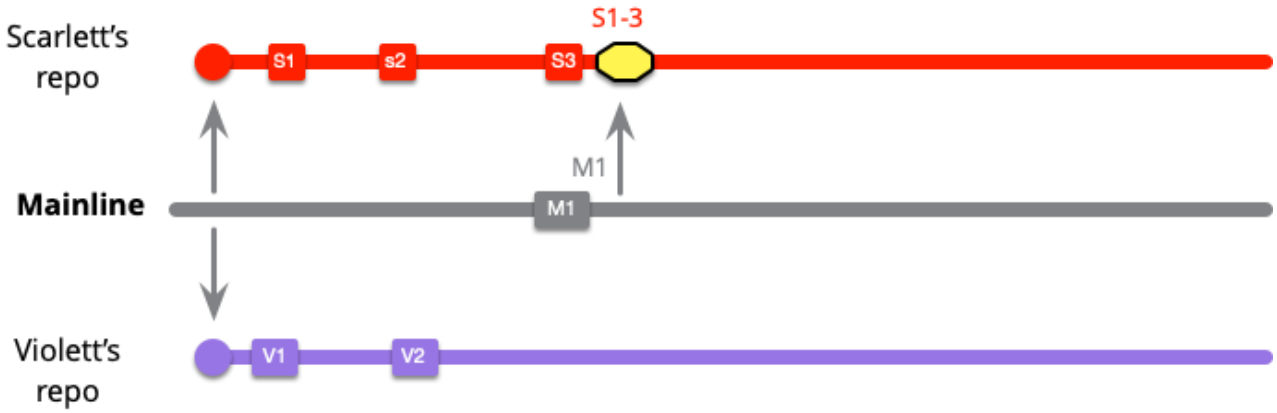
Low-Frequency Integration

I'll start with the low-frequency case. Here, our two heroes begin an episode of work by cloning the mainline into their branches, then doing a couple of local commits that they don't want to push yet.

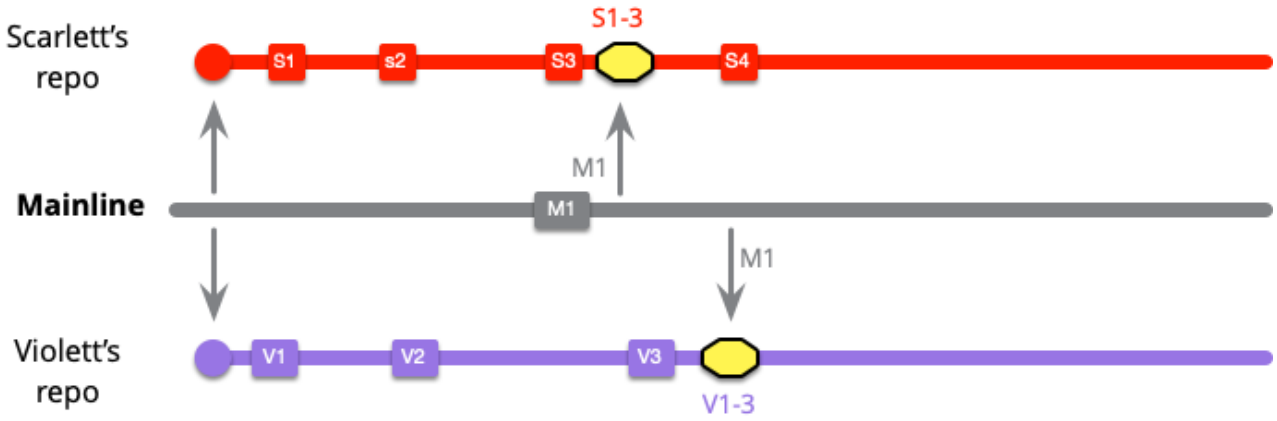
As they work, someone else puts a commit onto mainline. (I can't quickly come up with another person's name that's a color - maybe grayham?)



This team works by keeping a healthy branch and pulling from mainline after each commit. Scarlett didn't have anything to pull with her first two commits as mainline was unchanged, but now needs to pull M1.

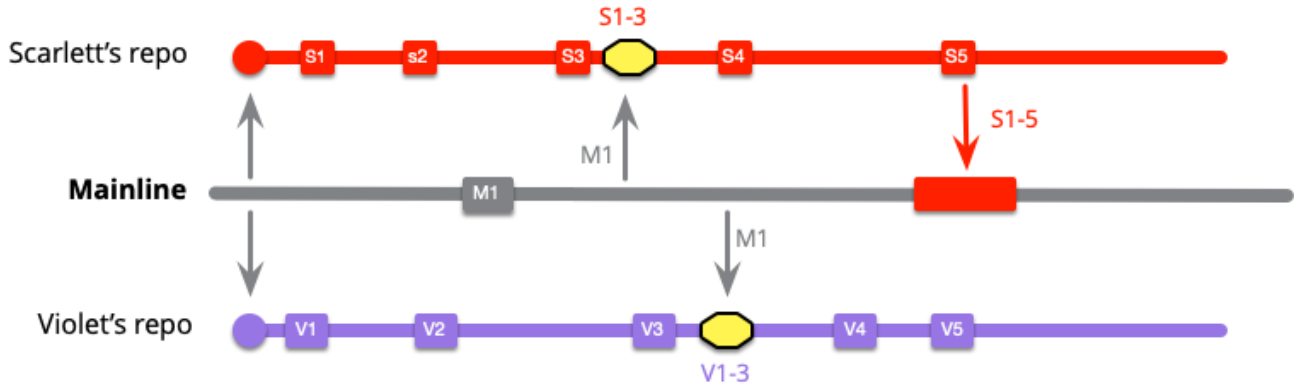


I've marked the merge with the yellow box. This one merges commits S1..3 with M1. Soon Violet needs to do the same thing.

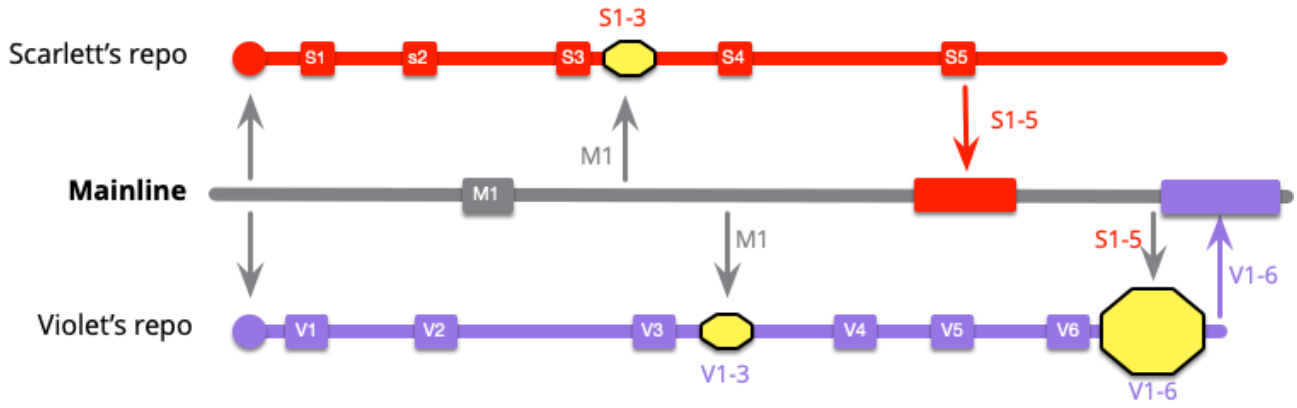


At this point both developers are up to date with mainline, but they haven't integrated since they are both isolated from each other. Scarlett is unaware of any changes Violet has made in V1..3.

Scarlett makes a couple more local commits then is ready to do mainline integration. This is an easy push for her, since she pulled M1 earlier.



Violet, however has a more complicated exercise. When she does mainline integration she now has to integrate S1..5 with V1..6.



I have scientifically calculated the sizes of the merges based on how many commits are involved. But even if you ignore the tongue-shaped bulge in my cheek, you'll appreciate that Violet's merge is the mostly likely to be difficult.

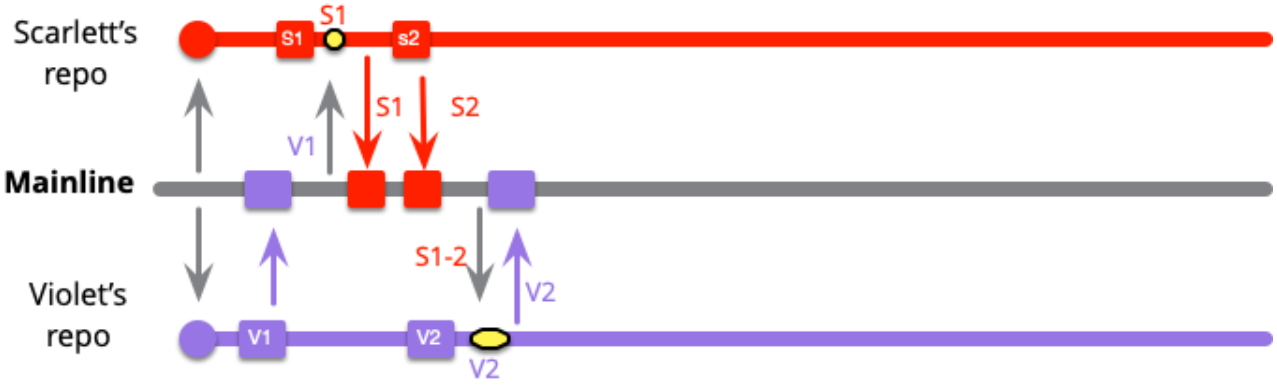
High-Frequency Integration

In the previous example, our two colorful developers integrated after a handful of local commits. Let's see what happens if they do mainline integration after every local commit.

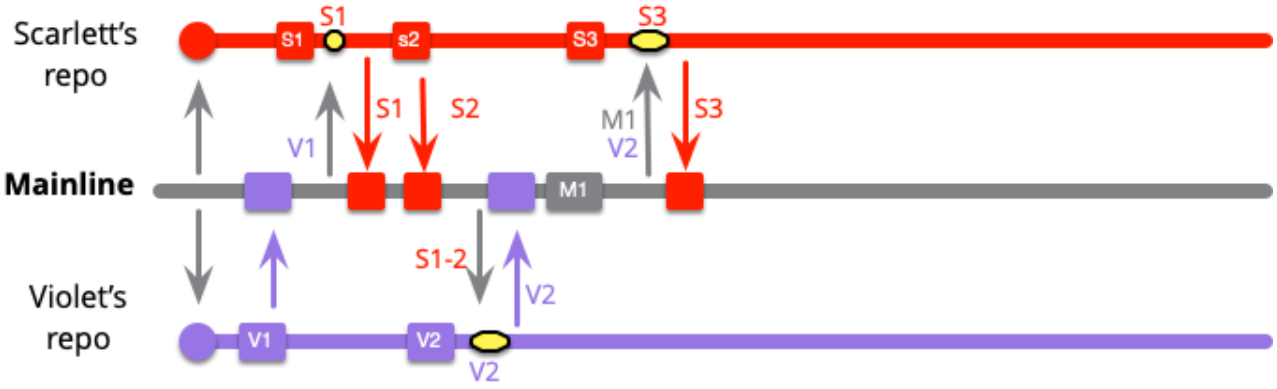
The first change is apparent with Violet's very first commit, as she integrates right away. Since mainline hasn't changed, this is a simple push.

Scarlett's first commit also has mainline integration, but because Violet got there first, she needs do a merge. But since she's only merging V1 with S1, the merge is small.

