Patterns for Agile Codelines

Steve Berczuk

steve@berczuk.com

Abstract

This paper presents a number of patterns from a Pattern Language for Agile Codelines. This pattern language builds on the previously published *Software Configuration Management Patterns* but is a complete rewrite, focusing on creating a development ecosystem that enables quicker, more robust deployments while gaining the benefits of human feedback.

An Overview of the Pattern Language

Since *Software Configuration Management Patterns* was published in 2002, a few things have changed:

- Distributed Version Control Systems, in particular Git, have become standard,
- Agile Development practices are more widely used
- Planning and team culture are seen as key to effective delivery.

What hasn't changed is that teams struggle with using their codelines best to develop in an agile context. Many teams still use byzantine branching structures that lead to slow integration. Others advocate skipping branching altogether. This pattern lan-

guage shows practices to help teams use basic and simple branching patterns and the planning, testing, and culture patterns that support them to get the most out of their tools while becoming more agile.

This paper updates the pattern language to reflect how small—to medium-sized (5-10 person) software teams can use a DVCS like Git in their development process.

While there is a trend towards Trunk Based Development, with minimal branching, a guide to using branching well is useful because:

- Many acknowledge that *short-lived* branches are consistent with the Trunk Based Development approach.
- The prerequisites for merging directly to a Main Line without a branch are the same as those for using short-lived branches, so the material is relevant in both cases
- Many teams still struggle with using branching effectively and eliminating branching seems too risky.

This pattern language is opinionated about short-lived branches and support for distributed teams, though some of the underlying practices are relevant in other contexts.

This pattern language assumes some familiarity with branching and Agile software development. Refer to the Appendix for some background material.

The Scope of the Pattern Language

Figure 1 shows the pattern language. The patterns in this paper are highlighted with a bold border.

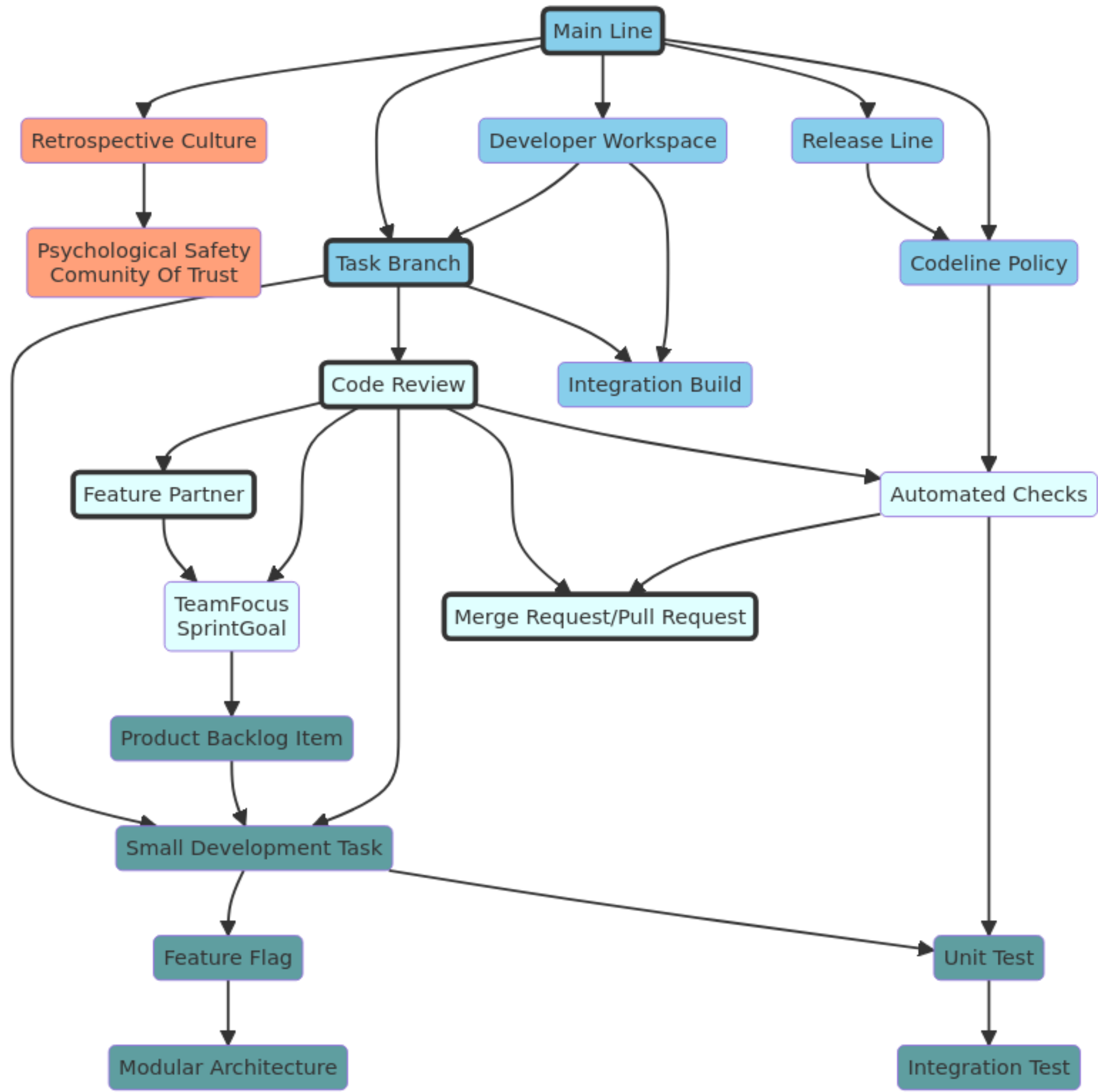


Figure 1: Pattern Language Overview

A Pattern Sequence

To give you a sense of how the Software Configuration Management Patterns work, here is one scenario showing how you might use them to improve your delivery process using the *Pattern Names*. This story covers the entire language with the patterns in this paper in **bold**.

You realize your current integration process, which uses GitFlow, isn't working as well as it could. Changes take too long to get delivered, frustrating the developers and the product owners. You want to simplify the process, making it faster, but you are concerned about quality. Your current process is slow, but it seems to identify issues, though some issues slip through, and you aren't delivering the value you feel you could.

You think that the SCM Patterns could help, so you start to change your process so that all changes go to, and all releases come from, the ***Main Line***. Mechanically that's easy; you can just change how your branching strategy. But you want more than speed. The things that slowed down gave you an illusory sense of control; going more slowly doesn't always mitigate risk to the extent that you think it might and introduces business risk, so you look to the patterns to maintain a stable *Main Line*.

The first step is to make it easier for developers to have confidence that what they code and test locally will also work for others and in the production environment. Having the ability to create a *Developer Workspace* that looks as much like the production as possible, including versions of tools. 100% match may not be easy, but you can avoid many obvious errors.

