Feature Branches und Continuous Integration sind kein Widerspruch

Sebastian Damm, Orientation in Objects GmbH

Continuous Integration und Feature Branches sind zwei allgegenwärtige Themen in der heutigen Software-Entwicklung. Sie repräsentieren zwei Lager, deren Anhänger teilweise keine besonders hohe Meinung über den jeweils anderen haben. Für viele Puristen sind die beiden Themengebiete derart widersprüchlich, dass eine Kombination der beiden Ansätze nicht nur unpraktikabel, sondern sogar undenkbar ist. Dieser Artikel zeigt, warum und wie eine solche Kombination – unter Zuhilfenahme einiger Kompromisse – dennoch möglich ist.

Bevor wir uns den Differenzen der beiden Lager widmen, werden die beiden Ansätze kurz vorgestellt. Leser, denen die Grundprinzipien bereits ausreichend bekannt sind, können die beiden folgenden Kapitel überspringen.

Continuous Integration

Der Kerngedanke von Continuous Integration (CI) ist die mindestens tägliche Integration der Arbeit aller Projekt-Mitglieder. Mit jedem Commit in das Repository wird über einen CI-Server (beispielsweise Jenkins oder Bamboo) der CI-Prozess *(siehe Abbildung 1)* ausgelöst, der die Anwendung kompiliert, paketiert und in der Regel automatisiert testet.

Die stetige und kontinuierliche Integration der Arbeit aller Entwickler soll im Verbund mit dem automatisierten Bauen und Testen der Anwendung gewährleisten, das Projekt möglichst fehlerfrei zu halten

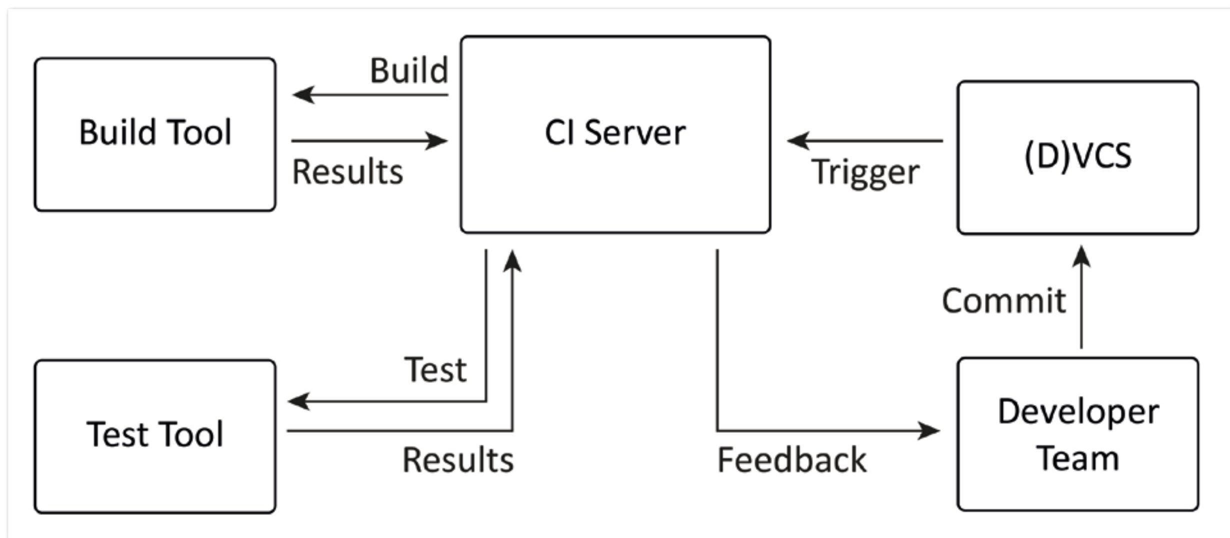

Abbildung 1: Schematische Darstellung des CI

beziehungsweise beim Auftreten von Fehlern diese zumindest unverzüglich zu erkennen. „Continuous Integration doesn't get rid of bugs, but it does make them dramatically easier to find and remove" *[1]*.