Scarlett's next integration is a simple push which means Violet's next commit will also require merging with Scarlett's latest two commits. However it's still a pretty small merge, one of Violet's and two of Scarlett's.



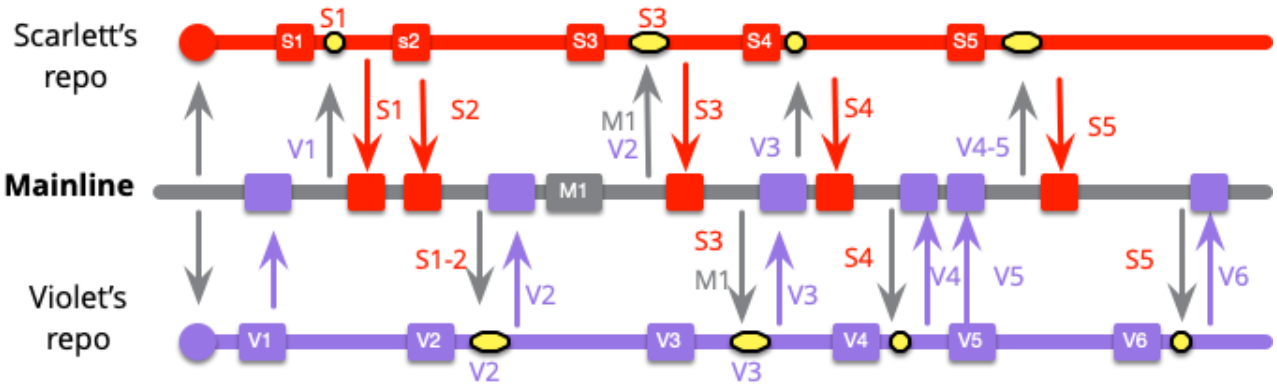
When the external push to mainline appears, it gets picked up in the usual rhythm of Scarlett and Violet's integrations.



While it's similar to what happened before, the integrations are smaller. Scarlett only has to integrate S3 with M1 this time, because S1 and S2 were already on mainline. This

means that Grayham would have had to integrate whatever was already on mainline (S1..2, V1..2) before pushing M1.

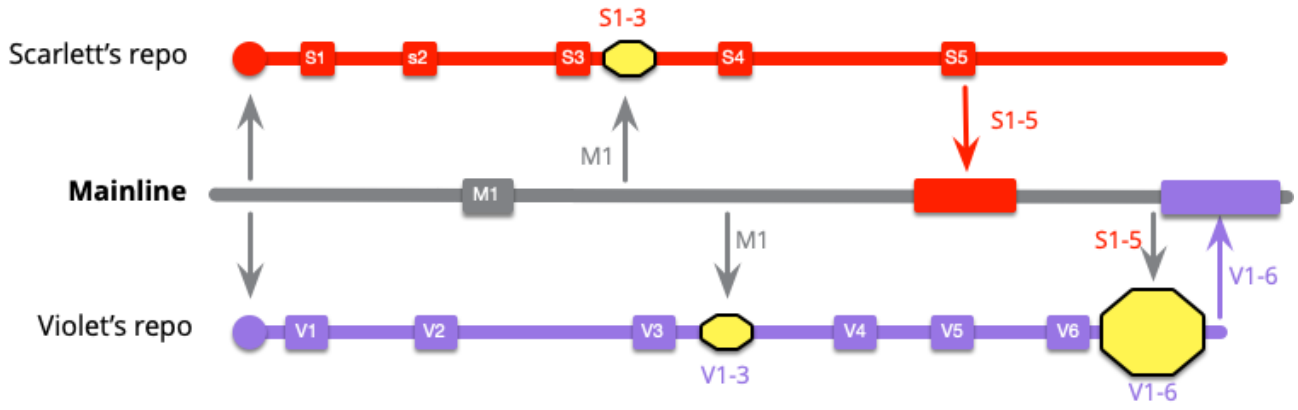
The developers continue with their remaining work, integrating with each commit.



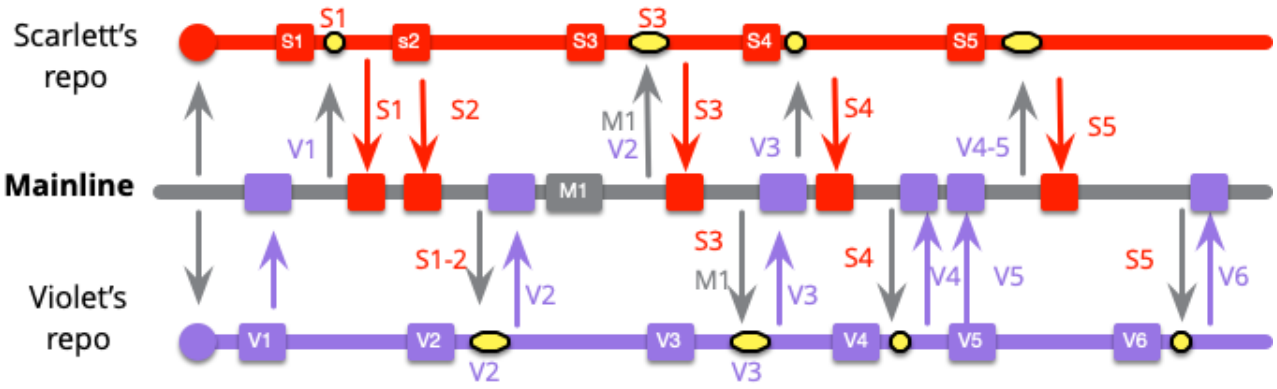
Comparing integration frequencies

Let's look again at the two overall pictures

Low Frequency



High Frequency



Integration Fear

When teams get a couple of bad merge experiences, they tend to be wary of doing integration. This can easily turn into a positive feedback loop - which like many positive feedback loops, has very negative consequences.

The most obvious consequence is that the team does integration less frequently, which leads to more nasty merge incidents, which leads to less frequent integrations... and so on.

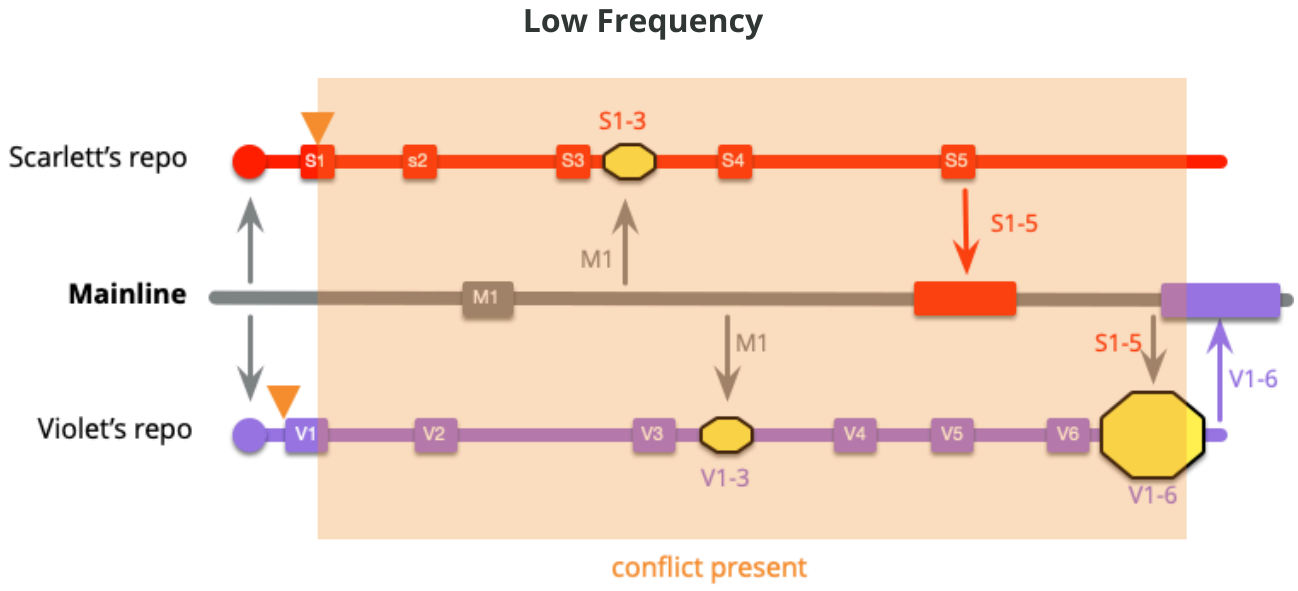
A more subtle problem is that teams stop doing things that they think will make integration harder. In particular this makes them resist refactoring. But reducing refactoring leads to the code base getting increasingly unhealthy, difficult to understand and modify, and thus slowing down the teams feature delivery. Since it takes longer to complete features, that further decreases integration frequency, contributing to the debilitating positive feedback loop.

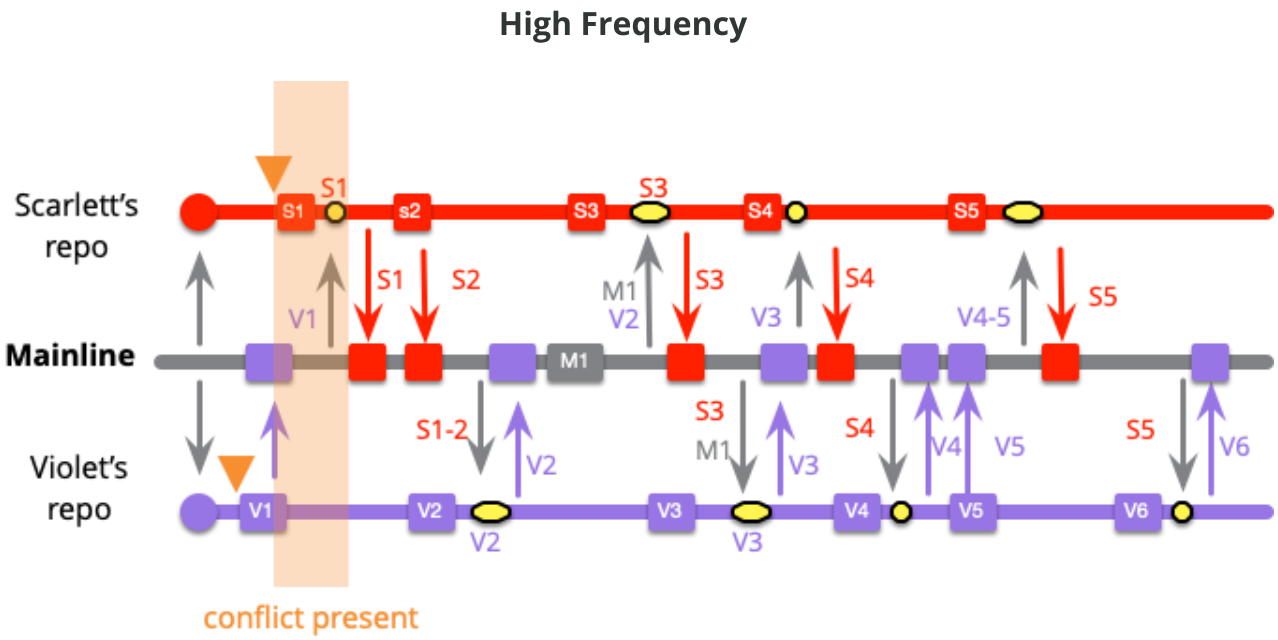
The counter-intuitive answer to this is captured by the slogan - "if it hurts... do it more often"

There are two very obvious differences here. Firstly the high-frequency integration, as the name implies, has a lot more integrations - twice as many just in this toy example. But more importantly these integrations are much smaller than those in the low-frequency case. Smaller integrations mean less work, since there's less code changes that might hold up conflicts. But more importantly than less work, it's also less risk. The problem with big merges is not so much the work involved with them, it's the uncertainty of that work. Most of the time even big merges go smoothly, but occasionally they go very, *very*, badly. That occasional pain ends up being worse than a regular pain. If I compare spending an extra ten minutes per integration with a 1 out of fifty chance of spending 6 hours fixing an integration - which do I prefer? If I just look at the effort, then the 1-in-50 is better, since it's 6 hours rather 8 hours and twenty minutes. But the uncertainty makes the 1-in-50 case feel much worse, an uncertainly that leads to integration fear.