While committing from a *Developer Workspace* to the *Main Line* might be your eventual goal, you still see some benefit from using branches, albeit differently than you had in the past. Two kinds of branches you'll use are a *Release Line* — which is where you will keep track of the current release in case you need to deploy a fix; you

don't expect to make changes to the code in the *Release Line* , but it's there in cases your in a spot where delivery from the *Main Line* won't be timely enough. You will use a short-lived ***Task Branch*** each of which will be used to work on a *Small Development Task*. Your goal is to have all development work integrated daily or better.

To keep the team on the same page about the rules for using branches, you document a *Codeline Policy* and add some simple enforcement mechanisms in the SCM system.

For each *Small Development Task*, developers write *Unit Tests* to provide a goal (for new tests) and confirm that your change isn't breaking anything else. *Unit Tests*, while valuable, make assumptions about interfaces. To provide a sanity check that the contract the units tests assume didn't break, the team also writes some lightweight *Integration Tests*.

Small Development Tasks mean that you will be committing consistent units of work, but the work will not implement a complete feature. In some cases, that work won't be visible, but in others, it may be. To avoid unexpected behavior, you can use *Feature Flags* to hide work in progress in certain environments. A *Modular Architecture* makes feature flags easier to implement.

When a *Small Development Task* is complete and the *Task Branch* Is ready to merge developers as for a ***Code Review***, leveraging your ***Pull Request*** process to allow for knowledge sharing, feedback about design and implementation. *Feature Partners* ensure that someone can promptly give good feedback, and a *Team Focus* around completing a *Product Backlog Item* helps to ensure that the team prioritizes reviews.

In the past the team seemed to spend a lot of effort commenting on style and formatting issues rather than design and requirements. *Automated Checks* identify these

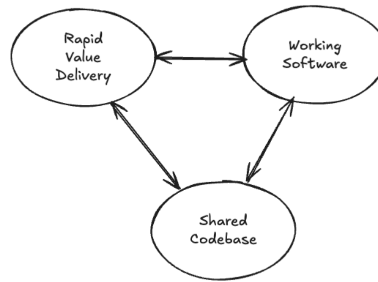
kinds of issues, not to and confirm that all the automated tests pass. These automated checks can run as part of an *Integration Build* .

You realize that when you move fast things will break. And that's OK since no one is perfect and most failures will be related to things you haven't anticipated. You want to encourage an inspect and adapt mindset, and avoid a culture of blame; you do what you can to create a *Retrospective Culture*. A *Retrospective Culture* means having retrospectives, but also incorporating the values of retrospectives into daily work.

At each step of the process change, as your implementation of the patterns improved, your delivery speed improved, and the quality didn't seem to suffer; any mistakes that happened were small and no worse than(and often less severe) the errors that survived the old process. There is still more to do; eventually, you can further streamline your process. You realize that, while there were technical changes to be made, what held you back before was not giving enough credit to how planning and design interacted with technical practices. Improving one can help, but working through the pipeline gave you some big wins

Main Line

When you use an agile software development process like Scrum, you plan iteratively. You want your codeline process to support your agile delivery model. This pattern describes the high-level structure – a *Main Line* – that makes it easier to deploy code quickly to reveal business value while maintaining stability and traceability by providing a central integration point for code.



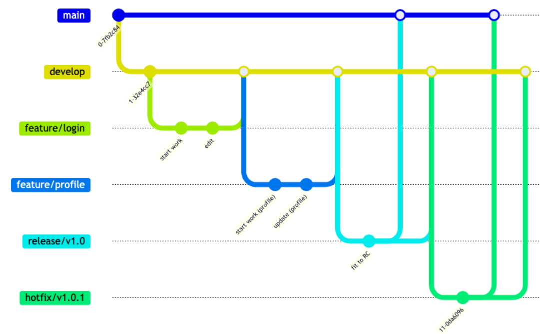
Forces for Main Line

Challenges of Balancing Speed and Stability

Agile software development is based on frequent iteration and feedback as measured by inspecting working software. But stability and speed can appear to be at odds. Some teams try to reduce the risk of error by slow, disciplined steps when integrating work into a delivery code line. These teams may have:

- More manual testing
- More peer reviews
- Intricate integration testing
- Strict approval gates to prevent 'accidental' integration
- One or more staging branches

The team integrates into a single shared code line only once it is "certain" that the code works. For example, in a *GitFlow* model, work in progress is integrated into a "Develop" branch, which is considered a working branch. The changes are merged to the shared mainline only after they have been approved for release.



There are variations, all of which work on the belief that isolation and moving slowly is safer. Keeping work on isolated branches preserves the stability of the eventual target branch in the short term but defers the problem: the target branch receives changes slowly, leading to process and business risk:

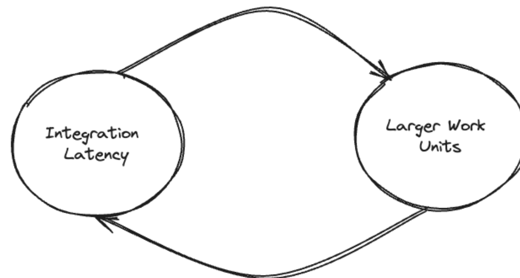
- Process risk: The longer work stays isolated, the greater the risk of merge conflicts and divergent design decisions, as well as more work-streams to maintain.
- Business risk: Slower code delivery means that features take longer to be released, which can lead to opportunity costs and longer feedback cycles based on larger features.

Since agile software development is about adapting to uncertainty in the project space, it is valuable to evaluate the current state of the code sooner.

Moving slowly can lead to a self-fulfilling dynamic:

- A slow integration process leads to a temptation to integrate larger, more complex units of work.
- The longer you keep your changes isolated, the harder the integration will be, both for your work and work started after your work stream started.

- The more overhead for a merge, such as more testing due to change set size, the greater the temptation to introduce more work before you merge.



Faster integration means you could miss an error, but slow integration doesn't guarantee perfect software. Regardless of the size of the unit of work or how quickly you integrate it, merging code that breaks the shared integration codeline will slow down the entire team.

Frequent integration is more productive: the more frequently you integrate, the simpler each integration will be because the change is smaller and the work started with more recent code.

Create a Stable Baseline

Therefore:

Work on a Main Line, where all work is integrated. Use mechanisms to allow work to be integrated frequently while maintaining stability so that the Main Line is potentially deployable.

A Main Line is a code line that:

- contains the “record” of the latest work
- tracks the current state of working code and is the starting point for any new work
- is the source of all releases (with the rare exception of emergency patch releases).

The *Main Line* is never deleted. It lives throughout the entire project, and the entire team contributes to it.

Work in progress, before it is merged to the *Main Line* could be:

- on another branch (as this pattern language describes) or
- In a developer workspace with no independent tracking branch.

