Testest Du noch oder entwickelst Du schon wieder?

Optimiertes Testen: von Gateways, Gatekeepern und Keymastern

Björn Feustel, Steffen Schluff

Hohe Testabdeckung und Continuous Integration (CI) sind Teil agiler Softwareentwicklung. Sobald die Testlaufzeiten jedoch eine gewisse Dauer überschreiten, sieht man sich mit neuen Problemen konfrontiert. Der Entwickler will aus Zeitgründen nicht alle Tests vor dem Einchecken ausführen, weiß aber nicht, welche Tests nun relevant sind. Der CI-Server testet Änderungen nicht isoliert, sondern gebündelt, was die etwaige Fehlersuche erschwert. Dieser Artikel stellt neue Lösungsansätze, wie Selective Testing und Gateway Builds, sowie entsprechende Werkzeuge aus dem Java-Umfeld vor.

Been there, done that

Der Einsatz eines Continuous Integration (CI)-Servers gehört in der Softwareentwicklung heutzutage fast schon zum guten Ton. Die zugrunde liegende Idee ist, dass ein Entwicklerteam seine Quellen gemeinschaftlich in einem sogenannten Version Control System (VCS) verwaltet, auf das auch der CI-Server Zugriff hat. In regelmäßigen Abständen prüft dieser das VCS auf Änderungen, um beim Erkennen von solchen einen automatischen Build- und Testlauf zu starten und die Entwickler über die Folgen ihrer Änderungen zu informieren. Abbildung 1 zeigt diesen Sachverhalt schematisch.

Bei größeren Projekten wird man sich häufig in der Situation befinden, dass ein einzelner Build-Lauf des CI-Servers die Änderungen mehrerer Entwickler beinhaltet. Kommt es nun zu einem Fehler, stellt sich die Frage, welche konkrete Änderung diesen verursacht hat. Zudem besteht die Gefahr, dass sich der Fehler bereits bei anderen unbeteiligten Kollegen in der Entwicklung bemerkbar macht, da ja erst der Commit in das VCS und somit eine mögliche Verteilung und daran anschließend der CI-Server-Lauf erfolgt. Wurden darüber hinaus noch Änderungen eingecheckt, als die Codebasis bereits fehlerhaft war, verschlimmert sich die Lage noch weiter.

Dem geschilderten Problem kann man bis zu einem gewissen Maß dadurch begegnen, dass der CI-Server bei jedem von ihm erkannten Commit einen Build sowie einen Test anstößt und somit immer klar ist, welche Änderung zum Fehler geführt hat. Dies scheitert in der Praxis aber häufig an langen Build-Laufzeiten und mangelnder CI-Server-Hardware-Ausstattung, mittlerweile bekannt als das „Slow Machine"-Antimuster [Duv07].

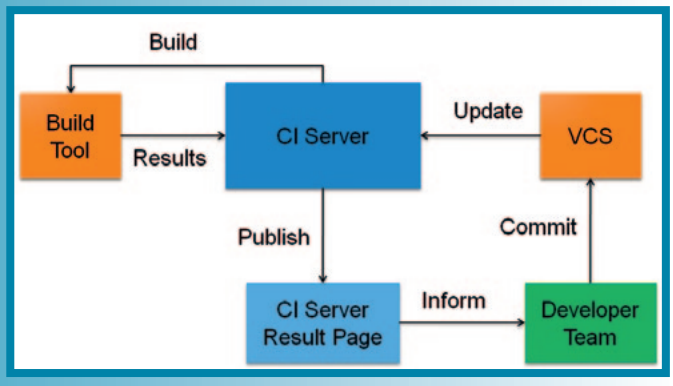

Abb. 1: CI-Server

Gateway-Builds

Einen konsequenteren Ansatz bieten „persönliche" Gateway-Builds. Hierbei schickt der Entwickler seine neuesten, aber noch nicht eingecheckten (bzw. commiteten) Änderungen zum Testen an einen Gateway-Build-Server (Schritt 1 in Abb. 2). Dieser testet nun diese Änderungen im Zusammenspiel mit dem aktuellen Codestand im VCS (Schritt 2) und sendet das Testergebnis zurück an den Entwickler (Schritt 3). Waren die Tests erfolgreich, kann dieser jetzt seine Änderungen gefahrlos in das VCS überführen und dem Team verfügbar machen (Schritt 4). Abbildung 2 verdeutlicht den schrittweisen Ablauf.

