SCM Patterns Looking Back and Moving Forward

Steve Berczuk steve@berczuk.com

Riva Health

The Software Configuration Management Pattern Language had its origins in patterns workshopped at some of the first PLoP conferences and was published in the book *Software Configuration Management Patterns* in 2001. This paper provides an overview of the pattern language, discusses lessons learned as well as it's impact and durability, and explores some possible future directions for the pattern language.

Goals

The Software Configuration Management Pattern Language (SCM Patterns) had its origins in patterns workshopped at some of the first PLoP conferences, and *Streamed Lines* [2]. It was published in the book *Software Configuration Management Patterns* [3] in 2001.

This paper discusses the impact and durability of the SCM pattern language and explores some possible future extensions.

The overall goal of this paper is to identify:

- Ways to make the SCM Pattern Language more useful to more teams
- General lessons for authors of pattern languages to create more valuable, and approachable, pattern languages

This paper has three main parts:

- The origins and history of the Pattern Language.
- Lessons learned since the Pattern Language was published:
    - Feedback we received about the patterns over time.
    - Areas where the pattern language might have gotten stale.
    - Reasons why the pattern language is still relevant and useful.
- The pattern language:
    - An overview of the existing patten language.
    - A brief summary of possible additions and changes.

I also hope to make the case that The SCM Pattern Language is not just about how to build effective codelines, but also that codelines are the path that connects a backlog with a deliverable; I'm developing this extended SCM Pattern Language for a new book.

Origins of the SCM Pattern Language

This section discusses the history of the pattern language and the reasons we wrote it.

Configuration Management Patterns emerged from material that Brad Appleton and I — individually and collectively — workshopped at PLoP and during ChiliPLoP 1998 in Wickenburg AZ.

The *Software Configuration Management Patterns* Pattern Language is a guide to how to build codelines to facilitate frequent delivery. The pattern language is about creating codelines that facilitate frequent, iterative, collaboration — a good practice for any team, agile or not. I was motivated to document the patterns as I observed some teams struggling with basic release management, seemingly unaware of practices many other teams were using effectively.

The SCM Pattern Language has elements of an organizational/process pattern language rather than one about system design. Codelines sit at a nexus between Project Process (ie, Scrum), Development Process (using techniques like TDD, or DevOps), and Design, so that in addition to *branching* patterns the original pattern language includes patterns about:

- Testing
- Developer Workspaces

An effective SCM process helps a team focus on writing code that delivers value, rather than arcane details about branching techniques. Adopting a consistent process helps you to focus on answering hard questions, rather than day to day mechanics.

Since the original publication, there have been changes in the development ecosystem that lead to a need to update the SCM Pattern Language:

- Common Practice: CI/CD, DevOps, and Agile are widely known, if not practiced perfectly.
- SCM Tooling: SCM Tools implement some of the patterns directly.
- IDEs and CI Platforms, including AI extensions, make it easier to identify problems in the developer workspace.

Fit for Patterns

Software Configuration Management Patterns differs from other SCM “Best Practices” work:

- It uses the framework of a pattern language.
- It is tool and process framework agnostic.

The feedback from PLoP workshops gave us high confidence that the patterns we were writing were good candidates for patterns and not just “good ideas in pattern form” because:

- Each codeline element worked best in the context of other patterns, both codeline and process related. The pattern form makes context explicit and reflects the reality that a process lives in the context of other organizational practices.
- Successfully using the practices that the patterns describe seemed correlated with success in delivery.

Alternatives to an SCM Pattern Language

Branching and codeline process decisions are often based on history, trends, aspirations, or even fear. While there are certain things that qualify as “best practices” they often have a context.

Many branching strategies attempt to improve reliability by slowing down integration:

- Developers merge to a “develop” branch, which is only merged with the “Main Line” after some vetting or even after a release.
- The “Main Line” receives few merges.

For example, Git Flow[10] – a branching model that involves the use of feature branches and multiple primary branches might be a great approach for an open source project that has a variety of committers and a set of gatekeepers. For a small agile product team with a unified goal Git Flow can add risk and overhead. Someone might be using Git Flow because “it’s popular in open source,” ignoring the detail that the context for their project is different. SCM Patterns was meant to help teams find approaches to branching and delivery that aligned with their goals and development practices.

The SCM Pattern Languages creates structures that make it easier to merge to the Mainline quickly with short lived task branches, and structures that support creating an environment that enabled one to merge with confidence that the codeline would not break.