In either case, the prerequisites are the same. This pattern language describes how to use short-lived branches, which is consistent with what *Accelerate* describes:

> *Following our principle of working in small batches and building quality in, high- performing teams keep branches short-lived (less than one day’s work) and integrate them into trunk/master frequently*.

The goal of Agile Software development is to manage uncertainty. As Mike Cohn wrote in *Agile Estimation and Planning*:

> *The best way of dealing with uncertainty is to iterate. To reduce uncertainty about what the product should be, work in short iterations, and show (or, ideally, give) working software to users every few weeks…*

Being biased toward quicker integration into a shared *Main Line*, rather than reducing error, can help you manage uncertainty by showing you an accurate current state. Errors will still happen; being able to recover when they do is more valuable than slowing down in an attempt to avoid them all. As Gary Klein writes in *Seeing What Others Don't*:

> *"When we put too much energy into eliminating mistakes, we're less likely to gain insights. Having insights is a different matter from preventing mistakes."*

A goal of an agile project is to gain insights into the state of the software by frequently inspecting the latest code in a running application.

Example

A *Main Line* development flow will look like the following, though each team can decide what the correct length of time is:

1. Checkout the HEAD of the Main Line into a development workspace
2. Code, backed by a *Task Branch*
3. Within a day, merge the code after an appropriate feedback process.

Cautions

Maintaining an active, healthy, *Main Line* takes discipline. An occasional error is inevitable, so while you might feel comfortable eliminating intermediate branches and getting code to the *Main Line* quickly, you may be tempted to add extra gates between "merged to main" and "released." While this might be a reasonable starting place, you want to work to get to a point where the merge to the *Main Line* is quick, automated and gives you high confidence.

Next Steps

While the Main Line model's simplicity, with fewer codelines, has advantages, you need some mechanisms to allow frequent integration to happen safely and reliably. You can't avoid all errors, but you can avoid major ones and reduce the impact of any that slip through.

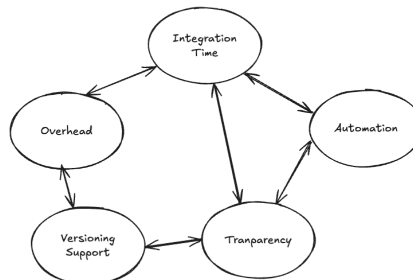
To help ensure a healthy *Main Line* you need to:

- Define the rules for integrating to the *Main Line* and when to use other codelines: *Codeline Policy*.
- Provide a place to reliably do development with the correct dependencies and tools: *Developer Workspace*
- Allow for delivery of critical fixes to released code: *Release Line*
- Enable Parallel, Independent Work that can be integrated into the *Main Line* quickly and reliably: *Task Branch*
- Get feedback on work before it's integrated into the Main Line: *Code* Review
- Build and Test automatically: *Integration Build*
- Get Feedback on design and implementation: *Pull Request*

- Create a *Retrospective Culture* that is robust in the face of the inevitability that things will break despite best efforts and has a continuous improvement mindset.

Task Branch

You need place work that facilitates the use of feedback mechanisms such as testing and *Code Review* to help you merge with confidence so that you can have consistently working code on the *Main Line*. This pattern describes a codeline structure that enables your team to work in parallel to deliver multiple changes to the *Main Line* quickly while preserving the integrity of the *Main Line* and maintaining focus.



Balancing Isolation and Collaboration

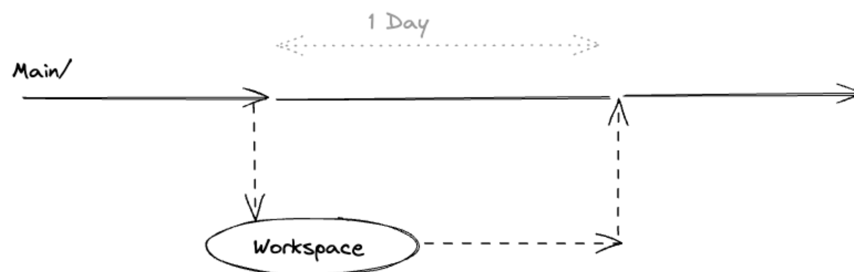
A typical development workflow includes the following steps:

- Pulling the current state of the code from the *Main Line*.
- Making changes to the code.
- Getting feedback on the change from team members and/or tools

- Merging the work into the *Main Line*.

It's typical to code and test in your private workspace before sharing your changes with the rest of the team. Unless you work in a communal workspace (which would be challenging), there is always a period of time when your changes are isolated from the *Main Line*. While this means that your work isn't always up to date, having no isolation can become chaotic as the number of contributors increases.

A workspace, with or without a local branch, *is a parallel work stream*, much like a shared branch, but without the transparency and automation support branching enables.



There are a few options for managing this isolated work stream. Each option differs in:

- How visible it is to other team members
- The ability to use the mechanisms of your version management system to have more freedom to experiment.
- How much you can leverage your shared CI workflow.

A good solution will allow you to work in a way that yields the benefits of (brief) isolation while minimizing the time until the code is merged into the *Main Line*.

Some of the ways you can manage the code in your workspace before integrating with the *Main Line* are:

- Push directly to the *Main Line* from your workspace.
- Create a branch locally that you don't push, and push the result to the *Main Line*
- Create a branch for your work and push it to the shared repository. When complete, merge this branch to the *Main Line* (directly or via a *Pull Request*).

A good solution will allow you to work in a way that yields the benefits of (brief) isolation while minimizing the time until the code is merged into the *Main Line*.

A workspace without a backing branch is straightforward to manage because it requires fewer interactions with the SCM tool. However, you cannot easily track steps during the coding process. Some of the disadvantages of this approach are:

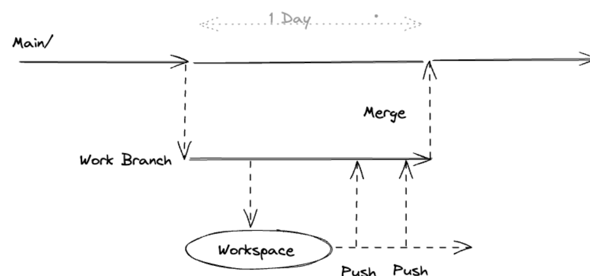
- Limited ability to get feedback using the tools available in a Continuous Integration Environment *before* you merge
- Making it more complicated to get input from team members who are not co-located (temporally or geographically) because your work-in-progress code is invisible to other team members
- An increased possibility of losing work in progress due to the lack of version and tooling support in a shared repository

A workspace with a local branch lets you track changes locally and experiment easily (see the *Private Versions* Pattern from SCM Patterns). However, this approach has the

same issues relating to lack of transparency and tooling support workspace-only approach.

Using a branch, you can collaborate on a feature with other developers and get the advantage of quickly testing the code in the branch in a Continuous Integration Environment, opening up more potential for automation, information sharing, and collaboration without the right expectations. However, working on a branch can encourage slower, asynchronous interactions.