Let's look the difference between these frequencies from another perspective. What happens if Scarlett and Violet develop a conflict in their very first commits? When do

they detect the conflict has occurred? In the low-frequency case, they don't detect it until Violet's final merge, because that's the first time S1 and V1 are put together. But in the high-frequency case, they are detected at Scarlett's very first merge.





Frequent integration increases the frequency of merges but reduces their complexity and risk. Frequent integration also alerts teams to conflicts much more quickly. These two things are connected, of course. Nasty merges are usually the result of a conflict that's been latent in the team's work, surfacing only when integration happens.

Perhaps Violet was looking at a billing calculation and saw that it included appraising tax where the author had assumed a particular taxation mechanism. Her feature requires different treatments to tax, so the direct route was to take the tax out of the billing calculation and do it as a separate function later. The billing calculation was only

called in a couple of places, so it's easy to use Move Statements to Callers - and the result makes more sense for the future evolution of the program. Scarlett, however, didn't know Violet was doing this and wrote her feature assuming the billing function took care of tax.

Self Testing Code is our life-saver here. If we have a strong test suite, using it as part of the healthy branch will spot the conflict so there's far less chance of a bug making its way into production. But even with a strong test suite acting as a gatekeeper to mainline, large integrations make life harder. The more code we have to integrate, the harder it is to find the bug. We also have a higher chance of multiple, interfering bugs, that are extra-difficult to understand. Not just do we have less to look at with smaller commits, we can also use Diff Debugging to help narrow down which change introduced the problem.

What a lot of people don't realize is that a source control system is a communication tool. It allows Scarlett to see what other people on the team are doing. With frequent integrations, not just is she alerted right away when there are conflicts, she's also more aware of what everyone is up to, and how the codebase is evolving. We're less like individuals hacking away independently and more like a team working together.

> Increasing the frequency of integration is an important reason to reduce the size of features, but there are other advantages too. The smaller the feature, the quicker it is to build, quicker to get into production, quicker to start delivering its value. Furthermore smaller features reduces the feedback time, allowing a team to make better feature decisions as they learn more about their customers.

✢ Continuous Integration ✢

> *Developers do mainline integration as soon as they have a healthy commit they can share, usually less than a day's work*

Once a team has experienced that high-frequency integration is both more efficient and less stressful, that natural question to ask is “how frequently can we go?” Feature branching implies a lower bound to the size of a change-set - you can't be smaller than a cohesive feature.

> For more details on how to do Continuous Integration effectively, take a look at the my detailed article. For even more details, consult the book by Paul Duvall, Steve Matyas, and Andrew Glover. Paul Hammant

maintains a website full of techniques for continuous integration at trunkbaseddevelopment.com

Continuous Integration applies a different trigger for integration - you integrate whenever you've made a hunk of progress on the feature and your branch is still healthy. There's no expectation that the feature be complete, just that there's been a worthwhile amount of changes to the codebase. The rule of thumb is that "everyone commits to the mainline every day", or more precisely: *you should never have more than a day's work sitting unintegrated in your local repository*. In practice, most practitioners of Continuous Integration integrate many times a day, happy to integrate an hour's worth of work or less.

Developers using Continuous Integration need to get used to the idea of reaching frequent integration points with a partially built feature. They need to consider how to do this without exposing a partially built feature in the running system. Often this is easy: if I'm implementing a discount algorithm that relies on a coupon code, and that code isn't in the valid list yet, then my code isn't going to get called even if it is production. Similarly if I'm adding a feature that asks an insurance claimant if they are a smoker, I can build and test the logic behind the code and ensure it isn't used in production by leaving the UI that asks the question until the last day of building the feature. Hiding a partially built feature by hooking up a Keystone Interface last is often an effective technique.

Continuous Integration and Trunk-Based Development

When Thoughtworks started using Continuous Integration in 2000, we wrote CruiseControl, a daemon that automatically built the software product after each commit to mainline. Since then, many such tools (e.g. Jenkins, Team City, Travis CI, Circle CI, Bamboo, and many others) have been developed. But most organizations that use these tools use them to automatically build feature branches on commit - which, while useful, means they don't actually practice Continuous Integration. (A better name for them would be Continuous Build tools.)

Because of this Semantic Diffusion, some people started to use the term "Trunk-Based Development" instead of "Continuous Integration". (Some people do make subtle distinctions between the two terms, but there isn't a consistent usage.) Although I'm usually a descriptivist when it comes to language, I prefer to use "Continuous Integration". Partly this is because I don't think trying to continually come up with new terms is a viable way to fight semantic diffusion. Perhaps the main reason, however, is because I think changing terminology rudely erases the contribution of the early Extreme Programming pioneers, in particular Kent Beck, who coined and clearly defined the practice of Continuous Integration in the 1990s.

If there's no way to easily hide the partial feature, we can use feature flags. As well as hiding a partially built feature, such flags also allow the feature to be selectively revealed to a subset of users - often handy for a slow roll-out of a new feature.

Integrating part-built features particularly concerns those who worry about having buggy code in mainline. Consequently, those who use Continuous Integration also need Self Testing Code, so that there's confidence that having partially built features in mainline doesn't increase the chance of bugs. With this approach, developers write tests for the partially built features as they are writing that feature code and commit both feature code and tests into mainline together (perhaps using Test Driven Development).

In terms of a local repo, most people who use Continuous Integration don't bother with a separate local branch to work on. It's usually straightforward to commit to the local master and perform mainline integration when done. However it's perfectly fine to open a feature branch and do the work there, if developers prefer it, integrating back into the local master and mainline at frequent intervals. The difference between feature branching and continuous integration isn't whether or not there's a feature branch, but when developers integrate with mainline.

When to use it

Continuous Integration is an alternative to Feature Branching. The trade-offs between the two are sufficiently involved to deserve their own section of this article, and now is the time to tackle it.

Comparing Feature Branching and Continuous Integration

Feature Branching appears to be the most common branching strategy in the industry at the moment, but there is a vocal group of practitioners who argue that Continuous Integration is usually a superior approach. The key advantage that Continuous Integration provides is that it supports higher, often a much higher, integration frequency.

The difference in integration frequency depends on how small a team is able to make its features. If a team's features can all be done in less than a day, then they can perform both Feature Branching and Continuous Integration. But most teams have

longer feature lengths than this - and the greater the feature length, the greater the difference between the two patterns.

As I've indicated already, higher frequency of integration leads to less involved integration and less fear of integration. This is often a difficult thing to communicate. If you've lived in a world of integrating every few weeks or months, integration is likely to be a very fraught activity. It can be very hard to believe that it's something that can be done many times a day. But integration is one of those things where Frequency Reduces Difficulty. It's a counter-intuitive notion - “if it hurts - do it more often”. But the smaller the integrations, the less likely they are to turn into an epic merge of misery and despair. With feature branching, this argues for smaller features: days not weeks (and months are right out).

Continuous Integration allows a team to get the benefits of high-frequency integration, while decoupling feature length from integration frequency. If a team prefers feature lengths of a week or two, Continuous Integration allows them to do this while still getting all the benefits of the highest integration frequency. Merges are smaller, requiring less work to deal with. More importantly, as I explained above, doing merges more frequently reduces the risk of a nasty merge, which both cuts out the bad surprises that this brings and reduces the overall fear of merging. If conflicts arise in the code, high-frequency integration discovers them quickly, before they lead to those nasty integration problems. These benefits are strong enough that there are teams that have features that only take a couple of days that still do Continuous Integration.

The clear downside of Continuous Integration is that it lacks the closure of that climactic integration to mainline. Not just is this a lost celebration, it's a risk if a team isn't good at keeping a Healthy Branch. Keeping all the commits of a feature together also makes it possible to make a late decision on whether to include a feature in an upcoming release. While feature flags allow features to be switched on or off from the users' perspective, the code for the feature is still in the product. Concerns about this are often overblown, after all code doesn't weigh anything, but it does mean that teams who want to do Continuous Integration must develop a strong testing regimen so they can be confident that mainline remains healthy even with many integrations a day. Some teams find this skill difficult to imagine, but others find it to be both possible and liberating. This prerequisite does mean that Feature Branching is better for teams that don't force a Healthy Branch and require release branches to stabilize code before release.

While the size and uncertainty of merges is the most obvious problem with Feature Branching, the biggest problem with it may be that it can deter refactoring. Refactoring is at its most effective when it's done regularly and with little friction. Refactoring will introduce conflicts, if these conflicts aren't spotted and resolved quickly, merging gets fraught. Refactoring thus works best with a high frequency of integration, so it's no surprise that it became popular as part of Extreme Programming which also has Continuous Integration as one of the original practices. Feature Branching also discourages developers from making changes that aren't seen as part of the feature being built, which undermines the ability of refactoring to steadily improve a code base.

> *We found that having branches or forks with very short lifetimes (less than a day) before being merged into trunk, and less than three active branches in total, are important aspects of continuous delivery, and all contribute to higher performance. So does merging code into trunk or master on a daily basis.*
>
> *-- State of DevOps Report 2016*

When I come across scientific studies of software development practices, I usually remain unconvinced due to serious problems with their methodology. One exception is the State Of Dev Ops Report, which has developed a metric of software delivery performance, which they correlated to a wider measure of organizational performance, which in turn correlates to business metrics such as return on investment and profitability. In 2016, they first assessed Continuous Integration and found it contributed to higher software development performance, a finding that has been repeated in every survey since.

Using Continuous Integration doesn't remove the other advantages of keeping features small. Frequently releasing small features provides a rapid feedback cycle which can do wonders for improving a product. Many teams that use Continuous Integration also strive to build thin slices of product and release new features as frequently as they can.

Feature Branching

- ✓ All the code in a feature can be assessed for quality as a unit
- ✓ Feature code only added to product when feature is complete
- ✗ Less frequent merges

Continuous Integration

- ✓ Supports higher frequency integration than feature length
- ✓ Reduced time to find conflicts

✓ Smaller merges
✓ Encourages refactoring
✗ Requires commitment to healthy branches (and thus self-testing code)
✓ Scientific evidence that it contributes to higher software delivery performance

Feature Branching and Open Source

Many people ascribe the popularity of Feature Branching to GitHub and the pull-request model which originated in open-source development. Given that, it's worthwhile to understand the very different contexts that exist between open-source work and much commercial software development. Open-source projects are structured in many different ways, but a common structure is that of a one person, or a small group, that acts as the maintainer doing most of the programming. The maintainer works with a larger group of programmers who are contributors. The maintainer usually doesn't know the contributors, so has no sense of the quality of the code they contribute. The maintainer also has little certainty about how much time the contributors will actually put into the work, let alone how effective they are at getting things done.

> **Pull Requests**
>
> I see Pull Requests as a mechanism designed to support the combination of Feature Branching and Pre-Integration Review. To decide if and how to use pull requests, I'd first consider the role of those underlying patterns in the team's workflow

In this context, Feature Branching makes a whole lot of sense. If someone is going to add a feature, small or large, and I have no idea when (or if) it's going to be finished, then it makes sense for me to wait till it's done before integrating. It's also more important to be able to review the code, to ensure it passes whatever quality bar I have for my code base.