The goal of the pattern language is to help teams get past making branching process decisions based on intuition, risk aversion, or even hype, and to be able to follow a sustainable process. While the pattern language specifies some structure it is adaptable, unlike other more prescriptive approaches.

SCM Patterns in Brief (Original Pattern Language)

A brief overview of the Pattern Language will help set the context for the discussion.

The goal of the SCM pattern language is to help teams get past making branching process decisions based on intuition, risk aversion, or even hype, and to be able to follow a sustainable process.

This section has:

- A *diagram* of the pattern language.
- A summary of each pattern that includes:
  - The *name* of the pattern.
  - The *problem* the pattern addresses
  - A brief *description* of the pattern and the value it adds.
  - Optionally, brief *commentary* on its current relevance.

The diagram below shows the structure of the pattern language, including context relationships, as indicated by arrows. For example, the arrow from *Private Workspace* to *Integration Build* means that *Private Workspace* sets the context for *Integration Build*:

- You would want *Integration Build* if you’ve implemented *Private Workspace.*
- *Integration Build* supports the *Private Workspace* pattern.

The arrows can also indicate a sequence of how to approach the patterns in terms of goals and deliverables, so that the Pattern Language moves from high level to details.

Patterns which would be removed in an update are in light gray with ~~strikethrough~~ text. The rational for the change is noted in the summary commentary.

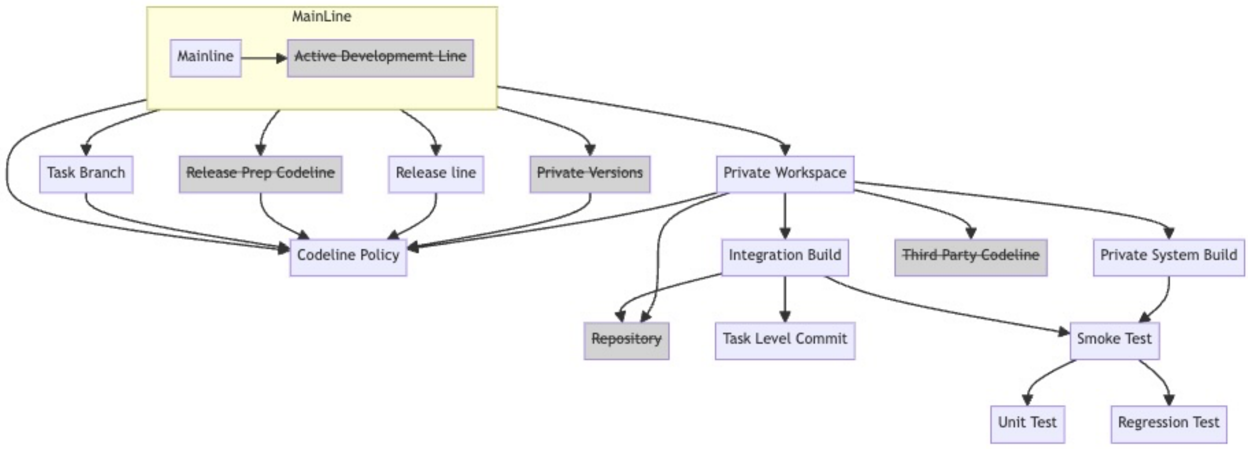

Diagram of the Pattern Language (2001)

Main Line

Keep the number of active code lines to a manageable set by maintaining a Mainline.

Main Line is similar to Trunk Based Development but doesn't preclude short lived branches. Mainline sets the context for all the other patterns. All codelines integrate with the Mainline (with possible exception of emergency fix branches off a *Release Line* and branches related to *Private Versioning*).

Main Line is still central to good SCM Practices. Develop processes to integrate all work to the main line as quickly as possible, while maintaining the Mainline in a working state — using the patterns in the rest of the pattern language. Even if you were to never branch, the workspace, build, and change/commit patterns would apply.

Active Development Line (removed: redundant)

Keep a rapidly evolving Main Line *stable by creating an* Active Development Line.

An Active Development line is a *Mainline* that always works, so this is redundant.

Active Development Line was an artifact of getting overenthusiastic about decomposition of patterns. These two patterns are grouped together in the diagram for that reason.

Private Workspace

Prevent integration issues from distracting you and from your changes causing others problems when working with a Main Line *by developing in a* Private Workspace.

Consistent build, test, and execution environments are essential to agile development.

Private Workspace is still relevant, though much simpler to implement given the availability of better dependency and package management tools, as well as containers.

