A Guide to Branching and Merging Patterns
Executive Summary

Software configuration management (SCM) practices are at the forefront of managing a process for a development team. Choosing the correct branching pattern can either make a good development team great, or cause confusion and pain for the development team. Understanding how branching and merging patterns work and applying them to your projects have a profound effect on how development teams deliver software.

Gartner analyst Sean Kenefick says “Mostly folks need to improve processes. The tools are strong; they do what they are meant to do for the most part. It’s how you use the tools that’s the real problem.”

Introduction

Modern software development practices are centered on delivering code quickly and getting rapid feedback. Teams doing Agile, XP or a hybrid approach, have customers demanding that changes are delivered immediately. There is no room for failure in this type of environment.

Two-thirds of all software projects fail, according to the Standish Group’s CHAOS study. Improper usage of software configuration management (SCM) is largely to blame. After project management, IT users cite configuration management as the process that most needs improvement, according to Anne Hass in Configuration Management Principles and Practice.

It’s no wonder that SCM best practices with branching and merging patterns have a profound effect on the everyday lives of developers. When teams choose a pattern they can choose between traditional techniques that have been developed over years of Waterfall-based projects or they can use workflow-based patterns that can enhance modern software development practices such as Scrum and other Agile methods.

Reasons People Create Branches

Development teams often create a branching pattern, usually drawn out on a white board or in a Visio document, that is used as a model to follow for the overall development process. While this is all well and good, many times these well-laid plans become quite complex for reasons that can’t be foreseen.

As teams scale, the most common reason for a branch is to support parallel development. This means that many different teams working on different features or projects in the same codebase will have to “fork” a version of the software in order to keep working on the same code base while preserving their changes.

It’s often claimed that parallel development increases a team’s productivity and performance, but that isn’t the only reason to do parallel development. Branches are created for many needs.

Some of these types of needs might be:

- **Maintenance release**—A branch created for a previous patch release of the software
- **Customer specials**—A special version of the software just for one customer
- **Development branches**—Developer branches to test code not ready for QA
- **Code configurations**—Different configurations of the code based on an environment, such as UAT, QA, etc.
- **Special content**—Content that might not fit under the same code structure as source code, such as images, db, binaries or static content

Business needs are often the reason for branching. If we take a more philosophical view of what branches represent, they are actually workflows for different aspects of the software development process that go beyond regular parallel development practices.

The Goals of a Branching Pattern

The goal of a branching pattern should always be to manage the software team’s development process and make it easy and straightforward to follow all of the things that need to happen in order to release a piece of software.
Make It Easy and Straightforward to Merge Changes
Branching patterns should make it easy for developers and team members to move and integrate code between different branches. It should be easy and straight-forward to know where a change belongs at any time.

But with more complex environments, understanding how the branching and merging process can become complex quickly. Teams often create spreadsheets and white board diagrams to solve this problem. Unfortunately, scaling change management on a white board isn’t a sustainable option.

A straightforward pattern should give development teams the ability to know where to merge to and from, and when to push changes from one branch to another.

Provide Private Areas for Teams to Check-In and Integrate
Committing early and often is an SCM best practice. Over the years, developers have been told that if it’s not in source control, it never happened. Typically many teams require people to check code in every day, so that work isn’t lost.

A practical branching pattern will allow teams and developers to create private workspaces and branches to allow them to create builds, releases and tests of code before they push those changes to other team members.

Manage Distributed Teams
Collaborating and sharing code with distributed teams is more complex than ever. Teams routinely perform software development in many locations and sometimes test or perform other tasks in another location. This distribution of teams strains the development process. There are security auditing and integration problems throughout the process.

Development teams must appear to be co-located while utilizing the same process with lower complexity. This means code integrations with other projects should happen in real time, so teams can give each other feedback immediately. The branching pattern and process must be visible to everyone, to ensure they are all on the same page.

Understand What’s Been Delivered
User stories, bugs, and requirements drive any process, and what is often overlooked is the ability to see what changes match these items and the location of those code changes in the branching structure. Planning tools are an excellent way to create and assign issues to team members, but this is only the first step in the process. When an issue is in a development branch, there are active changes being made against that code. The ability to see which code changes have been delivered to a release and which files and history match with them gives full visibility into the process.

Traditional Branching Patterns
Traditional SCM branching patterns are designed around decades-old software methodologies. Mapping them to modern practices only hinder and get in the way of modern software development processes, such as Agile or iterative development.

On the following page is an example of a basic SCM branching pattern. This pattern is usually created by development teams because it is an easy and straightforward type of structure to understand. The benefits are that developers can work in their own developer branches while teams can isolate changes for other releases and projects.

These basic branching patterns are often created out of a need for a mix of private project branches, developer branches, the current release or a special release sometimes used for a patch or customer. These types of branches are really created from a need perspective, meaning they are not designed around the software development process workflow, but based on whatever pressing business need is the issue of the moment.