But many commercial software teams have a very different working context. There's a full-time team of people, all of which commit substantial, usually full-time, to the software. The leaders of the project know these people well (other than when they just start) and can have a reliable expectation of code quality and ability to deliver. Since they are paid employees, the leaders also have greater control about time put into the project and on such things as coding standards and group habits.

Given this very different context, it should be clear that a branching strategy for such commercial teams need not be the same as that which operates in the open-source

world. Continuous Integration is near-impossible fit for occasional contributors to open-source work, but is a realistic alternative for commercial work. Teams should not assume that what works for an open-source environment is automatically correct for their different context.

✢ Pre-Integration Review ✢

Every commit to mainline is peer-reviewed before the commit is accepted.

Code review has long been encouraged as a way of improving code quality, improving modularity, readability, and removing defects. Despite this, commercial organizations often found it difficult to fit into software development workflows. The open-source world, however, widely adopted the idea that contributions to a project should be reviewed before accepting them onto the project's mainline, and this approach has spread widely through development organizations in recent years, particularly in Silicon Valley. A workflow like this fits particularly well with the GitHub mechanism of pull-requests.

A workflow like this begins when Scarlett finishes a hunk of work that she wishes to integrate. As she does Mainline Integration (assuming her team practices that) once she has a successful build, but before she pushes to mainline, she sends her commit out for review. Some other member of the team, say Violet, then does a code review on the commit. If she has problems with the commit, she makes some comments and there's some back-and-forth until both Scarlett and Violet are happy. Only once they are done is the commit placed on mainline.

Pre-integration reviews grew in popularity with open source, where they fit very well with the organizational model of committed maintainers and occasional contributors. They allow the maintainer to keep a close eye on any contributions. They also mesh well with Feature Branching, since a completed feature marks a clear point to do a code review like this. If you're not certain that a contributor is going to complete a feature, why review their partial work? Better to wait until the feature is complete. The practice also spread widely at the larger internet firms, Google and Facebook both build special tooling to help make this work smoothly.

Developing the discipline for timely Pre-Integration Reviews is important. If a developer finishes some work, and goes onto something else for a couple of days, then

that work is no longer on the top of their mind when the review comments come back. This is frustrating with a completed feature, but it's much worse for a partially completed feature, where it may be difficult to make further progress until the review is confirmed. In principle, it's possible to do Continuous Integration with Pre-Integration Reviews, and indeed it's possible in practice too - Google follows this approach. But although this is possible, it's hard, and relatively rare. Pre-Integration Reviews and Feature Branching are the more common combination.

When to use it

> *Conflating OSS and private software development team needs is like the original sin of current software development rituals*
>
> -- *Camille Fournier*

Although Pre-Integration Reviews have become a popular practice over the last decade, there are downsides and alternatives. Even when done well, Pre-Integration Reviews always introduces some latency into the integration process, encouraging a lower integration frequency. Pair Programming offers a continuous code review process, with a faster feedback cycle than waiting for a code review. (Like Continuous Integration and Refactoring, it's one of the original practices of Extreme Programming).

Many teams that use pre-integration reviews don't do them quickly enough. The valuable feedback that they can offer then comes too late to be useful. At that point there's an awkward choice between a lot of rework, or accepting something that may work, but undermines the quality of the code-base.

Code review isn't confined to before the code hits the mainline. Many tech leaders find it useful to review code after a commit, catching up with developers when they see concerns. A culture of refactoring is valuable here. Done well this sets up a community where everyone on the team is regularly reviewing the code base and fixing problems that they see - a practice I call Refinement Code Review

The trade-offs around pre-integration reviews rest primarily on the social structure of the team. As I've already mentioned, open-source projects commonly have a structure of a few trusted maintainers and many untrusted contributors. Commercial teams are frequently all full-time, but may have a similar structure. The project leader (like a maintainer) trusts a small (perhaps singular) group of maintainers, and is wary of code contributed from the rest of the team. Team members may be allocated to multiple

projects at once, making them much more like open-source contributors. If such a social structure exists, then Pre-Integration Reviews and Feature Branching make a great deal of sense. But a team with a higher degree of trust often finds other mechanisms keep code quality high without adding friction to the integration process.

So, while pre-integration reviews can be a valuable practice, it's by no means a necessary route to a healthy code base, particularly if you're looking to grow a well-balanced team that isn't overly dependent on its initial leader.

✢ ✢ ✢

Integration Friction

> *Pull requests add overhead to cope with low-trust situations, e.g. to allow people you don't know to offer contributions to your project.*
>
> *Imposing pull requests on devs in your own team is like making your family go through an airport security checkpoint to enter your home.*
>
> -- *Kief Morris*

One of the problems of Pre-Integration Reviews, is that it often makes it more of a hassle to integrate. This is an example of **integration friction** - activities that make integration take time or be an effort to do. The more integration friction there is, the more developers are inclined to lower the frequency of integration. Imagine some (dysfunctional) organization that insists that all commits to mainline need a form that takes half-an-hour to fill in. Such a regime discourages people from integrating frequently. Whatever your attitude is to Feature Branching and Continuous Integration, it's valuable to examine anything that adds this kind of friction. Unless it clearly adds value, any such friction should be removed.

Manual process are a common source of friction here, particularly if it involves coordination with separate organizations. This kind of friction can often be reduced by using automated processes, improving developer educations (to remove the need), and pushing steps to later steps of a Deployment Pipeline or QA in production. You can find more ideas for eliminating this kind of friction in material on continuous integration and continuous delivery. This sort of friction also crops up in the path to production, with the same difficulties and treatments.

One of the things that makes people reluctant to consider continuous integration is if they've only worked in environments with a high degree of integration friction. If it takes an hour to do an integration, then it's clearly absurd to do it several times a day. Joining a team where integration is a non-event, that someone can dash off in a few minutes feels like a different world. I suspect much of the argument about the merits of Feature Branching and Continuous Integration is muddied because people haven't experienced both of these worlds, and thus can't fully understand both points of view.

Cultural factors influence integration friction - in particular the trust between members of a team. If I'm a team leader, and I don't trust my colleagues to do a decent job, then I'm likely to want to prevent commits that damage the codebase. Naturally this is one of the drivers for Pre-Integration Reviews. But if I'm on a team where I trust the judgment of my colleagues, I'm likely to be more comfortable with a post-commit review, or cutting out the reviews entirely and rely on regular refinement reviews to clean up any problems. My gain in this environment is removing the friction that pre-commit reviews introduce, thus encouraging a higher-frequency of integration. Often team trust is the most important factor in the Feature Branch versus Continuous Integration argument.

An interesting approach that retains pre-integration review when needed, yet encourages a path of less friction is Rouan Wilsenach's Ship/Show/Ask. This classifies changes as either Ship (integrate into mainline), Show (integrate into mainline, but open a pull request to communicate and discuss the change), or Ask (open a pull request for pre-integration review).

The Importance of Modularity

Most people who care about software architecture stress the importance of modularity to a well-behaved system. If I'm faced with making a small change to a system with poor modularity, I have to understand nearly all of it, since even a small change can ripple through so many parts of the codebase. With good modularity, however, I only need to understand the code in one or two modules, the interfaces to a few more, and can ignore the rest. This ability to reduce the effort of understanding I need is why it's worth putting so much effort on modularity as a system grows.

Modularity also impacts integration. If a system has good modules then most of the time Scarlett and Violet will be working in well-separated parts of the code base, and their changes won't cause conflicts. Good modularity also enhances techniques like Keystone Interface and Branch By Abstraction to avoid the need for the isolation that

branches provide. Often teams are forced to use source branching because the lack of modularity starves them of other options.

> *Feature Branching is a poor man's modular architecture, instead of building systems with the ability to easy swap in and out features at runtime/deploytime they couple themselves to the source control providing this mechanism through manual merging.*
>
> *-- Dan Bodart*

The support goes in both directions. Despite many attempts, it remains extremely difficult to build a good modular architecture before we start programming. To achieve modularity we need to constantly watch our system as it grows and tend it in a more modular direction. Refactoring is the key to achieving this, and refactoring requires high-frequency integration. Modularity and rapid integration thus support each other in a healthy codebase.

Which is all to say that modularity, while hard to achieve, is worth the effort. The effort involves good development practices, learning about design patterns, and learning from experience with the code base. Messy merges shouldn't just be closed off with an understandable desire to forget about them - instead ask why the merge is messy. These answers will often be an important clue to how modularity can be improved, improving the health of the code base, and thus enhancing the productivity of the team.

Personal Thoughts on Integration Patterns

My aim as a writer isn't to convince you to follow a particular path, but instead to inform you about the factors that you should consider, as *you* decide which path to follow. Despite that, I will add my opinion here on which I prefer in the patterns I've indicated earlier.

Overall I much prefer to work on a team that uses Continuous Integration. I recognize that context is key, and there are many circumstances where Continuous Integration isn't the best option - but my reaction is to do the work to change that context. I have this preference because I want to be in an environment where everyone can easily keep refactoring the codebase, improving its modularity, keeping it healthy - all to enable us to quickly respond to changing business needs.

These days I'm more of a writer than a developer, but I still choose to work at Thoughtworks, a company that's full of people who favor this way of working. This is because I believe this Extreme Programming style is one of the most effective ways we can develop software, and I want to observe teams further developing this approach to improve the effectiveness of our profession.

The path from mainline to production release

The mainline is an active branch, with regular drops of new and modified code. Keeping it healthy is important so that when people start new work, they are starting off a stable base. If it's healthy enough, you can also release code directly from mainline into production.

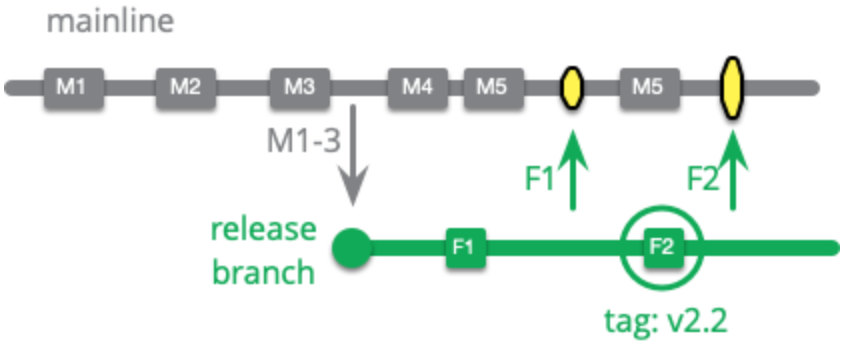
This philosophy of keeping the mainline in an always-releasable state is the central tenet of Continuous Delivery. To do this, there must be the determination and skills present to maintain mainline as a Healthy Branch, usually with Deployment Pipelines to support the intensive testing required.

Teams working this way can usually keep track of their releases by using tags on each released version. But teams that don't use continuous delivery need another approach.