Die Ähnlichkeit der Gateway-Build-Idee zur „normalen" CI-Server-Verwendung ist offensichtlich und besteht letztlich nur aus einer Änderung der Reihenfolge im Ablauf, nämlich den CI-Server-Lauf vor dem Commit in das VCS auszuführen. Dementsprechend planen viele CI-Server-Hersteller dieses Feature unter dem Namen „pre-tested commit" für die nahe Zukunft ein beziehungsweise bieten dieses sogar schon an [PTCHudsonWiki,PTCJetBrains].

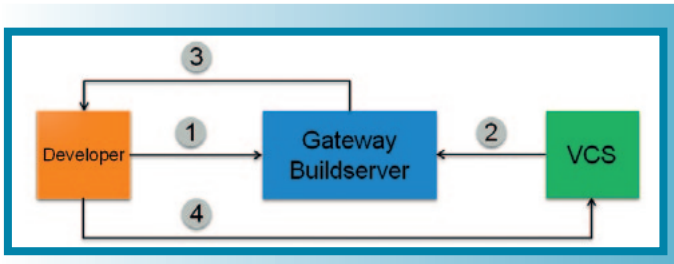

Abb. 2: Gateway-Build

Obwohl Gateway-Builds eine saubere Hauptentwicklungslinie im VCS gewährleisten, gilt es zu bedenken, dass die Entwickler zur Durchführung ihrer jeweiligen Gateway-Builds einen großen Ansturm auf den CI-Server auslösen und somit die Feedback-Zeiten für den Einzelnen erhöht werden. Hinzu kommt, dass insbesondere bei größeren Entwicklungsaufgaben meist keine echte Isolation vorliegt, sodass man früher oder später auf die Aussage *„The keys which open real [...] parallel development are branches."* [San08] treffen wird.

Branches: verzweigte Entwicklung

Ein Branch ist ein sich abspaltender neuer Entwicklungszweig. Insbesondere verteilte Versionsverwaltungssysteme (DVCS) erleichtern den Umgang mit Branches deutlich und verwenden sie dementsprechend auch als Grundlage für zahlreiche Workflows.

Erzeugte Branches müssen irgendwann auch wieder zusammengeführt, verschmolzen, werden. Dies geschieht mittels eines sogenannten Merge. Die einzelnen Branches können sich dabei je nach Größe und Alter textuell und semantisch stark unterscheiden und wahrscheinlich wird jeder der Aussage von Martin Fowler [Fow09] *„It's [...] particularly semantic conflicts that make big merges scary"* zustimmen können.

Are you the Keymaster?

Auch beim Einsatz von Branches darf die Hauptentwicklungslinie (Head) des VCS, in der irgendwann sämtliche Entwicklung zusammenläuft, nicht leiden. Aus diesem Grund werden bei stark verzweigter Entwicklung häufig sogenannte Gatekeeper eingesetzt, insbesondere im Bereich der DVCS-Workflows [BazaarVCS] lassen sie sich häufig antreffen. Die Bezeichnung kann je nach konkretem Werkzeug variieren, zum Beispiel findet gelegentlich auch der Begriff Integrator anstelle von Gatekeeper Verwendung.

Grundsätzlich gilt, dass der Gatekeeper ein Mensch oder eine Maschine sein kann, dessen Aufgabe es ist, die Stabilität der Hauptentwicklungslinie zu gewährleisten. Da er als einziger Schreibrechte in die Hauptentwicklungslinie besitzt, muss jeder von Entwicklerseite gewünschte Merge zurück in die Hauptentwicklungslinie von ihm kontrolliert und ausgeführt werden. Der Gatekeeper fungiert also als Kontrollinstanz der Hauptentwicklungslinie ähnlich wie der bereits zuvor beschriebene Gateway-Build, wie Abbildung 3 verdeutlicht.