Using branches can cause problems when teams encourage working on a long-lived branch—such as a Feature Branch—until work is complete. While this seems to offer some superficial advantages, especially if the work is isolated from the rest of the code base, the cost of a longer gap between integration easily outweighs any potential advantage. The longer the delay, the higher the risk of merge conflicts and errors.



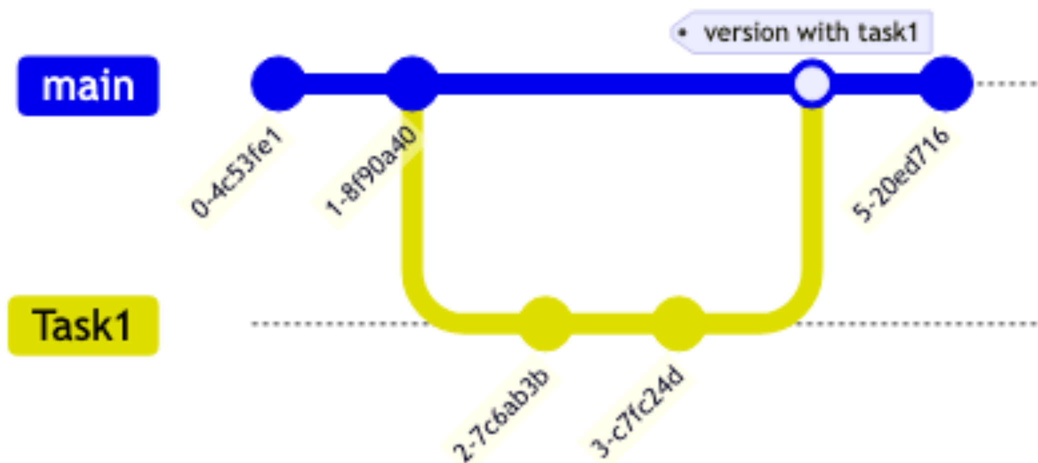
You want to enable collaboration and reliable testing while also working to minimize the length of time between starting work and merging, which includes:

- Time between work starting and code being completed.
    - This depends on the developer's skill and speed, and the task size
- Time between work being complete and merging
    - This depends on the dynamics of the feedback cycle

Branch for Tasks, Merging Quickly

Therefore:

For each development task, work off a Task Branch. A Task Branch represents a small coherent unit of work that can be done reasonably quickly. Merge into the Main Line as quickly as possible



Basic Task Branch

A Task Branch starts from the Mainline. The branch ends when the task is merged.

A *Task Branch* enables a developer to:

- Have a backing store for work in a workspace related to a *Small Development Task*.
- Maintain flow.
- Experiment
- Obtain Feedback

Using short-lived *Task Branches* is consistent with rapid integration to a *Main Line*. In *Accelerate* the authors say:

> *Following our principle of working in small batches and building quality in, high- performing teams keep branches short-lived (less than one day's work) and integrate them into trunk/master frequently.*

A "Task", which is described in more detail in *Small Development Task* can be a User Story or an intermediate step for a user story. The main attribute is that it is a small, coherent unit of work. Small can vary by team, but a typical goal is to be able to integrate at least daily.

Example

A *Task Branch* follows a familiar workflow:

1. Check out the *Main Line*
2. Create a *Task Branch*
3. Make code changes, including tests. This can include multiple commits
4. Push changes to the shared repository periodically.
5. Get feedback.
6. Merge the completed work.
7. Delete the *Task Branch*

The goal is to integrate the work into the *Main Line*quickly.

Cautions

Don't confuse a *Task Branch* with a *Feature Branch*. A *Task Branch* is shorter and allows for incremental work. A *Feature Branch* often represents a larger unit of work that survives until the "feature" is complete, at which point the code merges to the main line.

Feature branching is rarely a useful pattern to follow; use Task Branches for any work in progress.

Be mindful of:

- Merge Conflicts: To minimize the risk of merge conflicts causing delays at the end, pull from the *Main Line* periodically to simplify later merges and identify possible design divergence early.
- *Task Branches* that last a long time. Gather data (either metrics or heuristics) to identify when Task Branches take a long time to merge. Discuss these at Retrospective to evaluate if the long branches were problematic and, if so, how to fix the underlying issues.
- Overly restrictive *Codeline Policies* for the mainline that require a slow Code Review process.

Using a *Task Branch* can make these problems more obvious (since the SCM tooling makes the parallel work stream visible), but that doesn't mean that the branching policy is the cause of the problem. These issues can also manifest when doing a no-branch directly from the workspace workflow. The causes of long integration times are often related to problems external to the code line flow, such as planning and prioritization.

Aside: Branch Reuse

One approach to balancing the overhead of branch creation with frequent integration is to create one branch for a larger task, but merge multiple times during the branch's lifetime. If your policy if to delete the (remote) branch after a merge, this is conceptually the same as multiple task branches. The team should decided if there is any value

to creating uniquely named branches for each merge. In many cases, the "multiple merges" approach can work fine if the team finds it easier.

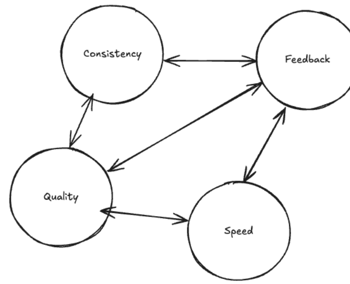
Next Steps

To integrate a task branch quickly while minimizing the risk of errors, you need to:

- Ensure that tasks are the right size to complete quickly and well-defined enough to know when they are done. Identify *Small Development Tasks* that support completing work quickly.
- Have a *Code Review* process to support shared understanding and identify likely errors.
- Use an *Integration Build* in a CI environment to ensure that a consistent set of checks is run on the code.

Code Review

When you are developing on *Task Branch*, quickly moving work to the *Main Line*, you want to ensure quality and consistency since changes affect the entire team. Tests and other automation help and feedback from other developers on approach, style, design, and testing approach can provide useful insights. This pattern describes a way to get prompt and useful feedback from team members before you integrate changes into the *Main Line*.



Balancing Feedback, Learning, Speed, and Quality

Before work related to a development task in a development workspace is ready to merge into the *Main Line*, the developer wants to have confidence that the code is:

- complete, robust, and free from errors: does what it's supposed to do and handles the range of inputs appropriately
- maintainable: won't add undue burdens or unreasonable tech debt to the code base,
- safe to merge: won't break other parts of the code

Some of these criteria can be addressed by automated testing and static analysis tools, but:

- Static analysis tools can provide insight into some maintainability issues, but they can sometimes be narrow in scope and not cover the entire product development context.
- The results of tests are only as good as the quality of the tests. In particular, when new code with tests is added, you want to be sure that the tests are appropriate for the task's goals.