Repository (removed: trivial)

Set up a new Private Workspace *or* Integration Build *Workspace by populating it from a* Repository *that contains everything that you need.*

A Repository enables creation of a *Private Workspace* as well as an *Integration Build*.

In practice Maven or PyPy Repositories, container repositories, and other (for example, the Go Module System) with a source code repository make this trivial.

Private System Build

Check to see that changes you make in your Private Workspace *will not break the build by doing a* Private System Build *before committing changes to the* Repository.

This seems basic, though I still encounter scenarios where people use a pull request CI build to build and test. Local builds and unit tests improve cycle time. The key is to make it easy to run, for example a Makefile target or simple script execution.

Private System Build and *Private Workspace* are so closely related they could be merged.

Integration Build

Ensure that your code base always builds reliably, and that your Private Workspace *is consistent with general expectations, by doing an* Integration Build *periodically*.

Agile, Continuous Integration/ Deployment were gaining traction as ideas at about the time that SCM Patterns book was published. Those concepts require a reliable *Integration Build*.

An *Integration Build* is central to a CI process. The variety of tools that can support this -- from Git Hub Actions to systems like Jenkins -- point to the importance of the pattern.

Third Party Code Line (removed: trivial in many cases)

Manage vendor code in your Private Workspace *by using a* Third Party Code Line.

This pattern describes how to manage custom changes to a third-party module.

For open-source projects (and even some commercial libraries) in public GitHub repositories, this is straightforward to do by using mechanisms like forks.

Task Level Commit

Simplify the way you manage changes to your Integration Build *by organizing source code changes by task-oriented units of work.*

To commit often with coherent unit of work and clear commit messages depends on the development work in collaboration with Product and task planning, and tools to manage the incremental rollout of features (like feature flags).

A *Task Level Commit* could be superseded by *Task Branch*

Codeline Policy

When you have a number of codeline types, create a Codeline Policy *to help developers decide what procedures to follow before a checkin on each code line.*

This is simply "say what the codeline is for and enforce it." Repositories like GitHub make this straightforward.

The pattern is still useful, and some patterns to support it could be valuable.

Smoke Test

Ensure that the system still works after you make a change by running a Smoke Test as part of your Private System Build *or* Integration Build.

Smoke Test describes a basic automated integration test that covers basic functionality. A *Smoke Test* supplements more complete testing.

A *Smoke Test* still makes sense, but could be merged with *Integration Test.*

Unit Test

A Smoke Test *is broad. A* Unit Test *is a deeper way to verify that a module after a change.*

Unit tests are an essential part of how you keep a code line active and agile. Ignoring semantic debates about what a “unit” is, key factors are speed and isolation. Container based testing blurs some of the lines between unit and integration testing, but it’s still a unit test because it does not touch other running systems.

Unit tests are very relevant to the pattern language as they can help with refactoring and the development of a *Modular Architecture*, as *Testable Code* often has less coupling.

Regression Test

A Smoke Test *is a test of basic functionality. Additional tests like a* Regression Test *ensure that existing code doesn’t get worse as you make other improvements.*

Like *Smoke Test,* a Regression *Test* is an integration test that gives you confidence that you didn’t break anything.

A *Regression Test* is useful but could be redundant with *Integration* and *Unit* Testing.

Private Versioning (removed: implicit)

When you work off a Mainline *you may want to explore non-trivial changes. Use* Private Versioning *to allow you to experiment with complex changes locally, yet still be able to take advantage of the features of a version control system.*

A DVCS like Git gives you this implicitly by letting you create branches and versions without pushing to the server, so this pattern is trivial in a DVCS. In Git *Private Versions* is essentially the following:

```
git checkout -b task_branch # Create a task branch
git push origin task_branch # do some and push changes for safety
git checkout -b private_version # create a private version to explore options
git checkout task_branch && git merge origin private_version # merge the work
git branch -d private_version # delete the private version branch
```

With a centralized VCS like subversion the process is more complicated, hence the value of a pattern.

Release Line

When you have a Mainline *you may still need to deploy a patch to released software. Maintain released versions without interfering with your current development by establishing a* Release Line.

A *Release Line* is the one long lived branch that you might want to have. This is where you can do emergency path releases if you can’t incorporate changes into a new release.

Release Lines, or at least tags are still useful for history even if you deliver off the *Main Line*.

Release-Prep Code Line (removed: not recommended)

Stabilize a code line for an upcoming release while also allowing new work to continue on active code lines by doing the stabilization work on a Release-Prep Code Line.