Der Begriff „Continuous Integration" wurde bereits im Jahr 2000 von Martin Fowler mit einem gleichnamigen Artikel *[1]* eingeführt. In diesem nannte er einige explizite Key Practices, die seiner Meinung nach erfüllt werden müssen, um effektiv CI betreiben zu können. Die beiden subjektiv wichtigsten Key Practices sind:

- *Key Practice Nr. 2: Automate the Build*
  Insbesondere in größeren Software-Projekten ist der Vorgang, das Projekt zu kompilieren und in ein ausführbares oder deploybares Format zu überführen, sehr komplex und somit fehlerträchtig, falls er manuell ausgeführt wird. Dementsprechend sollte dieser Prozess komplett automatisiert sein. Durch Build- und Dependency-Management-Tools wie Maven oder Gradle – die ihrerseits sehr einfach in CI-Server zu integrieren sind – ist dieser Punkt aus heutiger Sicht leicht realisierbar.
- *Key Practice Nr. 4: Everyone Commits To The Mainline Every Day*
  Jeder Entwickler sollte mindestens (!) einmal pro Tag seine Code-Änderungen in die Mainline (die Haupt-Entwicklungslinie innerhalb eines Repository, oft auch als „Master" bezeichnet) committen. Auf diese Weise sollen die Anzahl und die Komplexität von Konflikten zwischen mehreren Entwicklern möglichst gering gehalten werden.

Feature Branches

Feature Branches (FBs), beziehungsweise allgemein „Branching", sorgen für die Isolation von Programmcode, sodass mehrere Bestandteile eines Systems parallel bearbeitet werden können, ohne sich gegenseitig – in einem temporär instabilen Zustand – zu beeinflussen. Technisch werden dazu alle Objekte eines Zweigs in einen neuen Zweig kopiert, wodurch fortan Entwickler in beiden Zweigen unterschiedliche Versionen des gleichen Objekts oder der gleichen Datei bearbeiten können.

Der Leitgedanke beim Branching ist, dass Code-Änderungen erst dann, wenn sie sich in einem stabilen und getesteten Zustand befinden, in die Mainline integriert werden. *Abbildung 2* zeigt allerdings auch direkt das größte Problem des Branching: Sobald man einen neuen Branch erstellt, muss dieser irgendwann wieder integriert beziehungsweise gemergt werden, ein unter Umständen sehr fehleranfälliger Vorgang.

Doch nicht nur die CI-Fraktion kann in Form von Martin Fowler mit prominenten Fürsprechern punkten, auch Feature Branches haben mit Linus Torvalds namenhafte Prominenz auf ihrer Seite. Dieser argumentiert, dass der Linux-Kernel mithilfe von Branching entwickelt wird und dass sich viele Entwickler zu viele Sorgen über etwaige Merge-Konflikte machen: „The linux-next tree has something like 8.600 commits, and I've merged about 5.600 in the last few days. [...] Only 17 of the merges had any conflicts. And only a couple of those were annoying. The rest is all git doing all the work." *[2]*.