You can ask another developer to review the code before it's merged as a person can provide perspectives that tools can't, but a review by another developer adds a handoff step to the process, and handoffs can introduce delays that can significantly slow down integration into the *Main Line*.

Another option is to use a "fail fast" approach, where a developer merges code to the main line when they think it is ready, and the team places mechanisms in place to recover from errors. Having reliable rollback mechanisms is an attribute of a robust development process, and this will help improve the speed of integration but:

- Issues detected after code is merged can slow down the whole team
- While "hard failures" would be easy to detect, more subtle design issues might linger until well after the time they were introduced, making them harder to understand (and fix)

Feedback can sometimes be viewed as judgmental and critical rather than a collaborative attempt to improve code quality. You want to ensure that the feedback can focus on significant issues rather than minutia or matters of opinion.

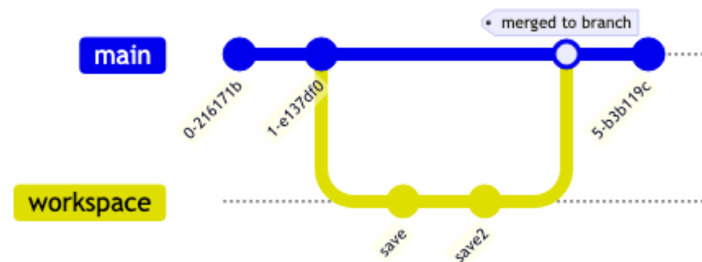
Quick feedback is better but feedback that is too cursory or not useful enough has little value. Spending more time giving can lead to a point of diminishing returns, reducing the impact of the feedback that you can get from deployed code.

You want to get the benefits of collaboration with your team to improve overall code quality while leveraging tools and automation and not injecting long delays into the integration process.

Relevant, Timely, and Actionable Feedback

Therefore:

Before merging code from a Task Branch into the *Main Line* get feedback from another developer. Identify practices that ensure that the feedback is timely and that the reviewer has the appropriate context. Leverage automation where possible to focus on human time. Acknowledge the cost-benefit for some trivial changes where cursory reviews might be sufficient.



Good Code Reviews are a form of collaboration that can:

- Reduce error
- Helps the code improve.
- Lead to a shared understanding of the code base.

Feedback that achieves these goals is:

- *Relevant*: Not just general comments about coding style but comments that take design conversations and the goal of the work into account.
- *Timely*: Feedback is shared shortly after the code is ready for review, and questions are answered promptly.
- *Actionable*: Feedback gives enough information to allow the developer to act on it.

Relevant feedback addresses the code's context, including requirements, design discussions, etc. General programming feedback about style and related issues is useful,

but *Automated Checks* are often better for that feedback. The more context the reviewer has about the problem, the solution, and the design, the easier it is to be relevant.

Timely feedback is sufficiently prompt so that the developer does not lose context because of time spent waiting. Timeliness does not mean "immediate," though sooner is often better. The definition of "timeliness" should be part of a team's working agreements.

Actionable feedback is as specific as possible and suggests specific actions to take or questions to discuss. Vague comments about concerns or errors slow down the process.

Ideally, reviewers should be participating team members. Avoid:

- Limiting reviews to senior people (or requiring certain reviewers). You want the reviewers to know the code and the problem, and initially, that might be the more experienced people, but the choice is about skill and knowledge, not status.
- Soliciting feedback from individuals who are not on the team. The team can prioritize features and reviews. Those outside the team might not be willing or able to, so tying integration to the schedule of someone outside the team can be problematic.

A designated reviewer model and external feedback can slow down the process due to scheduling bottlenecks and less contextually informed comments.

Code Review feedback can be delivered either interactively or asynchronously. Interactive feedback, where code is discussed in real-time, can take any number of forms:

- Pair Programming
- Desk Review with a *Feature Partner*.
- An interactive review after sharing a *Pull Request*

Synchronous feedback doesn't preclude some preliminary review and comment; much like preparing for a meeting, taking a few minutes to review code "offline" can make for a better review, as long as this internal is short.

Asynchronous Feedback is often associated with a *Pull Request* Process, though it can occur in other ways too. Asynchronous feedback can make sense for:

- Larger changes
- Trivial Changes, where feedback with comments is likely to be sufficient
- Distributed teams, where there is a working agreement setting out expectations and commitments about timeliness.

Even when feedback is asynchronous, it's essential that it be prompt and that conversations (as opposed to self-contained comments) move quickly to an interactive forum.

Remember:

- The goal is to *improve* the work. Yes, you want to identify problems, but you also want to suggest solutions.
- The participants are all creators; if you give feedback now, you may get feedback later. As a reviewer, consider how useful you might find a comment or request.
- To the extent that the code fulfills the requirements and meets team norms, and general design and implementation guidelines, trust the author of the code to make decisions about how to apply comments.

- A code review is not the time to impose personal preferences on the code, though it could be a time to identify gaps in guidelines. If you were not part of the design process for the code, acknowledge that you might be uncertain about some element of requirements or design,

If your process requires "approval" before merging and feedback is asynchronous, err on the side of permissive approval ("approved pending these changes", for example). Permissive approval avoids delays and demonstrates trust in team members.

Example

After a development task is done, a developer posts a message to the team Slack channel asking for feedback and indicating the location of the change. (The mechanism could be a *Pull Request*, or a *Task Branch* or simply a request to meet, though it's often useful to give people a short amount of time to read the code before gathering).

Team members offer feedback, and the developer makes appropriate changes and then merges the code.

Other Forums

Narrowing the scope of code review is an essential part of meeting the time to integration goals. Some goals are better achieved in other forums:

- Learning and knowledge sharing can happen in other feedback sessions or even "code workshops"
- Giving a new developer closer feedback as part of training could be done in an "out of band" review session by a manager or a mentor. This pattern is

about Code Review as part of the deployment pipeline. You can ask for reviews at any time.

Review Effort (and Skipping Review)

Not all code needs the same level of review; consider the risk of a change. Some changes you can identify as low risk (either via a tool or heuristic) might not need a peer review — passing automated tests and checks can be sufficient.

Metrics

If you feel like your code review process is bogging down, there are some metrics to track to help you identify bottlenecks:

- Time to first comment:
  - If the wait for a review frequently exceeds the team's working agreement, work to identify the issue. It could be over-commitment, or perhaps the expectation is unreasonable
- Time between interactions (if you do asynchronous reviews):
  - Asynchronous reviews can be useful for simple direct comments but don't work well for conversations. If the team feels that it takes too long to have a feedback conversation, consider defaulting to synchronous reviews.
- Time from Code Completion to Merge: This is the key metric for an agile codeline. If Code Review becomes the bottleneck in integration, understand why,

Using a *Pull Request* mechanism makes it easier to collect this data, though it is still relevant for any feedback process.