At the time of the book's writing, daily builds were a goal and code freezes were not uncommon. *Release Prep Code Line* was a way to avoid a code freeze and allow code to merge to the *Main Line* as you stabilize a release candidate.

This pattern is unnecessary with Continuous Integration; if you need to "stabilize" your code before release, you need to improve your test automation.

Task Branch

To keep the Main Line *active, isolate potential disruptive changes on a* Task Branch.

Task Branch was about collaboration on a subtask. It can also cover work by one person.

A *Task Branch* is common in practice with an emphasis on short-lived *Task Branches*.

Feedback

Since the patterns and the pattern language was published, we've gotten both positive and negative feedback on its usefulness:

The positive:

- The Pattern Language captured structures that continue to be useful and helped teams improve their process.
- The Pattern Language helps teams which struggle with complex and arcane branching strategies; since it's not explicitly agile, it can be more accessible.
- The SCM Pattern Language puts the mechanics of branching into context and helps teams to use branching more thoughtfully.

We also got feedback that, even though the patterns *are* well known practices, having them in a book helped internal change agents implement process improvements they were seeking. This was both because the presentation was useful, and because having an external reference can (unfortunately) add credibility to an argument — a well-known consulting cliche.

The less positive feedback from comments and Amazon Reviews suggested:

- The Pattern Language is too abstract and tool agnostic to be actionable.
- The ideas in the patterns aren't novel; "everyone knows them."
- The format was distracting.

Following sections discuss the feedback with a focus on the improvements.

Novelty

At an early PLoP conference (2000?) Walter Tichy, a key person in the development of version management systems, commented that there was "nothing new" in SCM patterns. Some Amazon reviews had a similar theme.

This issue comes to the main point of patterns. In *The Timeless Way of Building* [1] Christopher Alexander, speaking of the process of building says (p 13)

> *Although the process is precise and can be defined in exact scientific terms, finally it becomes valuable, not so much because it shows us things we don't know, but instead, because it shows us what we know already, only daren't admit because it seems so childish, and so primitive.*

The point of the patterns isn't novelty but to make the approach clear to those who aren't yet following the process. And in the case of the SCM Patterns, there are *many* teams that are not following the process even though it is likely to improve their delivery. The point of each pattern isn't just to enumerate a set of good practices, but rather to show when and how to apply the practice, or to provide:

> *A solution to a problem in a context*

New ideas are interesting, and useful, but there are many cases where there are already solutions that work, we just need to know how to apply them.

For example, the mechanics of a *Task Branch* are easy. Branching is a basic operation of SCM tools. The challenges teams have are knowing:

- How to reduce the chance of merge conflicts when it comes to merge to the main line (the pattern describes some rules and mechanics such as:
  - Keeping the branch short.
  - Pulling changes from main periodically.
- How to reduce the risk that code will break the main line. (Mechanisms like unit tests, and integration workspaces).

The value of the SCM Pattern Language over an SCM tool primer is that it puts the operation (in this case creating a branch) in the context of other SCM operations and the development process.

One could condense the entire pattern language to a very short sentence:

> *"use fewer code lines, make them short, and test"*

but that would leave out some nuance and give the user no guidance.

An update to the pattern language could:

- Frame the patterns in a way that makes the value of lack of novelty clearer for those who are not patterns (or domain) experts.
- Set the patterns in a wider organizational context.

The challenges with these changes are:

- Not spending too much time on what the Pattern Language is *not*; focusing on negatives can be distracting.
- Keeping the focus narrow enough so that the work doesn't become confusing or talk about things beyond my expertise. Leveraging other patterns and pattern languages will be important, while placing the patterns in the SCM Context.
- Framing the patterns as "Best Practices" could address the novelty issue, but patterns are more than "Best Practices" — they involve context.

Format

The format of the patterns was inspired by Christopher Alexander's Pattern Languages. Calling out the context, problem, solution, and details helped ensure that the patterns were more than just a list of things to do, re-enforcing their interrelatedness. An image with each pattern was meant to make a memorable connection.

Some readers unfamiliar with Alexander's pattern form found the format superfluous -- if not distracting; they would have preferred a more structured format with headings rather than typographically delineated sections and thought that the picture could be skipped or at least made less metaphorical and more relevant. Others thought that the format, including the images, helped enable connections, and made the work flow better.

While the format helped enforce a way of thinking, adhering to each detail may not always make sense. This is analogous to the User Story template:

> As a *user* I want *do something* so that *goal*

While this a good model for user stories and other requirements, being pedantic about the structure can make for awkward reading. The important part is having all the elements.