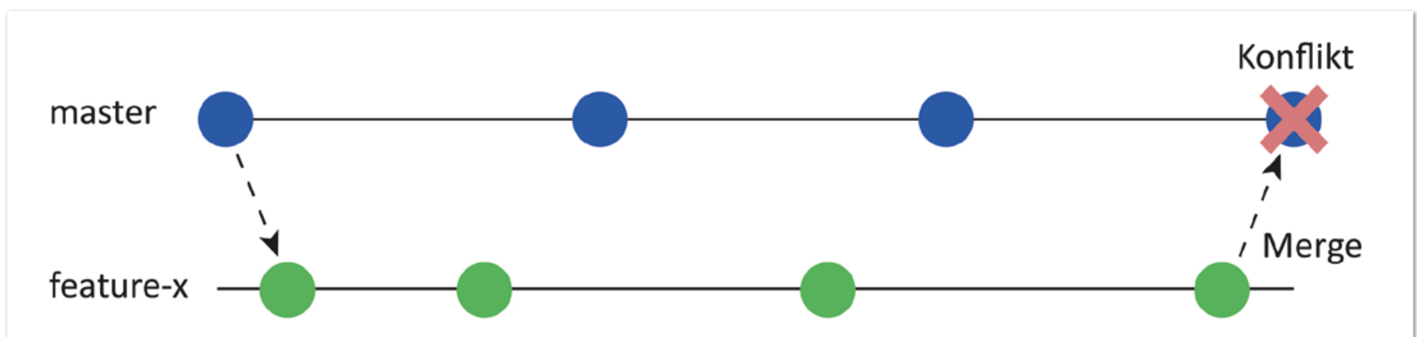

Abbildung 2: Darstellung von Branching und Merges/Merge-Konflikten

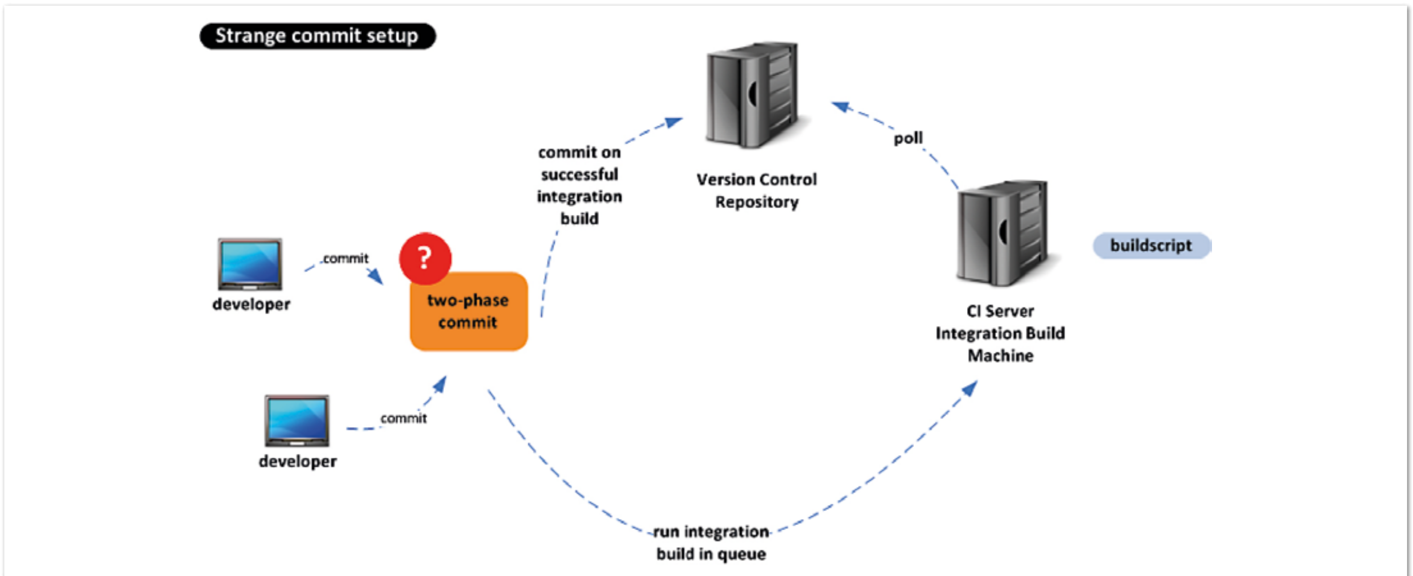

Abbildung 4: Two-Phase-Commit als „Future of CI" (Quellen: Beschreibung aus [8], grafische Darstellung aus [9])