Cautions

If the team loses track of the purpose of *Code Review*—collaboration—it can easily become a bureaucratic, low-value process.

In particular, avoid the following traps:

- Treating code reviews as "evaluations." The goal is to improve the code. If a pattern emerges that leads to the thought that someone needs coaching, address that in another context, ideally with an "elevate the person's skills" frame.
- Having reviews require explicit approval gates. If you have a team dynamic where someone will blithely ignore show-stopper issues, this points to larger team dynamics issues. If you must have gates, make them part of your automated tests.

While speed is important, do not skip steps in favor of speed. If the feedback cycle is taking too long, work to improve the process.

When gathering metrics, use them for understanding rather than evaluation. If the metrics look bad, the reasons might be out of individuals' control yet easy to fix.

Next Steps

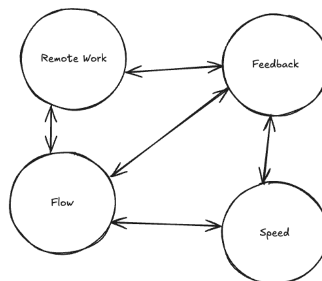
- Facilitate relevant (and timely) feedback by ensuring a *Team Focus* so that Team members can feel comfortable prioritizing reviews and finishing work

in progress over any individual goals. Avoid emphasizing individual tasks over larger goals.

- Use *Automated Checks* to catch mechanical issues with the code so that team members can focus on design and purpose.
- Make sure there is a reviewer with good context, such as a *Feature Partner* who understands the product scope and technical design.
- Enable automation and support remote team members using a *Pull Request*.
- Smaller tasks are easier to give good, prompt feedback on. Enable timely feedback with *Small Development Tasks*

Pull Request/Merge Request

You want a transparent mechanism for *Code Reviews* that facilitates *Automated Checks*, for a distributed team. This pattern describes an approach to facilitate code reviews that offer timely, relevant, and actionable feedback in a shared space, potentially including people who aren't in the same room or can't meet at the same time.



The Challenges of Remote Collaboration

When a change is ready for integration, you may want a *Code Review* from a team member. This should happen quickly to avoid delaying integration. A good *Code Review* process balances:

- Speed: Timely feedback so as not to slow down integration more than necessary,
- Quality of feedback: Relevant and actionable comments to improve the code.
- Flow: To minimize the cost of delays for the author, and context switches for the reviewer.

The easiest way to get feedback is to ask someone nearby to look at your code or perhaps to invite someone who is available to view the code by screen sharing. The most available person may not be the person who can give the best feedback, and the person who can give the best feedback may not be the most available.

Allowing for asynchronous feedback can address the availability issue; asking someone to review and comment "as soon as they can" can allow for a productive context switch, assuming that the delay isn't too long. Having time to prepare and comment thoughtfully can also improve quality, but handoffs and delays in conversation can negatively affect collaboration and productivity, You'd like to be able to get informed feedback 'quickly enough' without negatively impacting others on the team.

Scheduling times for review can minimize the cost of feedback as *Slow Productivity* suggests:

> *"A direct strategy for reducing collaboration overhead is to replace asynchronous communication with real-time conversations. … Arranging these con-*

versations, however, is tricky. There's a reason why the saying this meeting could have been an email has entrenched itself as a workplace meme in recent years. If every task generates its own meeting, you'll end up trading a crowded inbox for a calendar crowded with meetings–a fate that is arguably just as dire. … The right balance can be found in using office hours: regularly scheduled sessions for quick discussions that can be used to resolve many different issues."

While scheduled collaboration is a common pattern in agile methods (the Daily Scrum, the Sprint Review, etc), tasks get done at unpredictable times. So having one defined review time might be too few, but too many can lead to feeling of having a lack of solid focus time.

If you offer reviews a time to prepare for feedback, one option is to have each person download the branch locally, build it, and review it, which adds complexity to the process and limits collaboration possibilities.

An alternative, using a centralized mechanism like GitHub, which allows for browsing and comments, can simplify the browsing workflow, but the ability to comment can lead to batched, asynchronous review work.

For remote or distributed teams, mechanisms that depend on real-time, synchronous collaboration can be challenging.

The *Automated Checks* that a Continuous Integration Build offers can provide important insights into the state of the code and allow humans to focus on higher-level issues. You can do a risk analysis of the code based on certain heuristics (or at the judgment of the developer) to skip a review for trivial changes, as long as the *Automated Checks* pass. Having a way to ensure that these checks run before the human review starts is valuable: A consistent process to run *Automated Checks* and *Unit Tests*

in ant integration environment which can incorporate that feedback into the review provides a backstop for inadvertent errors, and provide an opportunity to catch errors before the code is merged.

Some regulated industries require some documentation of reviews, and you want to minimize the overhead of that documentation process. Unrelated to any compliance requirements, having data about the review process (timing, changes, etc.) might be useful for tuning the process and identifying process issues that you can address in a *Retrospective*.

You want to balance the speed, quality, and relevance of feedback.

Leverage Tools in an Adaptive Way

Therefore:

Establish a Pull Request Process that encourages collaboration and rapid feedback. Favor interactive reviews in a time frame that works for the team but allows for asynchrony when it reduces communication overhead without hurting communication. By default, avoid blocking Pull Requests for all but the most significant changes; trust team members to address issues or to reach out for clarification.

A Pull Request is a form *Code Review* done in a shared environment. This enables:

- Easier collaboration across locations.
- Automated checks to be run, with feedback shared in a common place.
- A place for team members to share feedback and propose specific changes.

The essential elements of a *Pull Request* are:

- A way to announce that code is “ready to merge into the Main Line.”
- A way to request feedback in the form of comments or proposed changes
- A way to respond to feedback through comments or code changes.
- A was for automated builds, and checks can run on the code before a change is integrated.

The following things are *not* essential elements of a *Pull Request*:

- Integration gated by approval: While approval can be part of the process, a sense of shared responsibility makes for a better delivery pipeline.
- Asynchronous comment threads: Asynchrony can be useful when there is a Time Zone difference between teams (for example, sharing end-of-day work for another team’s feedback might be useful), but the goal should be to integrate as quickly as possible.

While *Pull Requests* can be asynchronous, the comment process need not be; feedback can be collaborative, asynchronous, or a combination, depending on the team’s agreements. Pull Requests can be challenging when they introduce unnecessary delays. But not all delay is problematic, and some can be valuable. Consider the life of a pull request:

1. A Pull Request is opened
2. Automated checks are run
3. Team members review the code
4. Review conversation happens
5. Author makes code changes

6. Code is merged

Insert diagram pr-stream

The “review conversation” can happen:

- Immediately when complete: this optimizes for feedback speed but doesn’t allow for reflection or context switching.
- Scheduled review times: Much like the Daily Scrum is a way for a Scrum team to have a focused time to discuss how to collaborate, having an agreement about when reviews happen can reduce interruptions.
- Scheduled delay after code is complete. This is often an good balance between immediate and strictly scheduled, as it gives team members a chance to transition from other work.