The problem with this reactive type of branching model is that the level of coordination and planning becomes a burden for software teams to handle. Everyone must work on a single baseline, and having too many teams or projects sharing that code base slows progress and only encourages teams to work in isolation for fear of breaking the code for other teams.

Naturally, a team’s codebase is in flux for most of the development cycle, usually becoming more stable near the end of a release. There is no clear way to separate “finished” code from code that is currently ready for testing, UAT (user acceptance testing) or production. All of these separate pieces of code are mixed in together. This makes it difficult for teams to test code immediately, without spending lengthy amounts of time on separate branches.

This “baseline pollution” hinders development teams by making it difficult to merge code from one team or project, and integrate and test those code changes with other teams(s). Validating this work is error prone and manual. Trying to do this with a time-consuming merge process can bring projects to a screeching halt.
Intermediate Branching Pattern
An intermediate branching pattern allows for teams to push code for two releases while maintaining all of the branches for projects, developers and separate releases. This again appears to be an easy and straightforward design that allows for these teams to deliver change to the appropriate project.

While this design does solve the problem of multiple release lines that need to be maintained, it quickly creates a more complex model, often increasing the amount of conflicts and merge problems that can occur, despite its apparent straightforward design.

Traditional SCM branching patterns such as these models require developers and teams to manually indicate the linkage between process, issues and code when it’s moved between the branches or checked into the repository. At the end of each development cycle, teams are faced with the task of determining which issues are fully completed and which issues are partially done and need to be retargeted to the next release. Rooting through commit logs and comment fields is a tedious process that doesn’t scale with modern development practices.
Promotional-Based Pattern for Modern Practices

Promotional-based branching patterns differ from the traditional branching patterns because of their ability to map to a development process instead of just a release or project.

Philosophically, promotional branches can be analogous to the different states of the development cycle. Similar to how Issue Tracking Systems (ITS) move issues through each state, code can belong in different states also. Code might move through different statuses while making its way to production. Teams will start out in development and move the code to QA, UAT, and eventually to production. Along the way, there are processes and policies in place that the code must align with before it moves to each stage.

Promotional patterns are integration points, where transfers of roles, responsibilities and code integrations can be managed at each stage of the promotional hierarchy. This allows the code to grow more stable as it moves up the hierarchy.

Each branch is based on the previous branch in the hierarchy, meaning that if the parent branch changes, those changes will push down to the lower levels. This allows for easier and more frequent code integrations.
Below is an example of an Agile promotional model where issues are taken off the backlog then moved through a branch hierarchy from left to right. Each piece of code that matches with a user story stops at a stage of the development cycle while on its way to a finished product (wip, coded, tested, done). These branches separate code by state, allowing teams to pull environments and builds at any stage of the cycle and eventually only pushing finished production. This avoids the problem of "baseline pollution."

**Promotional-Based Pattern Implementations and Benefits**

Promotional-based patterns allow teams to deliver faster, and address many of the core problems that traditional patterns have had for years.

- Promotion levels with change packages can be readily identified and traced.
- Staged promotion branches function like a progression of gates, transferring changes and files with reliable and reproducible processes that teams can use to manage change.
- Merge overhead is minimized by frequent "early and often" integration of code through the hierarchy.
- Promotional structures offer a dynamic workflow, which can be modified and changed at any time while still preserving core development model needs.

- Each programmer can work in isolation from other programmers, to eliminate instances of "your changes broke my build." So each programmer needs a separate configuration of the code base.
- A QA engineer needs a configuration that combines the changes made by some or all of the programmers, in order to test all the new features in the next product release.
- To fix a bug in the previous product release, a maintenance engineer needs to start with the configuration that was used to build that release.
- While promotional models are a more natural way for teams to work inside an SCM system, implementing the model could require some changes to the SCM system, either with scripting or other tools to optimize the model.
Integration with issue tracking systems to provide change sets, or “change packages.” Change packages are the union of file and directory history to a particular issue. This will allow for easy movement of code through the stream hierarchy.

Automated merging and integration with parent and child branches. This must be in place so that when a hierarchy is established, most changes start at the top level but changes will flow down to the lower levels with parallel development happening at the same time.

Visualization to manage where in the process a team is and how the hierarchy is set up.

Promotional models are a great way for teams to manage a complex development process. It enables development organizations to manage effectively every configuration of the SCM codebase while ensuring that a process is followed.

**Conclusion**

Traditional SCM branching patterns are designed around decades-old software methodologies. Modern development teams deliver code very frequently, and have a need to manage a software development process right in the SCM repository.

Micro Focus® can help you to implement AccuRev, which uses natural- and process-based promotional patterns based on your development needs. This powerful solution will enable your teams to deliver rapidly with higher quality and maintains software configurations for every aspect of the development cycle.

To find out how Borland can help your organization to effectively branch and merge patterns with AccuRev, visit: [www.borland.com/accurev](http://www.borland.com/accurev)