✢ Release Branch ✢

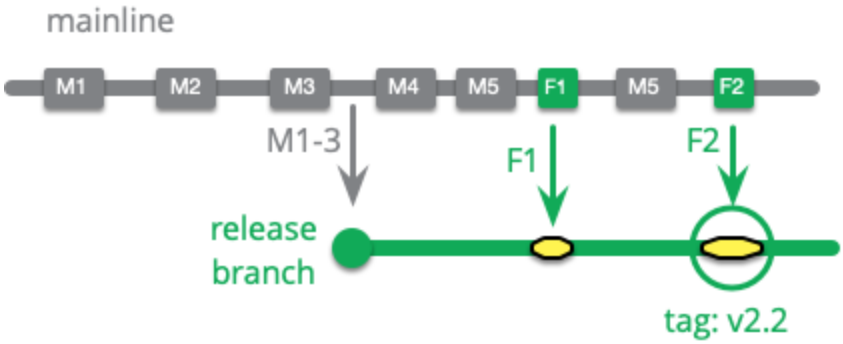
> *A branch that only accepts commits accepted to stabilize a version of the product ready for release.*

A typical release branch will copy from the current mainline, but not allow any new features to added to it. The main development team continues to add such features to the mainline, and these will be picked up in a future release. The developers working on the release focus solely on removing any defects that stop the release from being production-ready. Any fixes to these defects are created on the release branch and merged to mainline. Once there are no more faults to deal with, the branch is ready for production release.



Although the scope of work for the fixes on the release branch is (hopefully) smaller than new feature code, it gets increasingly difficult to merge them back into mainline as time goes on. Branches inevitably diverge, so as more commits modify mainline, it gets harder to merge the release branch into mainline.

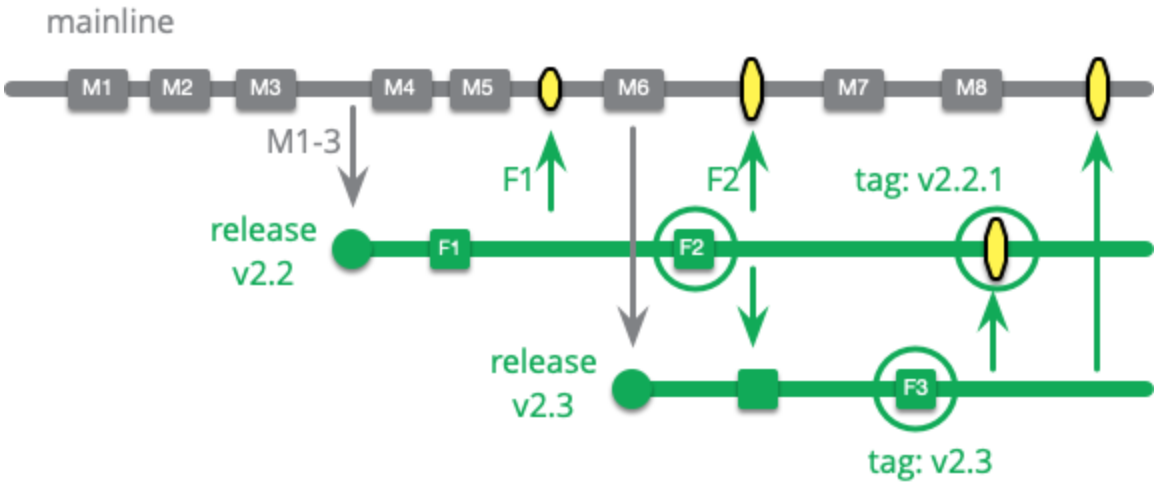
A problem with applying the commits to the release branch in this way is that it's too easy to neglect to copy them over to the mainline, particularly as it gets harder due to divergence. The resulting regression is very embarrassing. Consequently some people favor creating the commits on mainline, and only once they are working there to cherry-pick them into the release branch.



> A **cherry-pick** is when a commit is copied from one branch to another, but the branches aren't merged. That is, only the one commit is copied over, not the previous commits since the branch point. In this example, if I were to merge F1 into the release branch, then this would include M4 and M5. But a cherry-pick only takes F1. A cherry-pick may not cleanly apply to the release branch, as it may rely on changes done in M4 and M5.

The downside of writing release fixes on mainline is that many teams find it harder to do so, and frustrating to fix it one way on mainline and have to rework on the release branch before the release can occur. This is particularly true when there is schedule pressure to get the release out.

Teams that only have one version in production at a time will only need a single release branch, but some products will have many releases present in production use. Software that's run on customers' kit will only be upgraded when that customer wishes to. Many customers are reluctant to upgrade unless they have compelling new features, having been burned by upgrades that fail. Such customers, however, still want bug fixes, especially if they involve security issues. In this situation the development team keeps release branches open for each release that's still being used, and applies fixes to them as needed.



As development goes on it gets increasingly difficult to apply fixes to older releases, but that's often the cost of doing business. It can only be mitigated by encouraging customers to frequently upgrade to the latest version. Keeping the product stable is essential for this, once burned a customer will be reluctant to do an unnecessary upgrade again.

(Other terms I've heard for release branch include: "release-preparation branch", "stabilization branch", "candidate branch", and "hardening branch". But "release branch" seems to be the most common.)

When to use it

Release branches are a valuable tool when a team isn't able to keep their mainline in a healthy state. It allows a portion of the team to focus on the necessary bug fixes that are needed for it to be ready for production. Testers can pull the most stable recent candidate from the tip of this branch. Everyone can see what's been done to stabilize the product.

Despite the value of release branches, most of the best teams don't use this pattern for single-production products, because they don't need to. If the mainline is kept sufficiently healthy, then any commit to mainline can be released directly. In that case releases should be tagged with a publicly visible version and build number.

You might have noticed I stuck the clumsy adjective “single-production” into the previous paragraph. That's because this pattern becomes essential when teams need to manage multiple versions in production.

Release branches may also be handy when there is significant friction in the release process - such as a release committee that must approve all production releases. As Chris Oldwood puts it “In these cases the release branch acts more like a quarantine zone while the corporate cogs slowly turn”. In general, such friction should be removed from the release process as much as possible in a similar way that we need to remove integration friction. However there are some circumstances, such as mobile app stores, when this may not be possible. In many of these cases a tag is sufficient most of the time, and branch only opened if there's some essential change required to the source.

A release branch may also be an Environment Branch, subject to the concerns of using that pattern. There's also a variation of a long-lived release branch, which I'll be ready to describe shortly.

✢ ✢ ✢

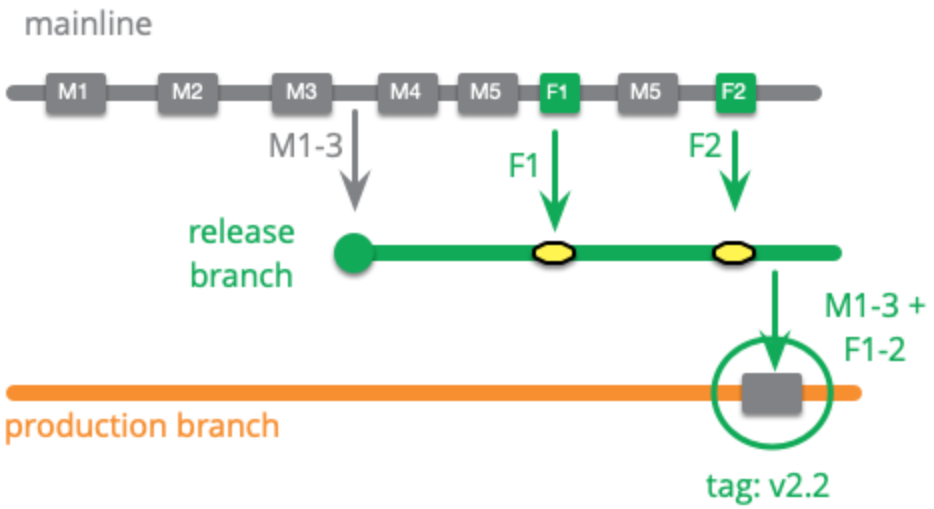
✢ Maturity Branch ✢

> *A branch whose head marks the latest version of a level of maturity of the code base.*

Teams often want to know what the most up to date version of the source is, a fact that may be complicated with a codebase that has varying levels of maturity. A QA engineer may want to look at the latest staging version of the product, someone debugging a production failure wants to look at the latest production version.

Maturity branches provide a way of doing this tracking. The idea that once a version of a code base reaches a certain level of readiness, it's copied into the specific branch.

Consider a maturity branch for production. When we are getting a production release ready, we open a release branch to stabilize the product. Once it's ready we copy it to a long-running production branch. I think of this as copy rather than a merge, as we want the production code to be exactly the same as what was tested on the upstream branches.



One of the appeals of a maturity branch is that it clearly shows each version of the code that reaches that stage in the release workflow. So in the example above, we only want a single commit on the production branch that combines commits M1-3 and F1-2. There are a few bits of SCM-jiggery-pokery to pull this off, but in any case this loses the link to the fine-grained commits on mainline. These commits should be recorded in the commit message to help people track them down later.

Maturity Branches are usually named after the appropriate stage in the development flow. Hence terms like “production branch”, “staging branch”, and “QA branch”. Occasionally I've heard people refer to a production maturity branch as the “release branch”.

When to use it

Source-control systems support collaboration and tracking the history of a codebase. Using a maturity branch allows people to get at a couple of important bits of information by showing the version history of particular stages in a release work-flow.

I can find the latest version, such as currently running production code, by looking at the head of the relevant branch. If a bug comes up that I'm sure wasn't there beforehand, I can look at what previous versions are on the branch and see the specific code base changes in production.

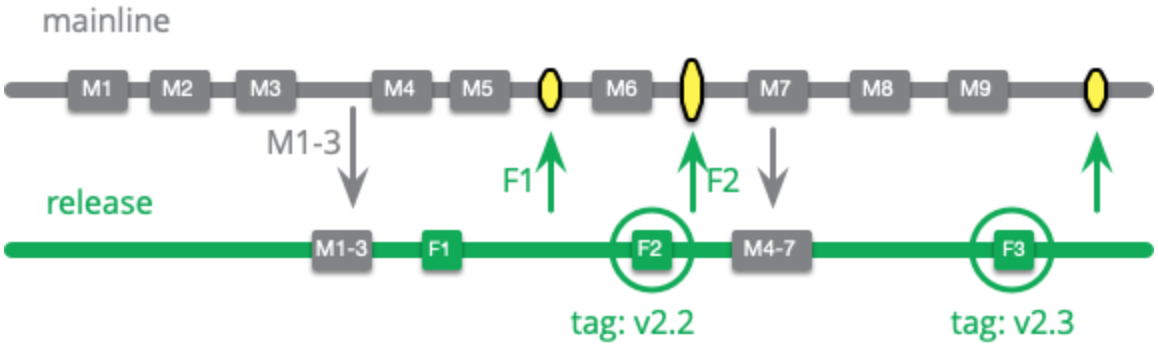
Automation can tie into changes to specific branches - for example an automated process can deploy a version into production whenever a commit is made to the production branch.

An alternative to using maturity branches is to apply a tagging scheme. Once a version is ready for QA, it can be tagged as such - typically in a way that includes a build number. So when build 762 is ready for QA it can be tagged “qa-762”, when ready for production it gets “prod-762”. We can then get a history by searching the code repository for tags that match our tagging scheme. Automation can similarly be based off tag assignments.

A maturity branch can thus add some convenience to the workflow, but many organizations find tagging to work perfectly well. So I look at this as one of those patterns that doesn't have strong benefits or costs. Often, however, the need to use the source-code management system for tracking like this is a sign of poor tooling for a team's Deployment Pipeline.

Variation: Long Lived Release Branch

I can think of this as a variation of the release branch pattern, which combines it with a maturity branch for a release candidate. When we wish to do a release we copy mainline into this release branch. As with the per-release branches commits are only made onto the release branch to improve stability. These fixes are also merged into mainline. We tag a release when it happens and can copy mainline in again when we want to do another release.