Even with scheduled times, the team can always agree to exceptions when needed and fall back to immediate review,

The choice of approach depends on the team context. If a team is highly focused, “immediate” could work. “Scheduled delay” (“in 30 mins”) or even “every hour”) could be a challenge if team members find their time consumed with meetings outside the team, in which case reserving a block of time for everyone to do reviews might be the best choice, though excessive batching can introduce waste. A few times a day or “every two hours” can be a reasonable compromise.

Sometimes, you can combine the “Review code” and “Review conversation” steps. You want to have a pull request process that minimizes the length of the “review conversation.” To do this:

- Avoid asynchronous comments threads. "Pre-review" comments are useful to give the team a chance to understand the code, but avoid long asynchronous dialogs:
    - If comments are minimal and clear, trust the author to address comments
    - If discussion is needed, arrange for an interactive discussion time.
- Don't require a second round of reviews. After any discussion, trust the author to address comments appropriately.

A Pull Request can be a framework for interactive feedback that balances the value of interactivity and flow.

A *Pull Request* will be effective with certain practices in place:

- Working agreements about collaboration:
    - How quickly review should happen
    - Times when people will be available for collaborative reviews.
- Interactive Reviews:
    - After an initial review period, review changes together, ending with an action plan of what changes the developer will make before merging.
- Conditional Approvals for asynchronous comments.
    - When comments happen asynchronously, trust that the developer will address the concerns or reach out for clarification. Save "Request Changes" for significant errors. And ask to meet.
- Limit the use of gates to things like:
    - Failed Tests

    - Failed *Automated Checks* that are known to indicate high-severity issues,
- Keeping the change sets small to minimize the other factors that slow down *Code Review*.

The overarching criteria is whether the Pull Requests makes it difficult for the team to meet its integration goal. If merging code “within a day” is the goal, this probably means a feedback cycle of approximately 2 hours.

Example

Insert Flow Chart PR-Flow

Pull Requests are often associated with heavyweight processes involving gates and approvals. This isn’t necessary, but you can leverage the Pull Request Too to improve the quality of collaboration. Consider this flow:

- The developer pushes their changes to the central repository and opens a Pull Request
- The build pipeline runs a series of automated checks, including tests, style, and security checks. The style and security checks add annotations to the build. When the checks are complete, the build pipeline notifies the other team members that the code is available for feedback and suggests the team meet in 45 minutes.
- Team members review the code and make comments.
- If there are no substantial items to discuss:
    - The team members approve the Pull Request, and the meeting doesn’t happen.

    - Code is merged by the author after they review and address any feedback
- If there are items to discuss:
    - The team meets at the appointed time and discusses comments.
    - After the meeting the team approves the change
    - After the author addresses comments in the way that makes the most sense, they merge the change.

Cautions

Since Code Review in general, and Pull Requests in particular, are part of your collaboration dynamic, it's important to periodically check in on whether:

- The team is meeting its commitments to each other
- If the commitments are still valid

If you feel that a *Pull Request* processes take too long. Consider gathering metrics, such as "time from open to close", "time to first feedback," and potentially "number of comments" to add some data to present at a retrospective. Also consider impressions, since a mismatch between data and feeling could indicate that the guidelines might need to be revisited.

All the criteria that make for a good *Code Review* will help you to avoid the major issue with Pull Requests, in particular ones that seem to drag out.

- A major problem with *Pull Requests* is a lag in starting (and finishing) reviews. This can be due to reviews not being a priority. Having the right *Team Focus* can mitigate this

- *Feature Partners* can help ensure that the reviews focus on the right things and are quick, and that reviewers have the correct context.
- *Automated Checks* enable the humans on the process to higher level issues, leading syntax, static analysis, etc tools to catch the things they are best at.
- *Small Development Tasks* can ensure that the code being reviewed is an appropriate size to be reviewed quickly and that the code can be merged independent of other changes (atomicity).

Some of the features of a Pull Request that make sense in the context of open source projects, such as required approvals and gating reviews can slow down a team unnecessarily, and lead to downstream costs. You don't need to enable all of the gating features of a Pull Request if they don't add value to your workflow.

While getting feedback is often helpful, and synchronous conversations can mitigate many risks related to delays, keep the review process aligned with risk level, and be wary of falling into a habit of synchronous feedback. Some changes might not need a formal review, and sometimes, asynchronous feedback makes sense. However , long comment exchanges can be a sign of deeper collaboration issues. Continually evaluate the process.

Notes

Consider making PR gates contingent on risk, perhaps using a tool like Shepherdly , which can assign a risk score to a change

The origins of the term PR in the context of Open Source are important to understand (and that the same toolchain is used in product and project settings, though with a different workflow.

Feature Partner

When *Code Review* is part of your development process you want to make it easier for developers to get timely and relevant feedback on their code changes. This pattern describes a structure you can apply to your team to ensure that your code gets reviewed by someone with enough context .

Tasks and Assignments

In some teams, coding is primarily done by individuals. The timely, relevant, and actionable feedback that makes for a useful code review requires context, in particular team members who:

- Are available to give feedback close to the time the change is done
- Have the context of the code to understand the requirements and design decisions.
- Know the tech stack and the code base well enough to give reasonable feedback.

Another value of *Code Review*is knowledge sharing, but a pre-merge core review is not the time for revisiting the details of every decision.

In the interest of getting prompt feedback, you might consider asking the team at large for comments but:

- Not all team members will have the requirements and design context and thus may only be able to provide superficial feedback.
- Any individual might not prioritize giving feedback due to other commitments or because of a belief that someone else will get to it,

You might consider having a designated reviewer — a manager or someone with architectural oversight and a broad understanding of the project — review all changes. Having a senior person review all the code seems like it might be a good way to mitigate the risk related to lack of context and ensure uniformity of reviews, but a designated-reviewer approach:

- might become a bottleneck either for reviews or for design discussions.
- might not know the rationales for some decisions that were discussed at length, and revisiting non-critical issues during a review is frustrating at best and time-wasting at worst.
- creates a dynamic that removes some ownership from the development team.

Pair Programming might provide an opportunity to have a reviewer with all the context, but:

- Not all teams do pair programming well
- Since a pair is co-creating code, some changes might benefit from the perspective of a non-author.

Having someone familiar with the context, but not too close to the implementation available for feedback can lead to valuable insights, as Adam Alter says in *Anatomy of a Breakthrough*:

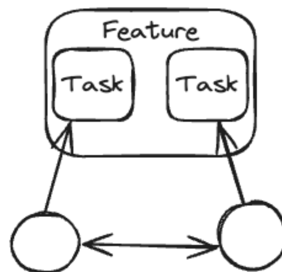
> *The more you consult with nonredundant outsiders, the more diverse your inputs, and the more likely you are to move beyond stubborn personal defaults.*

Ideally, you want feedback to be actionable and about improving the code, so it's often better to have someone with a day-to-day implementation role give input.