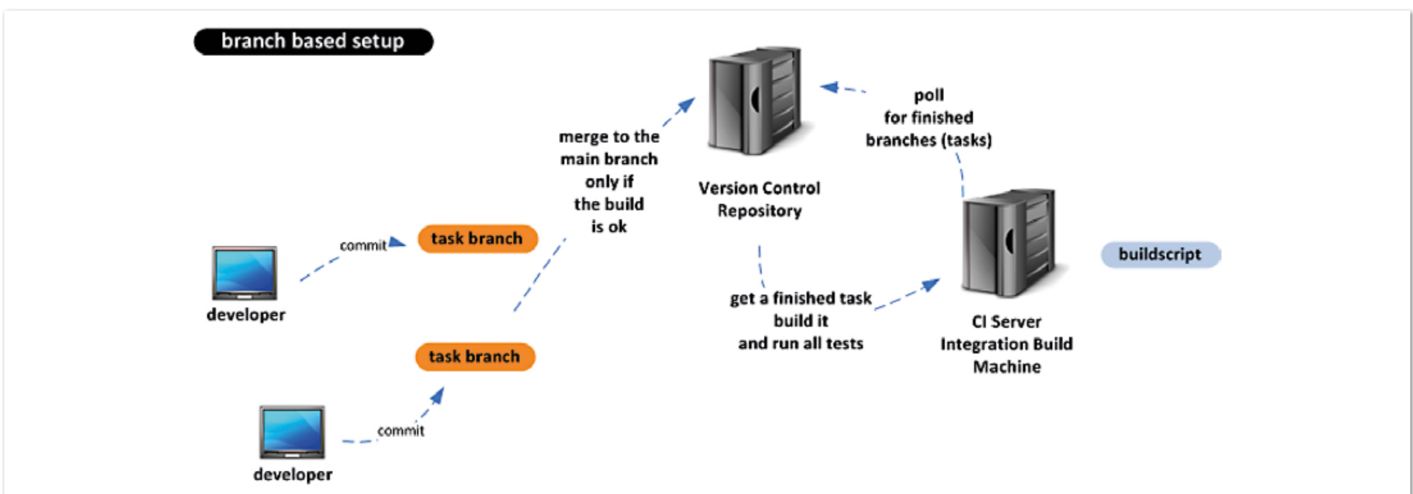

Abbildung 5: Branching Workflow als Antwort zum Two-Phase-Commit (Quelle: [9])

Während es im CI-Umfeld eine ganze Reihe von Artikeln mit Best Practices gibt, findet man im Bereich von Feature Branches in der Mehrzahl Artikel mit sogenannten „Anti-Pattern", also Praktiken, die man vermeiden sollte, falls man Branching erfolgreich einsetzen möchte. In [3] nennt der Autor eine ganze Reihe solcher Anti-Patterns, von denen wir auszugsweise zwei betrachten wollen:

- *Big Bang Merge*
  Merging has been deferred to the very end of the development effort and an attempt is made to merge all branches simultaneously.
- *Never Ending Merge*
  Merge activity never seems to end; there's always more to merge.

Let's get ready to rumble

Nachdem die beiden Kontrahenten vorgestellt sind, wollen wir uns nun der Frage widmen, in welchen Argumenten der Streit zwischen den beiden Parteien begründet liegt. Bereits in dem initialen CI-Artikel aus dem Jahr 2000 äußert sich Martin Fowler eher kritisch in Bezug auf FBs: „Keep your use of branches to a minimum".

Man könnte argumentieren, dass man diese Aussage auch im Kontext der Zeit sehen muss und Branching mit damaligen Versionsverwaltungs-Systemen wie CVS oder SVN weder sonderlich komfortabel noch performant war.

Allerdings bekräftigte Fowler im Jahr 2009 seinen Standpunkt zu FBs in einem dedizierten Artikel [4], in dem er sich der Frage widmet, wie CI und FBs zusammenpassen. Er nimmt insbesondere Stellung zu der Tatsache, dass viele Entwickler aus dem FB-Lager behaupteten, dass sie CI und FBs kombinieren, indem sie für jeden Branch mit jedem Commit per CI-Server das Projekt automatisch bauen und testen lassen: „So unless feature branches only last less than a day, running a feature branch is a different animal to CI. I've heard people say they are doing CI because they are running builds, perhaps using a CI server, on every branch with every commit. That's continuous building, and a Good Thing, but there's no integration, so it's not CI." Weitere Gegenargumente von Fowler sind Semantic Conflicts und die Beeinträchtigung von Opportunistic Refactoring durch Branching [4].

Fowler und das CI-Lager kritisieren allerdings nicht nur den (über-

triebenen) Einsatz von Branches, sie liefern auch Alternativen, um sich noch in der Entwicklung befindlichen Code vom Rest des Codes zu isolieren. Eine Alternative sind Feature Toggles.

Hier ist der betreffende Code zwar schon Bestandteil des Projekts, über einen globalen Schalter („Feature Toggle") ist die Ausführung des Codes allerdings zunächst deaktiviert. Sind die Code-Änderungen abgeschlossen und stabil, wird der Schalter umgelegt und der Code somit aktiv. Nachteilig ist hierbei, dass der Code so immer in die Produktion gelangt und dass der Toggle im Anschluss wieder entfernt werden muss. Manche Teams verwenden Feature Toggles allerdings auch wieder, anstatt die alten Schalter aus- und neue einzubauen. Dass dieses Vorgehen katastrophale Folgen haben kann, bewies die Knight Capital Group 2012, als sie innerhalb von 45 Minuten 460 Millionen Dollar verlor *[6]*.