Making the pattern language more approachable while retaining the context, problem, solution, elements has merit. I can see abandoning images that are too metaphorical if there are more domain specific images or diagrams to use, and adding some structure that make the language easier to navigate.

Actionability

The pattern format and the tool-agnostic approach kept the patterns more universal. The lack of tool-specific detail, however, meant that the patterns weren't as actionable as other resources with code or command examples.

While we considered adding more detailed examples, the tool landscape was changing quickly, and too many specific examples might have made the book seem dated more quickly. We also wanted the book to have a more wholistic focus; the details of how to create a given structure might vary slightly depending on the details of your tool and situation. We also didn't want to be a tool reference — there are better resources for that, and building on existing resources is consistent with the Patterns Way.

The value of a software pattern language is less the specifics of how to implement the individual patterns — the solutions can vary — but rather in knowing *why* and *when* to use them. This is very relevant in a teaching context, where people are learning about SCM process and often tooling at the same time. One element that could be added is more guidance about how to encourage adoption of the patterns in the face of resistance.

Blog posts and other articles helped to bridge the gap, which seems appropriate as patterns are living documents. However, the patterns could provide more implementation guidance.

Adoption and Impact

It's hard to *know* whether the patterns had an impact on the way teams worked, or whether they simply captured emerging adoption. Many of the SCM Patterns are reflected in universal practice, though again, it's hard to say how much SCM Patterns captured the evolving consensus vs having an influence.

The patterns we captured *seem to have been* central to what people call good development process:

- Some of the patterns are now built in to modern SCM Tools.
- The Patterns are part of the common SCM and process vocabulary.
- Processes that use fewer, shorter, branches are considered valuable in Agile and other widely used development processes.

Since the SCM patterns span the software development lifecycle, it isn't clear how much the SCM Patterns influenced these changes v capturing trends, v supporting teams in their journey. It probably doesn't matter as a Pattern Language; it captured good practices.

As a source of known good practices SCM Pattern have been a resource for those trying to improve their development process —having an external resource to refer to sometimes adds credibility to an argument for a simpler process — even when there is an internal expert who *does* know these things.

An SCM Pattern language is valuable if it can help the following groups implement codelines that support faster, more reliable, delivery.

- Developers on small teams looking to set up a good process.
- Evangelists/advocates for better process in larger organizations looking for better ways to make their case for more agile codelines.

Tool Integration

When patterns are valuable people tend to develop the tools to make them easier to implement, and in some cases, take for granted. In the case of *Design Patterns*[7], programming frameworks and languages began to incorporate certain patterns, making their use less about writing code and more about selecting the appropriate language or framework element. For example, a pattern like *Singleton* is an integral part of adependency injection framework like Spring, though one rarely defines Singletons explicitly. Other GoF Patterns are trivial to code. The "Pattern" still applies in the choice to use the language element.

Similarly, for SCM Patterns model tools make some patterns trivial to realize:

- *Private Versioning*, being able to use SCM mechanisms to do experiments, is as simple as making a local branch using a command like `git branch`.
- *Codeline Policy*, the rules that govern a code line, can be implemented by GitHub actions, or GitHub checks in addition to more cumbersome pre and post commit hooks.
- *Repository* which is about making it easy to retrieve the correct version of requirements is trivial to implement given the prevalence of Package Managers, Containers, Artifact Repositories, etc.
- *Private Workspace* and *Private System Build* are greatly simplified by module systems (rpm, go.mod, Python Virtual Environments). Containers and package managers make it straightforward to get the right version of tools. The only challenge would be using a language which relies on system (v project level) library dependencies.
- *Task Level Commit* is simplified by tooling that identifies issue identifiers in a commit message, associating the commit with a development task.

What tools don't do is to directly support the intent of the patterns. While tools can help enforce a *Codeline Policy* the details of what the *Task Branch* policy vary, and really all you can do in that case is inform, though it's conceivable to imagine a system that lets you follow a sequence that lets you:

- Identify a branch as a task branch.
- Warns you when the branch has been open for a longer than expected time.

As I discuss in the section on future directions, *how* to define the task is the harder problem that identifying the commit.

Vocabulary

The SCM Patterns — like other patterns — provide an unambiguous language for talking about the problems space. The Pattern Language uses term like Release Line, and Main Line that were in wide use, and the pattern language provides a reference for how to use the words consistently.

In terms of evolving language, Mainline, used in the Pattern Language to denote the "source branch" sets a usage pattern to use *main* to replace *Master* as the default branch name in git and other tools.