Commits may be copied in as is more typical in maturity branches, or merged in. If merged in we must be careful that the head of the release branch exactly matches the head of mainline. One way to do this is to revert all the of fixes that have been applied to mainline before merging. Some teams also squash commits after merging to ensure each commit represents a complete release candidate. (People who find this tricky have a good reason to prefer cutting a new branch for each release.)

This approach is only suitable for products with a single release in production at a time.

> One reason teams like this approach is that it ensures the head of the release branch always points to the next release candidate, rather having to dig out the head of the latest release branch. However, at least in git, we achieve the same effect by having a “release” branch name that moves with a hard reset when the team cuts a new release branch, leaving a tag behind on the old release branch.

✢ ✢ ✢

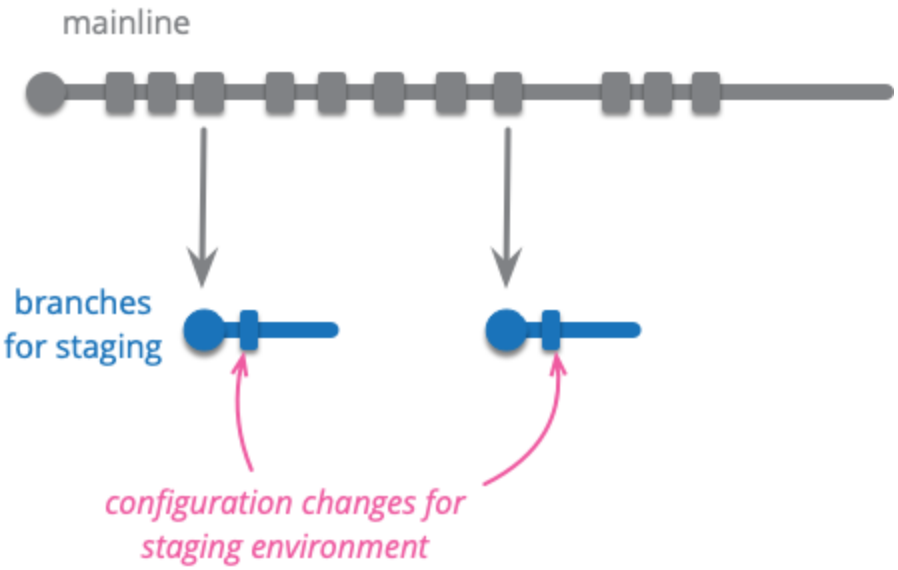
✢ Environment Branch ✢

> *Configure a product to run in a new environment by applying a source code commit.*

Software usually needs to be run in distinct environments, such as a developer's workstation, a production server, and perhaps various testing and staging environments. Usually running in these different environments requires some configuration changes, such as the URL used to access the database, location of a messaging system, and URLs for key resources.

An environmental branch is branch that contains commits that apply to the source code to reconfigure the product to run in a different environment. We may have

version 2.4 running on mainline, and now wish to run it on our staging server. We do this by cutting a new branch starting with version 2.4, apply the appropriate environment changes, rebuild the product, and deploy it to the staging environment.



The changes are usually applied by hand, although if the folks responsible are comfortable with git they may cherry pick the changes from an earlier branch.

The environment branch pattern is often combined with Maturity Branch. A long-lived QA maturity branch may include the configuration adjustments for the QA environment. Merges into this branch would then pick up the configuration changes. Similarly a long lived release branch may include these configuration changes.

When to use it

An environment branch is an appealing approach. It allows us to tweak an application in any way we need to ready it for a new environment. We can keep those changes in a diff that can be cherry-picked into future versions of the product. It is, however, the classic example of an Anti Pattern - something that looks appealing when you start, but soon leads to a world of misery, dragons, and coronaviruses.

The looming danger with any shift in environment is if the application's behavior changes as we move it from one environment to another. If we can't take a version running in production and debug it on a developer's workstation, it makes it much harder to fix problems. We can introduce bugs that only show up in certain environments, most dangerously production environments. Because of this danger we want to ensure that, as much as possible, the same code runs in production as it does everywhere else.

The problem with environmental branches is the very flexibility that makes them so appealing. Since we can change any aspect of the code in those diffs, we can easily introduce configuration patches that lead to varied behavior and the bugs that come with them.

As a result, many organizations wisely insist on an iron rule that once an executable is compiled, it must be the same executable that runs in every environment. If configuration changes are required they must be isolated through mechanisms such as explicit configuration files or environment variables. That way they can be minimized to simple setting of constants that won't change during execution, leaving less room for bugs to breed.

The simple demarcation between executable and configuration can easily become very blurred with software that executes its source directly (e.g. JavaScript, Python, Ruby), but the same principles hold. Keep any environmental changes minimal, and don't use source branching to apply them. The general rule of thumb is that you should be able to check out any version of the product and run it in any environment, so anything that changes purely due to different deployment environments shouldn't be in source control. There is an argument for storing combinations of default parameters in source control, but each version of an application should be able to switch between these different configurations as needed based on a dynamic factor such as environment variables.

Environment Branches are an example of using source branching as a poor man's modular architecture. If an application needs to run in different environments, the ability to switch between different environments needs to be first-class part of its design. Environment branches can be useful as a jerry-rigged mechanism for an application lacking that design, but then should be a high priority for removal with a sustainable alternative.

✢ ✢ ✢

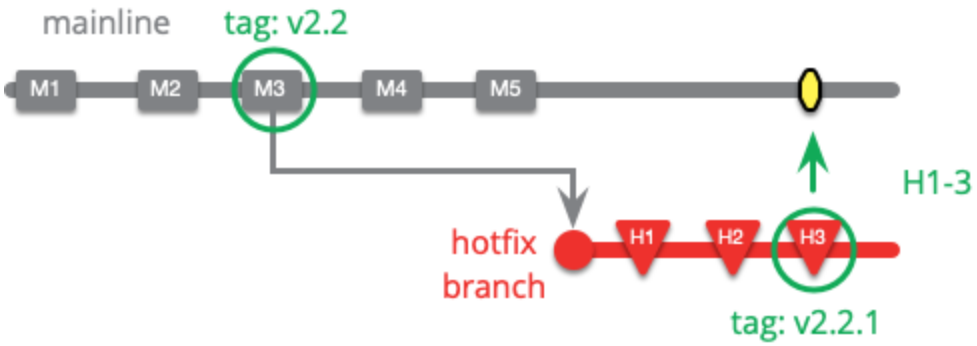
✢ Hotfix Branch ✢

A branch to capture work to fix an urgent production defect.

If a serious bug appears in production, then it needs to be fixed as soon as possible. Work on this bug will be a higher priority than any other work the team is doing, and

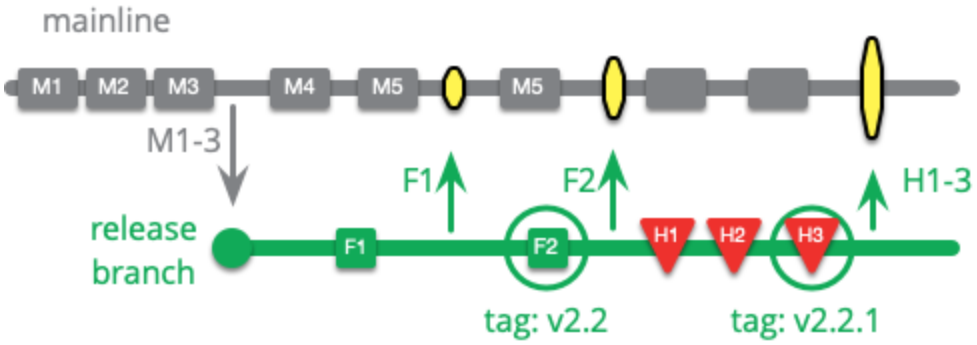
no other work should do anything to slow down work on this hotfix.

Hotfix work needs to be done in source control, so that the team can properly record and collaborate on it. They can do this by opening a branch at the latest released version and applying any changes for the hotfix on that branch.



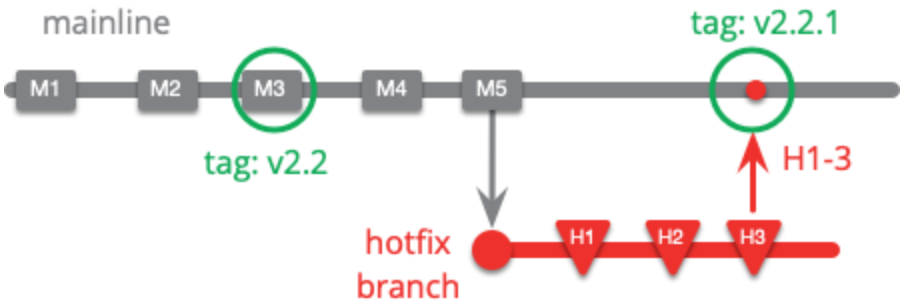
Once the fix is applied to production, and everyone has had the chance to have a decent night's sleep, then the hotfix can be applied to mainline to ensure there isn't a regression with the next version. If there's a release branch open for the next version, the hotfix will need to go on that too. If the time between releases is long, then the hotfix is likely to be made on top of code that's changed, so will be more awkward to merge in. In this case good tests that expose the bug are really helpful.

If a team is using release branches, hotfix work can be done on the release branch and a new release made when completed. Essentially this turns the old release branch into a hotfix branch.



As with release branches, it's possible to make the hotfixes on mainline and cherry-pick them over to the release branch. But this is less common as hotfixes are usually done under strong time-pressure.

If a team does Continuous Delivery, it can release hotfixes directly off mainline. They may still use a hotfix branch, but they'll start it from the latest commit, rather than the last released commit.



I've labeled the new release 2.2.1, since if a team's working this way it's likely that M4 and M5 don't expose new features. If they do, then the hotfix is likely to be just folded into a 2.3 release. This illustrates, of course, that with Continuous Delivery hotfixes doesn't need to sidestep the normal release process. If a team has a sufficiently responsive release process a hotfix can be handled just the same as normal - and this is a significant benefit of a Continuous Delivery mindset.

One special handling that is appropriate for a Continuous Delivery team is to disallow any commits to mainline until the hotfix is completed. This fits with the mantra that nobody has a more important task to do than to fix mainline - and in fact is true of any defect found on mainline, even those not yet sent into production. (So I suppose it isn't really special handling.)

When to use it

Hotfixes are usually done during times of considerable pressure, and when a team is most under pressure that it's mostly likely to make mistakes. In those conditions, it's even more valuable than usual to use source control and commit more frequently than seems reasonable. Keeping this work on a branch lets everyone know what's being done to deal with the problem. The only exception would be a simple fix that can be applied directly to mainline.

The more interesting issue here is deciding what is a hot bug to fix, and what is something that can be left the normal development workflow. The more frequently a team releases, the more it can leave production bug fixes to the regular rhythm of development. In most cases the decision will depend primarily on the business impact of the bug, and how that fits with the release frequency of the team.