Welche Argumente bringt die FBs-Fraktion gegen reines CI vor? Das Hauptproblem von CI ist, dass es eine reaktionäre Praxis ist. Auch wenn jeder Commit einen automatisierten Build- und Test-Prozess auslöst, so signalisiert ein Compilerfehler oder ein fehlgeschlagener Test letztlich nur, dass die komplette Mainline sich bereits in einem fehlerhaften Zustand befindet. Diese „Mainline Instability" wird auch in einem Artikel aus dem Jahr 2008 angeprangert *[7]*. „All the changes directly hit the mainline, so making it unstable is relatively easy."

Ein Fehler auf der Mainline beeinträchtigt dann mitunter das ganze Projekt-Team, da „[a] bug entering the mainline will be spread to all developers in a short time [and] several developers will end up fixing [it]". Diese Probleme sind der CI-Community allerdings auch bewusst. Paul Duvall et. al beschreiben im Epilog „The Future of CI" des Buches „Continuous Integration: Improving Software Quality and Reducing Risk" *[8]* zwei Lösungsansätze, mit denen man diesem Problem begegnen kann:

- *Two-Phase Commit*
  Before the repository accepts the code, it runs an integration build on a separate machine. Only if … successful will it commit the code …
- *Personal Builds*
  … capability for a developer to run an integration build using the integration build machine and his local changes along with any other changes committed to the version control repository

Wie dieser Two-Phase-Commit aussehen soll, ist allerdings nur schematisch und nicht sehr detailliert dargestellt. Vielleicht denken sich einige Leser an dieser Stelle, dass diese Anforderung des Two-Phase-Commit mit einem anderen Ansatz sehr einfach umsetzbar ist. Das dachte sich auch der Autor des Blog-Eintrags „Continuous integration future?" *[9]* und skizziert zunächst den Two-Phase-Commit, wie er in *[8]* von Duvall et. al beschrieben wurde *(siehe Abbildung 3)*. Er führt dann aus, dass das Problem, das das CI-Lager mit einem nicht näher spezifizierten Two-Phase-Commit lösen will, mit Branching und einem „Branch per Feature"-Workflow leicht zu erledigen ist *(siehe Abbildung 4)*.

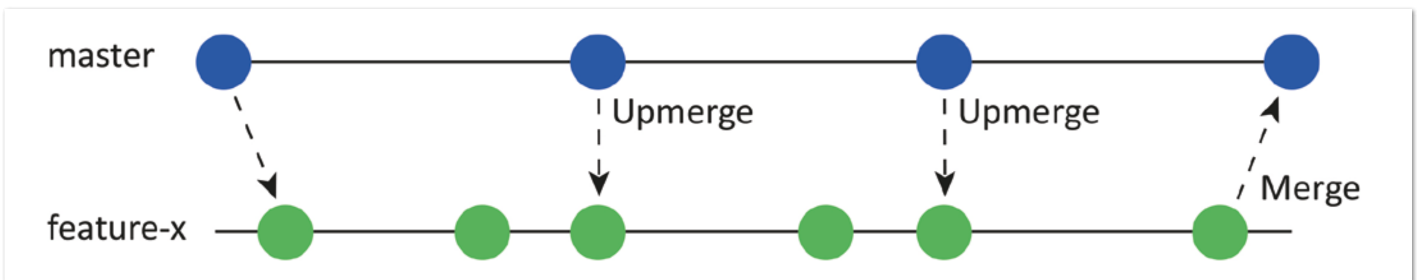

Abbildung 6: Vermeidung von Merge-Konflikten durch sofortige Integration von Änderungen auf der Mainline