Use of the Concepts

Continuous integration is something that you'd expect at some level on any project, and while continuous deployment might be less common, *more frequent* deployment is often a goal. Simpler branching models that use fewer code lines are widely discussed. Examples include "Trunk Based Development" and "GitHub Flow" both of which reify the basic structure the pattern language describes.

CI Environments make it easy to implement automated checks, such as *Unit Tests* which are essential to maintaining a working main line.

The themes of SCM Patterns are widely used, if not the patterns themselves. Teams realize that there is value in moving more quickly. The challenge is in agreeing on *how much* more quickly and *how*.

Extensions

The original pattern language was focused on the mechanics of branching and development:

- Branching Patterns
- Workspace Patterns
- Testing Patterns

Much of the pattern language can be implemented at the team level, but organizational constraints often form obstacles to fully implementing the pattern language:

- Adding the testing and infrastructure takes time and much be planned for.
- Task branches, Task Level Commits, and Code Review require some level of planning and prioritization.

Changing practices, tool, and team work styles, as well as experience, lead me to realize that an SCM Pattern Language could benefit from some additional scope. This is to:

- More fully reflect the role that codelines play as a "path" that work travels from a backlog item to working software.
- Recognize that SCM exists in and interacts with the planning and team environment.
- Acknowledge that "doing the mechanics" isn't enough to have a successful SCM process.

An updated SCM Pattern Language should include:

- Planning steps that raise visibility of smaller deliverables and the role of planning in setting the release cadence.

- Architecture that enables more incremental delivery.
- Organizational and Cultural elements that allow for change, including the importance of Psychological Safety.

In particular, this means patterns that cover:

- Human Feedback — peer feedback in general and Pull Requests in particular.
- Automated Feedback to help you get the most value from human feedback and still move quickly.
- Architecture Patterns to support incremental work and enable testing.
- Planning since the "small units of work" that are central to SCM Patterns start with the Product and Sprint backlog.

Many of the planning related items are addressed in Scrum Patterns, in particular those in *A Scrum Book*[8].

Distributed teams are another consideration that could span the pattern language, since remote and hybrid teams are common and SCM tools *can* facilitate collaboration.

SCM Patterns put SCM mechanics in context with other aspects of development with the goal of being a guide to introducing SCM Process to new developers, or developers who were unfamiliar with the impact of SCM on product delivery.

Structure of Patterns

In addition to better connecting the SCM Patterns with the planning and organization ecosystem, a revised pattern language would also be more actionable:

- Reframe the patterns into a more actionable format: Keeping the pattern elements but making sure that they add value, and consider simplifying the presentation so that format seems less distracting.
- Connect the SCM Patterns more closely to the ecosystem (feedback, planning, etc)

The next section describes some candidate additions to the SCM Pattern Language in capsule form

Some Possible New Patterns

This section presents summaries of patterns that could add to the current pattern language.

Code Review

Unit Tests *and similar checks can verify functionality, but can't give guidance as to design intent. You need a way to get feedback on whether the code does what it's supposed to do, that the test tests the right things, as well as an opportunity to share design insights.*

A *Code Review* by a peer developer can provide that kind of feedback if it can be done in a timely manner.

Automated Checks

A Code Review *is valuable, but also time consuming. Time spent on code review should focus on things that people do best and add the most value. Certain things like style, security risk analysis, and the like are still important.*

Add *Automated Checks* to your build pipeline to address style, formatting, issues with dependencies, etc. Ideally these checks can run both locally and in a CI environment, and these checks should be evaluated in the context of a Code Review

---

Pull Request

You need a time and place to do a Code Review *You want a tool that supports collaboration and access to feedback from* Automated Checks.

A *Pull Request* is a useful mechanism to identify when code is ready for feedback and provide tools to give the feedback, synchronously or asynchronously. Structure Pull Requests that balance speed and value, and avoid the delays that people experience with poorly implemented PR processes.

Actionable Alerting

When working on a Main Line *you acknowledge that moving more slowly won't provide security from errors that's greater than lost opportunity cost. You need a way to identify errors that pass through despite your best efforts.*

Provide *Actionable Alerting* to running systems to identify (and possibly predict) errors or other issues worth investigation so that they can be fixed promptly.