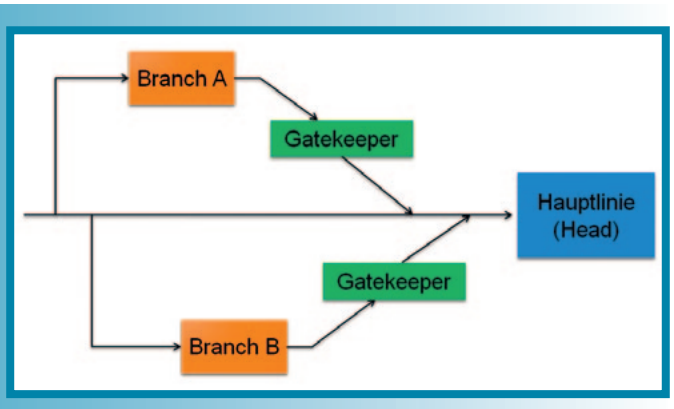

Abb. 3: Branches mit Gatekeeper

You're gonna need a bigger boat

Die meisten DVCS-Werkzeuge unterstützen oder beinhalten bereits Gatekeeper-Werkzeuge. Sollte man sich aber für Branch basiertes Arbeiten in Kombination mit einem Gatekeeper entscheiden, sieht man sich plötzlich mit einigen interessanten Aussagen und Ratschlägen konfrontiert:

- Jeder Branch bedeutet einen weiteren Build-Job auf dem CI-Server: *„each commit [...] comes with an automated build and test on the branch it's on"* [Fow09].
- Da prinzipiell alle Tests in allen Branches verwendet werden, ist mit erhöhter Last auf dem CI-Server zu rechnen: *„running all the tests again and again will be CPU demanding"* [San08].
- Vermutlich sind aber nicht alle Tests in jedem Branch von Interesse: *„subteams might be interested in a specific set of tests which is too slow to be run for every commit"* [PTCHudsonWiki].
- Der Merge-Durchsatz, den der Gatekeeper leisten kann, hängt von der Testlaufzeit ab: *„the gatekeeper does a merge, a compile and runs the test suite"* [BazaarVCS].
- Dementsprechend soll der Gatekeeper nur die wichtigsten Tests ausführen: *„when [an] integration is performed, the integrator runs a subset (smoke tests) of the complete test suite for each integrated branch"* [San08].
- Auf die Frage, welche der Tests denn nun tatsächlich die wichtigsten sind, findet man zu guter Letzt die folgende Antwort: *„you can use various tools and heuristics to decide which tests it might be prudent to run"* [PTCHudsonWiki].

Zusammenfassend kann man also sagen, dass bei allen hier vorgestellten Ideen, vom Gateway-Build bis hin zum Branch basierten Arbeiten, die Ausführungszeit eines einzelnen CI-Server-Testlaufs ein entscheidender Faktor für die Effizienz von Build-Systemen und damit der Produktivität jedes einzelnen Entwicklers ist.

Möglichkeiten der Testoptimierung

Doch was tun, wenn die Testlaufzeiten immer größer werden? Als erstes bieten sich sicherlich eher konventionelle Maßnahmen der Testoptimierung an:

- Reduzierung der Testanzahl: Auch wenn der einzelne Test im Idealfall sehr schnell zur Ausführung kommt, die Laufzeit von Hunderten oder Tausenden wird rasch zu einem Problem. Hier gilt es, zu hinterfragen, wie gut die einzelnen Tests sind. Sind sie aktuell, frei von unnötigen Überschneidungen und Dopplungen, prüfen sie ausschließlich wichtige Funktionen, Abläufe und Anforderungen?
- Strukturierung der Tests: Schnelle Unit-Tests sind im Hinblick auf ihre Laufzeit sicherlich nicht mit normalerweise deutlich langsameren Akzeptanz- oder Performance-Tests vergleichbar. Doch die Art der Tests bestimmt auch deren Ausführungszeitpunkt. Akzeptanztests müssen oder können nicht im gleichen Maße durch den Entwickler während der kleinschrittigen Entwicklungszyklen ausgeführt werden wie Unit-Tests. Sie sind vielmehr Teil der zentralen, automatisierten Integration per CI-Server oder nachgelagerter Teststufen. Eine Strukturierung in unterschiedliche Testsätze schafft hier die Möglichkeit, gezielt die jeweils notwendigen Tests auszuführen.

Je nach Projektgröße und -art reichen diese Maßnahmen aber nicht aus oder lassen sich nicht hinreichend gut umsetzen. Hier setzen zwei andere Möglichkeiten der Testoptimierung an: das Bilden von relevanten Test-Untermengen (Test Selection) und das Priorisieren von Tests (Test Priorization).