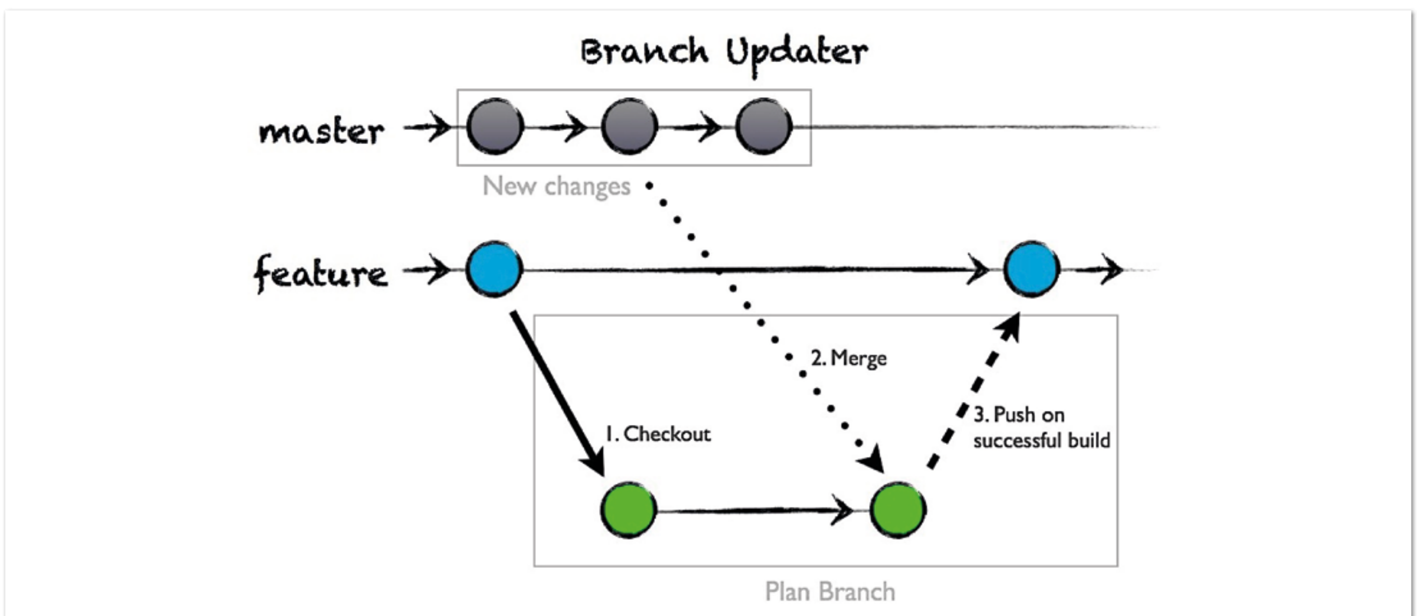

Abbildung 7: Automatischer Branch Updater von Bamboo (Quelle: [12])

Jetzt alles zusammen

Continuous Integration und (Feature) Branching sind zwei Ansätze, die sich in Teilen ihrer Definition nicht nur deutlich unterscheiden, sondern schlichtweg widersprechen: „By definition, if you have code sitting on a branch, it isn't integrated" [10]. So kann man zu dem Schluss kommen, dass CI und FBs sich gegenseitig ausschließen und man entweder nur einen der beiden Ansätze verfolgt oder gar keinen [10].

Man kann sich der Thematik allerdings auch von der anderen Seite nähern und nicht nur die Unterschiede der beiden Praktiken betrachten, sondern nach Gemeinsamkeiten suchen beziehungsweise nach Gründen, aus denen eine Kombination überhaupt sinnvoll sein könnte. Wenn man dies tut, stellt man fest, dass beide ein gemeinsames Hauptziel haben, nämlich den absoluten Schutz der Mainline beziehungsweise den Wunsch, zu jedem Zeitpunkt einen absolut fehlerfreien, vertrauenswürdigen Codezustand zu haben, den man so ohne größere Bedenken direkt auch in Produktion deployen würde. Da die beiden Ansätze also ein gemeinsames Hauptziel haben (das sie alleinstehend nicht perfekt erreichen) und das Hauptproblem von CI (die „broken builds", siehe [8] oder [9]) sich durch FBs beheben lässt, kann eine Kombination durchaus Sinn ergeben.