Retrospective Culture

Working on a Main Line *requires that you always inspect how you work so that you can improve. You want a way to continually review your process.*

Retrospectives are a useful tool for doing periodic review. A *Retrospective Culture* goes beyond the mechanics of meetings to describe a mindset of continuous improvement and review

Psychological Safety

A Retrospective Culture *can only work if people can be honest and open about issues. How can you make it possible for people to share the information that he team needs to improve?*

Creating a environment that supports *Psychological Safety* enables open, blameless dialog

This is similar to the Community of Trust Organizational Pattern [9]

Team Focus

A Main Line works best when work gets integrated quickly. Sometimes integrating work requires collaboration among team members. How can you help a team move work forward?

One challenge to rapid integration to a Main Line is mult- tasking: someone needs help, or a review, and can't get it because others are working on other items. Start each development cycle with a *Team Focus* that gives everyone context, and sets a priority on finishing work in progress.

This is closely related to *Sprint Goal* (#71 in Scrum Patterns [9])

Product Backlog Item

Given a Team Focus how do you define what to do?

The *Product Backlog* and the Items on it should be constructed in a way that aligns with the focus of the team for a development cycle.

This is a placeholder for *Product Backlog Item* #55 in Scrum Patterns [9]

Small Development Task

How do you organize work relating to a Product Backlog Item?

Each *Product Backlog Item* should map to one or more *Small Development Tasks*. These tasks should be completable in a timely manner (1 day is typical). Small Dev tasks also speed the rest of the process (testing, review etc)

This is similar, if not identical to *Sprint Backlog Item* (#73 in Scrum Patterns [9])

Feature Flags

A Small Development Task might not be ready for production use. How do you allow for integration (and testing) without exposing end users to partial work in progress, and not holding up work on branches?

Use a *Feature Flag* to isolate new in progress functionally. (This pattern talks about good practices and other alternatives.)

A Feature Flag is a mechanism to turn functionality on and off based on (deploy time or user) configuration.

Modular Architecture

Blocks of code isolated by Feature Flags *can be hard to manage if they are not small.. How can you help support work that can be done in an isolated manner?*

A *Modular Architecture*, minimizing dependencies between components is valuable for limiting the scope of work related to a feature.

Diagram of a Revised Pattern Language

The following diagram shows an updated pattern language. Patterns from the original Pattern Language are in *italics*. Patterns based on Scrum Patterns are in Yellow, and those based on Organization Patterns are in Orange.