Designated Partner and Shared Responsibility

Therefore:

Start each set of development tasks by ensuring that at least two people are committed to delivering a feature. Even if there is a single primary developer, the *Feature Partner* will be involved in all key design discussions and responsible for providing prompt feedback



Defining the partner relationship explicitly clarifies that time spent reviewing and designing code in progress — even you are not writing — is a priority. This can also increase knowledge sharing on a deeper level.

Having a *Feature Partner* addresses three of the main challenges to good code review feedback:

- Availability for review: The Feature Partner shares responsibility with the primary developer for delivering a set of tasks related to a feature.
- Availability as a sounding board: The feature partner is jointly responsible for delivering a feature, so being available to discuss issues as they arise is part of the job.
- Context: The Feature Partner is engaged in the feature's planning and can be a good, high-bandwidth sounding board for questions during development.

Any team, an agile team, in particular, needs to be aligned on goals rather than individual achievements. As Gil Broza says in *The Human Side of Agile*:

> *First, the team truly needs to be a team. Everyone needs a common goal and interdependent commitments, so that everyone has the same long-term view and the feeling of being together in the same boat. Individual ownership, obligations, and objectives take away from being a team.*

Feature Partners can take many forms, including:

- Developers participating in Pair Programming (being mindful about switching pairs and groupthink)
- Individual developers working on different tasks related to the same story using the same tech stack, for example, two backend developers working on tasks related to a story.

- Developers are working on different ends of an integration (backend and mobile, for example) to deliver the same feature.

Whatever the structure, the essential requirements are that the Feature Partners are:

- Committed to working on a feature set.
- Can provide appropriate code review feedback to their feature partner.

During a development iteration, feature partners will collaborate to:

- Design code and interfaces.
- Be a sounding board for ideas.
- Meet with stakeholders to clarify questions about requirements and show work in progress.
- Review each other's code and unblock integration.

Using *Feature Partners* doesn't preclude self-organization; the team can (and should) decide what partnering approach makes sense during planning and who the *Feature Partners* are. The important thing is that the organization acknowledges that having two (or more) people associated with a work item makes sense in terms of delivery and quality and that each development iteration starts with acknowledging this.

While it's best that the *Feature Partners* actively develop code around a common theme, simply having someone committed to being a sounding board and resource can minimize development delays. Even having two people who commit to discussing and reviewing each other's code will be valuable.

Example

Implementing feature partners is as simple as acknowledging that at least two people will be associated with every backlog item or feature. You can determine this:

- During Sprint Planning
- When a backlog item is moved to In Progress

Cautions

Team members should have a role in identifying what they will be feature partners on – they should not be assigned top-down.

Having *Feature Partners* doesn't mean others can't participate in discussions. Others on the team can also give valuable feedback. Explaining a design or code to someone, regardless of their knowledge, can be helpful. A person with a new perspective can often provide out-of-the-box insights, and explaining can help the developer identify gaps. However, a *Code Review* might not be the best time for these conversations.

If you use Pair Programming, consider whether the co-creation nature of a pair could take away some of the "external perspective" value of a review and if that element will be problematic or not.

If the larger organization doesn't prioritize team goals over individual delivery, having *Feature Partners* can get pushback.

Next Steps

- *Feature Partner* depends on there being a *Team Focus*; a commitment to working on *features* rather than *tasks*.

Conclusion

Codelines are a central part of the software development process, and the right codelines can support your process and help your team thrive and deliver. Codelines don't exist in isolation, and a pattern language that shows how codelines fit in with other elements of the software eco system can help teams build robust approaches to codeline management.

References

Appendix

What a Branch Is

While there are a variety of branching models, the use of version management in software development is almost universal. There is a primary *code line* , the Main Line, which contains the history of all the changes to a project, allowing for the identifica-

tion of what changed and who changed it. When making changes to code, it's common practice to "checkout" a copy of the latest version of the Main Line into a development workspace. Between the time the work starts until it is integrated into the Main Line, the Workspace represents a parallel code path from the Main Line. A parallel code path is a *Branch*.

A Branch workstream can be code in a workspace that is:

- not backed by a version management system
- backed by a local to the workspace branch (used for local tracking, but never pushed to a shared repository)
- backed by a branch that is pushed to a shared repository.

Team software development works best when these parallel workstreams are integrated quickly and frequently into the Main Line.

This pattern language describes how to use shared branches effectively when developing software in an agile development environment.

Kinds of Branches

While the central element of the pattern language is a short lived Task Branch, there are a number of branch types that teams commonly used for various purposes. The table below describes some common branch types.

A Way to Think About Branches

Branch Type	Purpose	Lifetime	Contributors	Final Disposition
Main Line	Where all the code lives	For the life of the project	The whole team	Lives on
Task Branch	To work on a Development task	For the life of the task. A couple of days. No longer than a sprint.	1 Developer (primarily)	Merge to main after a Review. Delete
Feature Branch	To collaborate on a disruptive feature or proof of concept	Until the feature is done. 1-2 sprints. Shorter is better	A subset of the team	Merge to Main and delete once complete
Release Line	To capture a version for deployment, and to potentially enable hot fix type releases	Until the next version is deployed, or until this release is no longer supported.	The whole team contributes at the start	Archive once no longer needed

While each type is useful in the right context, this patternsl anguage focuses on short lived *Task Branches* that you integrate into the *Main Line* frequently.

Agile Software Development

This pattern language uses the term Agile Software Development to mean a process that is iterative, using short planning and development cycles, which allows for working software at the end of any development iteration. Some of the elements of an agile process are:

- Frequent and rapid feedback, for both the work being done and the process itself.
- Collaboration among team members and between team members and the business owner
- The use of techniques and tools to facilitate the feedback and collaboration process, with an emphasis on improving the quality of human interactions.

The engineering processes a team uses, including around code lines, should facilitate collaboration, with a focus on delivering value and reducing development overhead as much as possible.

Most agile processes are aligned with the principles of the Manifesto for Agile Software Development:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

Scrum and XP are two common approaches. The pattern language its method agnostic but assumes that:

- Planning and Development is done in iterations (iterations can be any length)
- There is some sort of plan in the form of a *backlog*
- The goal is to deliver work represented by a *Backlog Item* before the end of the iteration, if not more frequently.

Role of SCM

The book *Accelerate* places SCM as central to a key agile development practice: Continuous Delivery:

> *In order to implement continuous delivery, we must create the following foundations: Comprehensive configuration management*.

Even if your team doesn't delivery continuously to production, being able to deploy the latest code somewhere on demand is a key milestone on the path to being more agile.

Brief overview of the patterns including patterns that are not in this paper.