Wie kann eine pragmatische und funktionierende Kombination von CI und FBs in der Praxis aussehen? Jilles van Gurp nennt einige Fakten für diese Kombination und beschreibt Maßnahmen, die in jedem Fall ergriffen werden sollten [11]:

- *You can't do decentralized versioning unless you also decentralize your testing and integration.*
  Ganz entscheidend ist also, dass bereits auf Branch-Ebene (teilweise) integriert sowie mit den gleichen CI-Prozessen gebaut und getestet wird, wie dies für die Mainline der Fall ist.
- *... decentraliz[ing] testing and integration solve[s most] problems before change is pushed upstream. ... by decentralizing you have many people working on smaller sets of changes that are much easier to deal with: the collaborative effort on testing decreases and when it all comes together you have a much smaller set of problems to deal with.*
  Integriert man alle Änderungen der Mainline, beziehungsweise (indirekt) von anderen Branches, direkt in seinen Branch, so werden die potenziell fehlerträchtigen Merges auf Branch-Ebene durchgeführt und bei Fehlern die Mainline nicht beschädigt. Bei einer erfolgreichen Integration der Mainline in einen Branch ist die anschließende Integration zurück in die Mainline automatisch konfliktfrei.
- *Rebase against main frequently – If your feature branch is way behind, you have done something very wrong.*
  Abgeleitet aus dem vorherigen Punkt ist es zwingend notwendig, dass jegliche Änderung der Mainline von jedem Entwickler sofort in seinen Branch integriert wird (siehe Abbildung 5). Wird dies nicht berücksichtigt, sammeln sich die Integrations- und Merge-Aufwände an, die Gefahr eines Big-Bang-Merge steigt.
- *Push as early as you can – Feature branches are not about hiding change but about isolating change.*
  Was mitunter falsch verstanden wird, ist, dass es bei Branches nicht darum geht, Änderungen zu verstecken, sondern sie lediglich zu isolieren, bis sie in einem stabilen Zustand sind. Anschließend befinden sich die Code-Änderungen auf einem Branch allerdings in einem stabilen und getesteten Zwischenzustand. Dies kann nur gewährleistet werden, wenn der erste Punkt erfüllt ist und für jeden Branch der CI-Prozess zur Verfügung steht; dann sollten diese bereits zwischendurch immer wieder in die Mainline integriert werden.
- *Pull change rather than pushing it*
  Ein weiterer schöner Aspekt in einem Branch-basierten Workflow ist die Tatsache, dass man Änderungen nicht direkt mergen muss, sondern dass man auch mit Pull Requests arbeiten kann. Entwickler können so eine Anfrage initiieren, die dafür sorgt, dass ein anderer Entwickler oder eine Person mit einer spezifischen „Integrator"-Rolle wie der Projektleiter zunächst die vorgenommenen Änderungen überprüft. Erst nach dessen Bestätigung werden die Änderungen integriert. Dieses Vorgehen kann durch entsprechende Rechte-Verwaltung sogar forciert werden, indem Entwicklern ein direkter Merge in die Mainline schlichtweg verweigert wird.

Es wird deutlich, dass viele dieser Vorgaben stark miteinander verzahnt sind beziehungsweise aufeinander basieren. Der Punkt „Rebase against main frequently" beispielsweise kann nur funktionieren, wenn auch die darauffolgende Empfehlung „Push as early as you can" berücksichtigt wird.

Noch mal Schritt für Schritt

Diese Punkte sind für eine erfolgreiche Kombination von CI und FBs zu berücksichtigen:

- Feature Branch erzeugen
- Continuous Integration für Feature Branch bereitstellen
- Im Feature Branch arbeiten
- Master-Änderungen sofort (!) in Feature Branch übernehmen
- Bei stabilen Zwischenzuständen Merge oder Pull Request vom Branch in die Mainline (nach vorheriger erfolgreicher Integration der Mainline in den Branch)
- Nach Fertigstellung abschließender „Up Merge" in den Master

Was sich sehr aufwendig anhört, ist dank heutiger Tool-Unterstützung nur mit wenig manuellem Aufwand verbunden. Alle gängigen CI-Server unterstützen ein Feature namens „Branch detection". Das Anlegen eines neuen Branch wird erkannt, die definierten CI-Prozesse werden automatisiert für den Branch kopiert, wodurch der zweite Punkt gewährleistet ist.