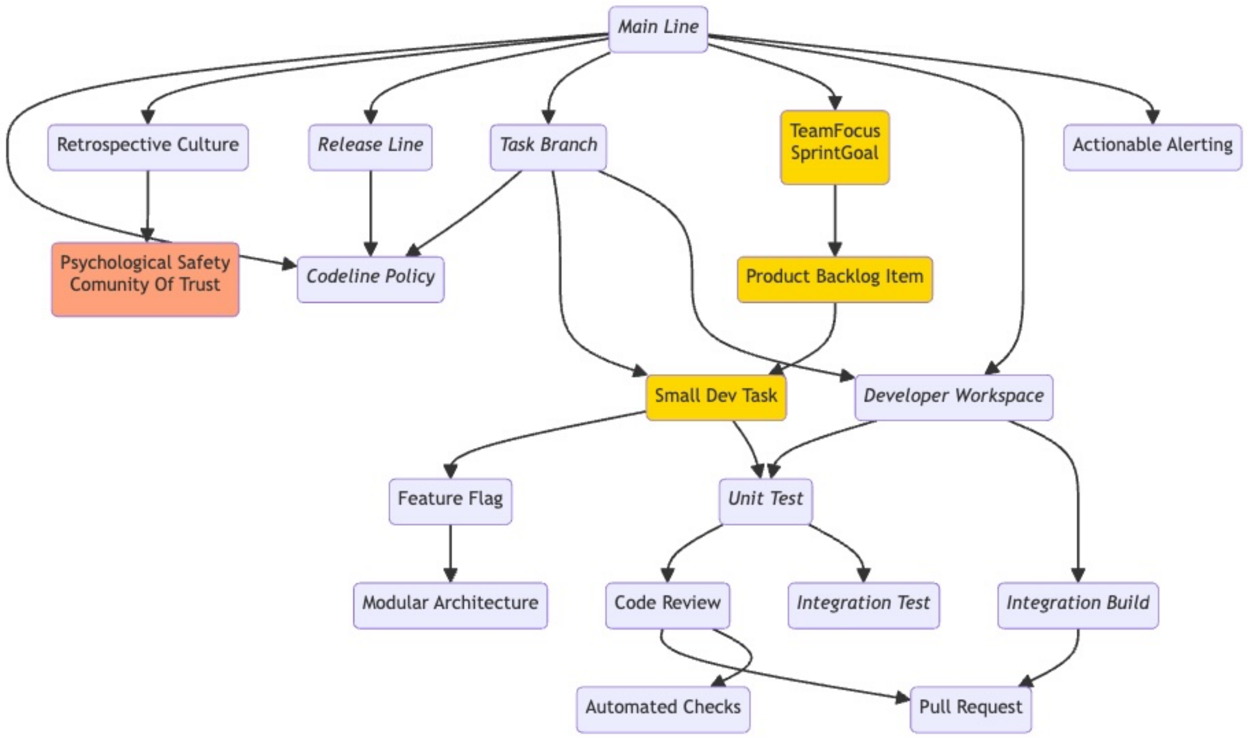

Diagram of a Revised Pattern Language

SCM Patterns in a Trunk Based World

The biggest push back against SCM Patterns I've heard is in the context of arguments of no-branching trunk based development. But even in that context, the SCM Patterns can be a useful guide in that the patterns that support rapid feedback can support any number of codelines.

A development process with branching and pull requests is often cited as a factor that slows down delivery because these steps cause delays in integration. The issue isn't simply isolated work; every process has work being done at some level of isolation of some amount of time. The issue is time to merge, and with that the circumstances that delay integration.

The answer isn't to never branches or use pull requests, but rather to keep the reasons for these structures in mind and do then *when they add value* and to *do them in a way consistent with good patterns.*

Even without branching, the rest of the patterns in the language support maintaining a working mainline (even one that developers commit directly to).

The SCM Patterns are still relevant because:

- Teams still struggle keeping working mainlines that they can deliver. Even without any branching the patterns provide a guide to keeping your single codeline working.
- Distributed and fully remote teams, and the reality of asynchronous work, make the right use of branching more useful.
- The Patterns form a guide to (eventually) removing the long lived branch structures that Trunk Based Development and similar approaches seek to eliminate.

Conclusion

While some of the patterns in SCM Patterns are either irrelevant or given current software development approaches and tooling counterproductive (as noted in the summary of the patterns), most of the core patterns are still very valid, and adding the process and architecture context should make them more so.

References

[1] C. Alexander, The timeless way of building, 8th printing. New York: Oxford Univ. Pr, 1979.

[2] B. Appleton, S. Berczuk, R. Cabrera, and R. Orenstein, "Streamed Lines : Branching Patterns for Parallel Software Development," in PLoP conf., 1998, pp. 98–25. [Online]. Available: https://www.bradapp.com/acme/branching/

[3] S. P. Berczuk and B. Appleton, Software configuration management patterns: effective teamwork, practical integration. in The software patterns series. Boston: Addison-Wesley, 2003.

[4] S. Berczuk, "Reliable Codelines," presented at the Pattern Languages of Programs 2001, Monticello, IL, Sep. 2001. [Online]. Available: https://www.hillside.net/plop/plop2001/accepted_submissions/PLoP2001/sberczuk0/PLoP2001_sberczuk0_3.pdf

[5] S. Berczuk, Appleton, Brad, and R. Cabrera, "Getting Ready to Work: Patterns for a Developer's Workspace," presented at the Pattern Languages of Programs 2000, Monticello, IL, Sep. 200AD. [Online]. Available: https://www.hillside.net/plop/plop2k/proceedings/Berczuk/Berczuk.pdf

[6] M. Cohn, User stories applied: for agile software development. in Addison-Wesley signature series. Boston: Addison-Wesley, 2004.

[7] E. Gamma, Ed., Design patterns: elements of reusable object-oriented software. in Addison-Wesley professional computing series. Reading, Mass: Addison-Wesley, 1995.

[8] J. Sutherland and J. O. Coplien, A Scrum book: the spirit of the game. in The pragmatic programmers. Raleigh, North Carolina: The Pragmatic Bookshelf, 2019.

[9] J. O. Coplien and N. Harrison, Organizational patterns of agile software development. Upper Saddle River, NJ: Pearson Prentice Hall, 2005.

[10] Atlassian, GitFlow Workflow https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

Related Sources

- PLoP Papers related to the SCM Patterns:
  - Software Reconstruction: Patterns for Reproducing Software Build in Proceedings of the 1999 Pattern Languages of Programming Conference — With Ralph Cabrera and Brad Appleton
  - Reliable Codelines in The 2001 Pattern Languages of Programs (PLoP) conference
  - Configuration Management Patterns in 1996 Pattern Languages of Programs on July 1997