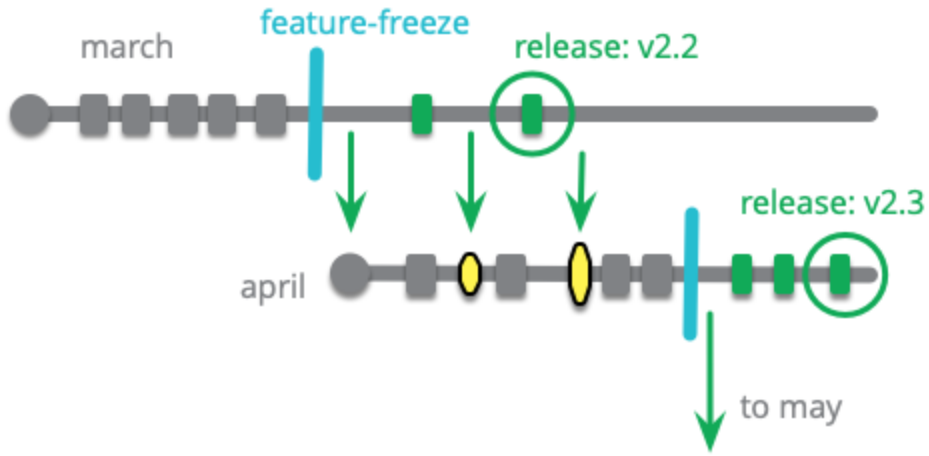
✢ ✢ ✢

✢ Release Train ✢

> *Release on a set interval of time, like trains departing on a regular schedule. Developers choose which train to catch when they have completed their feature.*

A team that uses a release train will set a regular cadence of releases, such as every two weeks, or every six months. Dates are set for when the team will cut a release branch for each release, following the metaphor of a train schedule. People decide which train they want a feature to catch, and target their work for that train, putting their commit onto the appropriate branch while the train is loading. Once the train departs, that branch is a Release Branch and will only accept fixes.

A team that is using monthly trains would begin a branch for March, based on the February release. They will add new features as the month goes on. On a set date, perhaps the third wednesday in the month, the train departs - feature-freezing that branch. They start a new branch for the April train, and add new features to that. Meanwhile some developers stabilize the March train, releasing it to production when it's ready. Any fixes applied to the March train are cherry-picked over to the April train.



Release trains are usually used with Feature Branching. When Scarlett senses when she'll be done with her feature, she will decide what train to catch. If she thinks she can be done for the March release she'll integrate into the March train, but if not she'll wait for the next one and integrate there.

Some teams use a soft-freeze a few days before the train departs (which is the hard freeze). Once the release train is in a soft-freeze state, then developers should not push work onto that train unless they are confident that their features are stable and ready for release. Any feature that exhibits a bug that's added after soft-freeze will be reverted (pushed off the train) rather than fixed on the train.

> These days, when people hear “release train” they are often hearing about the Agile Release Train concept from SAFe. SAFe's agile release train is a team organization structure, referring to a large-scale team-of-teams that shares a common release train schedule. While it uses the release train pattern, it's not the same as what I'm describing here.

When to use it

A central concept of the release train pattern is that regularity of the release process. If you know in advance when the release train should depart, you can plan your features to complete for that train. If you don't think you can get your feature finished for the March train, then you know you'll catch the next one.

Release trains are particularly useful when there is significant friction in the release process. An external testing group that takes a couple of weeks to verify a release or a release board that needs to agree before there's a new version of the product. If this is the case, it's often wiser to try to remove the release friction and allow more frequent releases. There are situations, of course, where this may be well-nigh impossible, such as the verification process used by app stores on mobile devices. Tuning release trains to match such release friction may then make the best of the situation.

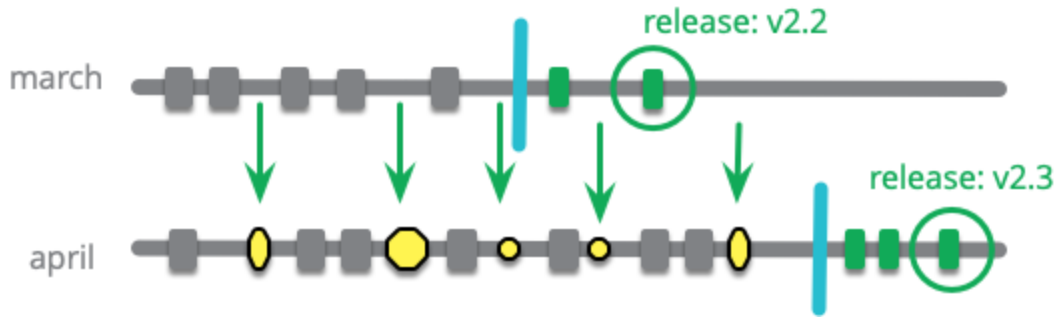
The release train mechanism helps concentrate everyone's attention on what features should appear when, and thus help predictability of when features will be completed.

A clear downside of this approach is that features completed early in the train's period will sit on the train reading a book while waiting for the departure. If these features are important, that's means the product misses an important capability for several weeks or months.

A release train can be a valuable stage in improving a team's release process. If a team has difficulty making stable releases, then jumping all the way to continuous delivery can be a jump too far. Picking a suitable release train period, one that's hard but plausible, can be a good first step. As the team gains skills they can increase the frequency of the trains, eventually abandoning them for continuous delivery as their capability grows

Variation: Loading future trains

The base example of a feature train has a new train arrive at the platform to pick up features at the same time as the previous train departs. But another approach is to have more than one train accepting features at the same time. If Scarlett doesn't think her feature will be done for the March train, she can still push her mostly-done feature to the April train, and push further commits to finish it off before it departs.

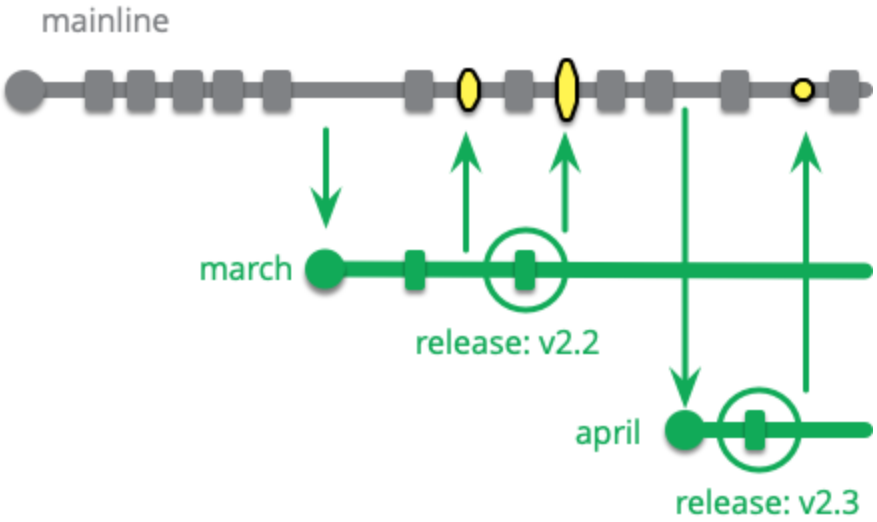


At regular intervals, we pull from the March train into the April train. Some teams prefer to only do this when the March train departs, so they only have one merge to do, but those of us who know small merges are exponentially easier would prefer to pull every March commit as soon as we can.

Loading a future train allows developers who are working on April features to collaborate without disturbing the work on the March train. It has the disadvantage that if folks on April make changes that conflict with the March work, the March workers don't get feedback, thus making future merges more complicated.

Compared to regular releases off mainline

One of the main benefits of a release train is a regular cadence of releases to production. But having multiple branches for new development adds complexity. If our aim is regular releases, we can achieve this using mainline just as well. Decide what the release schedule is, and then cut a release branch on that schedule from whatever is on the tip of the mainline.



If there's a Release-Ready Mainline, there's no need for a release branch. With regular releases like this, there's still the option for developers to hold back a nearly-done feature for the next release by not pushing to the mainline if it's just before the regular release date. With Continuous Integration, folks can always delay placing the keystone or keep a feature flag turned off if they want a feature to wait for the next scheduled release.

✢ ✢ ✢

✢ Release-Ready Mainline ✢

> *Keep mainline sufficiently healthy that the head of mainline can always be put directly into production*

When I started this section, I commented that if you make Mainline a Healthy Branch, and you make the health check sufficiently high, then you can release directly off mainline, recording the release with tag, whenever you wish.

I've spent a lot of time describing patterns that are alternatives to this simple mechanism, so I think it's time to highlight this one because, if a team can do it, it's an excellent choice.

Just because every commit made to mainline is releasable, doesn't mean it should be released. This is the subtle distinction between Continuous Delivery and Continuous Deployment. A team using continuous deployment does release every change accepted to mainline, but with continuous delivery while every change is releasable, it's a business decision whether or not to release. (Continuous deployment is thus a subset of continuous delivery.) We can think of continuous delivery as giving us the option to release at any time, our decision to exercise that option depends on broader issues.

When to use it

Coupled with Continuous Integration as part of Continuous Delivery, a release-ready mainline is a common feature of high-performing teams. Given that, and my well-known enthusiasm for continuous delivery, you might expect I'd say that a release-ready mainline is always the superior choice to the alternatives I've been describing in this section.

Patterns, however, are all about context. A pattern that is excellent in one context can be a trap in another. Release-ready mainline's effectiveness is governed by the integration frequency of the team. If the team uses Feature Branching and typically only integrates a new feature once per month, then the team is likely to be in a bad place, and an insistence on a release-ready mainline may be a barrier to their improvement. The bad place is that they cannot respond to changing product needs, because the cycle time from idea to production is too long. They are also likely to have complex merges and verification as each feature is large, leading to many conflicts. These may show up at integration time, or be a continuing drain on developers when they pull from mainline into their feature branches. This drag discourages refactoring, which reduces modularity, which exacerbates the problem.

The key to getting out of this trap is to increase integration frequency, but in many cases this can be difficult to achieve while maintaining a release-ready mainline. In this case it's often better to give up release-ready mainline, encourage more frequent integration, and use a Release Branch to stabilize mainline for production. Over time, of course, we'd hope to remove the need for a release branch by improving the deployment pipeline.

In the context of high-frequency integration, a release-ready mainline has the obvious advantage of simplicity. There's no need to bother with all the complexities of the various branches I've described. Even hotfixes can be applied to mainline and then to production, making them no longer special enough to deserve a name.

Furthermore, keeping the mainline release-ready encourages a valuable discipline. It keeps production-readiness at the top of developers' minds, ensuring that problems don't gradually creep into the system, either as bugs or as process-problems that slow down the cycle-time of the product. The full discipline of continuous delivery - with developers integrating many times a day into mainline without breaking it - seems dauntingly difficult to many. Yet once achieved and become a habit, teams find it remarkably reduces stress and is relatively easy to keep up. That's why it's a key element of delivering zone of the Agile Fluency® Model.

✢ ✢ ✢

Other Branching Patterns

The primary thrust of this article is on discussing patterns around team integration and the path to production. But there are some other patterns I'd like to mention.

✢ Experimental Branch ✢

> *Collects together experimental work on a code base, that's not expected to be merged directly into the product.*

Experimental Branches are where developers want to try out some ideas, but don't expect their changes to be simply integrated back into mainline. I may discover a new library, that I think would be a good replacement for one we're already using. To help make the decision whether to cut over, I start a branch and just try writing, or rewriting, a relevant part of the system using it. The aim of exercise isn't to contribute

code to the code base, but to learn about the applicability of a new tool in the my specific context. I may do this on my own, or work on it together with some colleagues.

Similarly, I have a new feature to implement, and can see a couple of ways to approach it. I spend a couple of days on each alternative to help me decide which one to go with.

The key point here is that the expectation is that the code on the Experimental Branch will be abandoned, and not merged into mainline. This isn't absolute - if it just happens that I like the result and the code can be integrated easily then I won't ignore the opportunity - but I don't *expect* that to be the case. I may relax a number of usual habits, less testing, some willy-nilly code duplication rather trying to refactor it in cleanly. I expect that if I like the experiment I'll work from scratch applying the idea to the production code, using the experimental branch as a reminder and guide, but not using any of the commits.