Einige CI-Server (wie Atlassian Bamboo mit dem „Branch Updater") bieten die Möglichkeit, eine Abhängigkeit zwischen mehreren Branches zu definieren. Ist ein Branch A von einem Branch B abhängig, so wird bei jedem Commit auf A geprüft, ob in der Zwischenzeit (also seit dem letzten Build von A) Änderungen auf dem abhängigen Branch B vorgenommen worden sind. Ist dies der Fall, werden die Änderungen von B automatisch in A integriert und der Build- und Testprozess angestoßen. Falls dieser erfolgreich ist, werden die integrierten Änderungen direkt auf A gepusht (siehe Abbildung 6).

Selbst für die Erstellung und Verwaltung von Branches oder Pull Requests bieten heutige Issue-Tracker und Repository-Management-Tools ausgezeichnete Unterstützung. Der größte Aufwand in einem nach diesen Vorgaben aufgesetzten Projekt liegt damit in der notwendigen Entwickler-Disziplin, stabile Zwischenzustände ihrer Branches zwischendurch immer wieder in die Mainline zu integrieren. Nur so kann der beschriebene und dank Tool-Unterstützung

weitestgehend automatisierte Workflow funktionieren.

Fazit

Auch wenn nicht geleugnet werden kann, dass die beiden Ansätze teilweise zueinander widersprüchliche Definitionen besitzen, ergibt eine Kombination aufgrund eines gemeinsamen Hauptziels dennoch Sinn. Die Maßnahmen für eine erfolgreiche Kombination sind zahlreich, zudem ist eine große Portion Disziplin im gesamten Entwicklerteam notwendig.

Gelingt es allerdings, diese Vorgaben (mithilfe von Tool-Unterstützung) erfolgreich umzusetzen, so wird das gemeinsame Ziel, also der absolute Schutz der Mainline, besser erreicht, als es mit einem der beiden Ansätze alleinstehend möglich ist.

Quellen und weitere Informationen

[1] *https://martinfowler.com/articles/continuousIntegration.html*
[2] *https://plus.google.com/+LinusTorvalds/posts/fDENcrCqHmD*
[3] *https://blog.codinghorror.com/software-branching-and-parallel-universes*
[4] *https://martinfowler.com/bliki/FeatureBranch.html*
[5] *http://martinfowler.com/bliki/FeatureToggle.html*
[6] *https://dougseven.com/2014/04/17/knightmare-a-devops-cautionary-tale*
[7] *http://www.drdobbs.com/architecture-and-design/scm-continuous-vs-controlled-integration/205917960*
[8] Paul M. Duvall, Steve Matyas, Andrew Glover: Continuous Integration: Improving Software Quality and Reducing Risk, Addison-Wesley Signature Series, Pearson Education, 2007
[9] *http://blog.plasticscm.com/2008/03/continuous-integration-future.html*
[10] *https://arialdomartini.wordpress.com/2011/11/02/help-me-because-i-think-martin-fowler-has-a-merge-paranoia*
[11] *http://www.jillesvangurp.com/2011/07/16/using-git-and-feature-branches-effectively*
[12] *https://confluence.atlassian.com/bamboo/using-plan-branches-289276872.html*

Sebastian Damm
sebastian.damm@oio.de

Sebastian Damm arbeitet bei der OIO – Orientation in Objects GmbH in Mannheim als Entwickler, Trainer und Berater. Daneben ist er an der DHBW Mannheim als Informatik-Dozent tätig. Seine Schwerpunkte liegen in den Bereichen Web-Anwendungen mit Java EE beziehungsweise GWT und Spring sowie in der Automatisierung funktionaler Systemtests.

Sie wollen uns persönlich kennenlernen?
Dann treffen Sie uns auf der JavaLand in Brühl!

Bist Du auch IT-Spezialist?

TimoCom ist ein mittelständischer, inhabergeführter IT-Spezialist und Anbieter von Europas größter Online-Plattform für die Transport- und Logistikbranche. In unserer Inhouse-Entwicklung bist Du Teil eines agilen Teams und arbeitest in einem modernen Entwicklungsumfeld (Java, Oracle, CI, DI, JSF). Klingt interessant?

Daria und David - User Experience & Quality
Marc und Jann - IT Development

TimoCom® Jetzt bewerben und weiterentwickeln: www.timocom.de/jobs