Once I've finished with work on an experimental branch, in git I'll usually add a tag and remove the branch. The tag keeps the codeline around in case I want to reexamine it later - I use a convention such as starting the tag name with “exp” to make its nature clear.

When to use it

Experimental branches are useful whenever I want to try something and I'm not sure I'll end up using it. This way I can do whatever I like, however zany, but am confident I can easily put it to one side.

Sometimes I'll be thinking I'm working on regular work, but realize that what I'm doing is really an experiment. If that happens I can open a new experimental branch and reset my main work branch to the last stable commit.

✢ ✢ ✢

✢ Future Branch ✢

> *A single branch used for changes that are too invasive to be handled with other approaches.*

This is a rare pattern, but crops up occasionally when people are using Continuous Integration. Sometimes a team needs to make a change that is very intrusive to the

code base, and the usual techniques for integrating work-in-progress don't apply well. In this case the team does something that looks much like Feature Branching, they cut a Future Branch and only pull from mainline, not doing Mainline Integration until the end.

The big difference between a Future Branch and Feature Branch, is that there's only one Future Branch. Consequently people working on the Future Branch never stray too far from mainline, and there's no other divergent branch to deal with.

It may be that several developers work on the Future Branch, in which case they do Continuous Integration with the Future Branch. When doing their integrations they first pull from mainline into the Future Branch before they integrate their change. This will slow down the integration process, but is the cost of using a Future Branch.

When to use it

I should stress that this is a rare pattern. I suspect most teams who do Continuous Integration never need to use it. I've seen it for particularly invasive changes to the architecture in the system. In general this is a last resort, only to be used if we can't figure out how to use something like Branch By Abstraction instead.

Future Branches should still be kept as short as possible, because they create a partition in the team, and like partitions in any distributed system, we need to keep them to an absolute minimum.

✢ ✢ ✢

✢ Collaboration Branch ✢

> *A branch created for a developer to share work with other members of the team without formal integration.*

When a team uses Mainline, then most collaboration occurs through the mainline. Only when Mainline Integration occurs do other members of a team see what a developer is doing.

Sometimes a developer wants to share their work before integration. Opening a branch for collaboration allows them to do this on an ad-hoc basis. The branch can either be pushed to the the team's central repository, collaborators can pull and push directly

from their personal repositories, or a short-lived repository can be set up to handle the collaboration work.

A collaboration branch is usually temporary and closed off once the work in integrated into mainline.

When to use it

Collaboration branches become progressively more useful as integration frequency decreases. Long-lived feature branches often require informal collaboration if team members need to coordinate some changes to an area of code that's important to several people. However a team using Continuous Integration will likely never need to open a collaboration branch, since they have only short periods when their work isn't visible to each other. The main exception to this is an Experimental Branch, which by definition never gets integrated. If several people work on an experiment together, they need to make the experimental branch be also a collaboration branch.

✢ ✢ ✢

✢ Team Integration Branch ✢

Allow a sub-team to integrate with each other, before integrating with mainline.

Larger projects may have several teams operating on a single logical code base. A team integration branch allows team members to integrate with each other without integrating with all members of the project using the mainline.

Effectively the team treats the team integration branch as a mainline within the team, integrating with that as they would the overall project mainline. In addition to these integrations the team carries out a separate effort to integrate with the project mainline.

When to use it

The obvious driver for using a team integration branch is for codebases that are being actively developed by so many developers that it makes sense to split them into separate teams. But we should be wary of that assumption, as I've run into plenty of teams that seem too large to all work off a single mainline, yet manage to all the same. (I've had reports of this with up to a hundred developers.)

A more important driver for team integration branches is a difference in desired integration frequency. If the project overall expects teams to do feature branches of a couple of weeks in length, but the sub-team prefers Continuous Integration, then the team can set up a Team Integration Branch, do Continuous Integration with that, and integrate the feature they are working on with mainline once it's done.

A similar effect is in play if there is a difference between the standards the overall project is using for a Healthy Branch and the health standards of the sub-team. A sub-team might choose to operate at a stricter level of health if the wider project cannot maintain mainline to high-enough degree of stability. Similarly if the sub-team is struggling to make its commits healthy enough for a well-controlled mainline, they may choose to use a Team Integration Branch and use their own Release Branches to stabilize code before going to mainline. That's not a situation I'd usually favor, but can be necessary in particularly fraught situations.

We can also think of a Team Integration Branch as a more structured form of Collaboration Branch, one that's based on formal project organization rather than ad-hoc collaboration.

✢ ✢ ✢

Looking at some branching policies

In this article, I've talked about branching in terms of patterns. I do this because I don't wish to advocate The One Approach to Branching, but rather to lay out the common ways people do this, and reflect on their trade-offs within the wide range of different contexts that we find in software development.

Lots of branching approaches have been described over the years. As I've tried to understand how they work and when they are best used, I've assessed them through the half-formed patterns in my mind. Now I've finally developed and written these patterns down, I think it's useful to look at some of these policies and see how I think about them in terms of the patterns.

Git-flow

Git-flow has become one of the most common branching policies that I've run into. It was written by Vincent Driessen in 2010, appearing as git was getting popular. In the days before git, branching was often seen as an advanced topic. Git made branches more appealing, partly because of improved tooling (such as handling file movement better) but also because cloning a repository is essentially a branch, requiring similar thinking about merge issues when pushing back to a central repository.

Git-Flow uses Mainline, (calling it “develop”) in a single “origin” repository. It uses Feature Branching to coordinate multiple developers. Developers are encouraged to use their personal repositories as Collaboration Branch to coordinate with other developers working in similar work.

The traditionally named core branch of git is “master”, in git-flow, master is used as a Production Maturity Branch. Git-Flow uses a Release Branch so that work passes from “develop” through the release branch to “master”. Hotfixes are organized through a Hotfix Branch.

Git-Flow doesn't say anything about the length of feature branches, hence nor the expected integration frequency. It's also silent on whether mainline should be a Healthy Branch and if so what level of health is needed. The presence of release branches implies it isn't a Release-Ready Mainline.

As Driessen pointed out in an addendum this year, git-flow was designed for the kinds of projects where there were multiple versions released in production, such as software installed on customer sites. Having multiple live versions is, of course, one of the main triggers for using release branches. However many users picked up on git-flow in the context of a single-production webapp - at which point such a branching structure easily becomes more complicated than necessary.

While git-flow is very popular, in the sense that many people say they use it, it's common to find people who say they are using git-flow are actually doing something quite different. Often their actual approach is closer to GitHub Flow.

GitHub Flow

While Git-flow really caught on, the unnecessary complexity of its branching structure for web applications encouraged plenty of alternatives. With the rise of popularity of

GitHub, it's no shock that a branching policy used by its developers would become a well known policy - called GitHub Flow. The best description is by Scott Chacon

With a name like GitHub Flow, there's no surprise that it was knowingly based on, and a reaction against, git-flow. The essential difference between the two is a different kind of product, which means a different context, and hence different patterns. Git-Flow assumed a product with several versions in production. GitHub Flow assumes a single version in production with high-frequency integration onto a Release-Ready Mainline. With that context, Release Branch isn't needed. Production issues are fixed in the same way as regular features, so there's no need for Hotfix Branch, in the sense that a hotfix branch usually means a deviation from the normal process. Removing these branches drastically simplifies the branching structure to a mainline and feature branches.

GitHub Flow calls its mainline “master”. Developers work with Feature Branching. They push their feature branches regularly to the central repository, to support visibility, but there's no integration with mainline until the feature is complete. Chacon indicates that feature branches can be a single line of code, or run for a couple of weeks. The process is intended to work the same way in either case. Being GitHub, the pull-request mechanism is part of Mainline Integration and uses Pre-Integration Review.

Git-flow and GitHub Flow often get confused, so as ever with these things, dig deeper than the name to really understand what's going on. The general theme of both of them is using mainline and feature branches.

Trunk-Based Development

As I wrote earlier, I mostly hear “trunk-driven development” as a synonym for continuous integration. But it's also reasonable to see Trunk-Driven Development as a branching policy alternative to git-flow and GitHub Flow. Paul Hammant has written an in-depth website to explain this approach. Paul was a long-time colleague of mine at Thoughtworks, and he had a solid record of wading into a client's ossified branching structures with his trusty +4 machete.

Trunk-Based Development focuses on doing all work on Mainline (called “trunk”, which is a common synonym for “mainline”), and thus avoiding any kind of long-lived branches. Smaller teams commit directly to mainline using Mainline Integration, larger teams may use short-lived Feature Branching where “short” means no more than a couple of days - which probably equates to Continuous Integration in practice. Teams

may use Release Branch (called “branch for release”) or Release-Ready Mainline (“release from trunk”).

Final Thoughts and Recommendations

Since the earliest programs, people found that, if they wanted a program that did something a little bit different to an existing program, it was easy to take a copy of the source and tweak it as desired. With all the source, I have the power to make any change I desire. But with that action, I make it harder for my copy to accept new features and bug fixes in the original source. In time, it can become impossible, as so many enterprises discovered with their early COBOL programs, and suffer with extensively customized ERP packages today. Even if the name isn't used, any time we copy source code and modify it we are Source Branching, even if there's no version control system involved.

As I said at the beginning of this long piece: branching is easy, merging is harder. Branching is a powerful technique, but it makes me think of goto statements, global variables, and locks for concurrency. Powerful, easy to use, but easier to over-use, too often they become traps for the unwary and inexperienced. Source code control systems can help to control branching by carefully tracking changes, but in the end they can only act as witnesses to the problems.

I'm not someone who says branching is evil. There are everyday problems, such as multiple developers contributing to a single codebase, where the judicious use of branching is essential. But we should always be wary of it and remember Paracelsus's observation that the difference between a beneficial drug and a poison is dosage.

So my first tip for branching is: **whenever you're considering using a branch, figure out how you are going to merge**. Any time you use any technique, you're trading off against alternatives. You can't make a sensible trade-off decision without understanding all the costs of a technique, and with branching the piper exacts her fee when you merge.

Hence the next guideline: **make sure you understand the alternatives to branching, they are usually superior**. Remember Bodart's Law, is there a way to solve your problem by improving your modularity? Can you improve your deployment pipeline? Is a tag enough? What changes to your process would make this branch unnecessary? It's quite likely that the branch is, in fact, the best route to go right now - but is a smell that's alerting you to a deeper problem that should be tackled over the next couple of months. Getting rid of the need for a branch is usually a Good Thing.

Remember LeRoy's Illustration: branches diverge exponentially as they run without integrating. So consider how often you integrate your branches. **Aim to double your integration frequency.** (There's obviously a limit here, but you won't be near it unless you're in the zone of Continuous Integration.) There will be barriers to integrating more often, but those barriers are often exactly the ones that need to be given an excessive dose of dynamite in order to improve your development process.

Since merging is the hard part of branching, **pay attention to what's making merging difficult**. Sometimes it's a process issue, sometimes its a failing of the architecture. Whatever it is, don't give in to Stockholm Syndrome. Any merge problem, especially one that causes a crisis, is a signpost to improving a team's effectiveness. Remember that mistakes are only valuable when you learn from them.

The patterns I've described here outline common configurations of branching I've come across in my and my colleagues' travels. By naming them, explaining, and above all, explaining when they are useful, I hope that helps you evaluate when to use them. Remember that, as with any pattern, they are rarely universally good or bad - their value to you depends on the context that you're in. When you run into branching policies (whether well-known ones like git-flow or trunk-based development, or something home-grown in a development organization) I hope understanding the patterns within them will help you decide whether they are a good fit for your circumstances, and what other patterns would be useful to blend into the mix.