Wer die Wahl hat: Test Selection

Test Selection versucht das Problem einer zu großen Anzahl von Tests dadurch zu lösen, dass nur eine Untermenge aller Tests ausgeführt wird. Die zugrunde liegende Annahme ist,

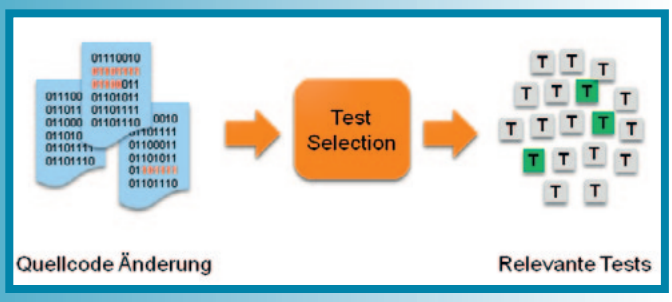

Abb. 4: Test-Untermengen

dass sowieso immer nur ein Teil der Tests als Regressionstests die Nebenwirkungen einer einzelnen Änderung an den Quell-artefakten testen kann und somit auch nur dieser Teil zur Ausführung kommen muss. Dadurch wird es im Idealfall möglich, die Testlaufzeit und den Bedarf an Ressourcen für die Tests immens zu reduzieren, wie in Abbildung 4 illustriert.

Doch wie lässt sich die dazu notwendige Verknüpfung der Testfälle zu den Quellartefakten herstellen? Hier gibt es mehrere Möglichkeiten:

- Testabdeckung: Bei der Messung der Testabdeckung mit Tools wie „Cobertura“ [Cobertura] oder „Clover“ [Clover] wird bereits die Relation der Tests zum Quellcode erfasst. Das heißt, es liegt also die Information darüber vor, welcher Test exakt welche Zeilen im Quellcode adressiert. Hieraus lassen sich direkt die Tests ableiten, die bei einer konkreten Änderung am Quellcode mindestens ausgeführt werden müssen. Was ist jedoch mit Artefakten, die keine direkte Testabdeckung besitzen, beispielsweise Konfigurationsdateien? Hier bieten die nachfolgenden Möglichkeiten Lösungsansätze.
- Failure History: Über die Information, welche Tests bei welchen Änderungen in der Vergangenheit fehlschlugen, lässt sich eine Verknüpfung zwischen Tests und Quellartefakten herstellen, die selbst keine Testabdeckung besitzen.
- Bugfix History: In vielen Projekten kann man davon ausgehen, dass für jeden in der Vergangenheit behobenen Fehler auch ein Regressionstest angelegt wurde und dass bekannt ist, welche Quellartefakten im Rahmen des Bugfixes geändert wurden. Somit lässt sich aus der Historie der Bugfixes ableiten, welche Tests bei Änderungen an den entsprechenden Quellartefakten ausgeführt werden sollten.

Weitere Möglichkeiten sind denkbar, zumeist jedoch wesentlich spezifischer in ihrer Machbarkeit und ihrem Nutzen. Beispielsweise ließen sich in Projekten mit umfassendem Requirements Tracing die über die Requirements bestehenden Assoziationen zwischen Quellartefakten und Akzeptanztests für die Testauswahl nutzen.

Doch egal welche Möglichkeiten für die Testauswahl genutzt werden, das automatisierte Erzeugen einer Untermenge aller Tests wird nie eine hundertprozentige Treffgenauigkeit bieten. Spätestens an zentralen Integrationspunkten oder nach einer bestimmten Anzahl durch „Test Selection“ festgelegten Testläufen müssen auch immer wieder alle Tests zur Ausführung gelangen.

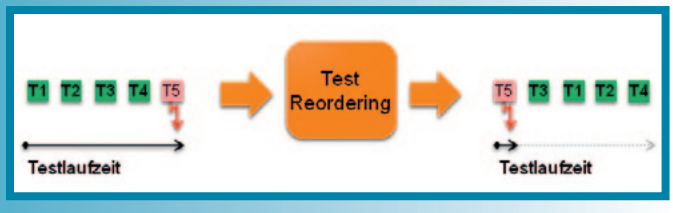

Abb. 5: Priorisieren von Tests

First things first: Test Reordering

Test Reordering, auch Test Priorization genannt, versucht die Ausführungsreihenfolge von Tests zu optimieren. Ziel ist, dass ein aussagekräftiges Feedback zu möglichen Nebenwirkungen einer Änderung so schnell wie möglich erzielt wird. Im bestmöglichen Fall wird dabei derjenige Test zuerst ausgeführt, der fehlschlägt („fail fast“), wodurch die Ausführung aller weiteren Testfälle hinfällig und somit entsprechend Zeit gespart wird, was Abbildung 5 veranschaulicht.

Wie schon bei der Test Selection zuvor stellt sich auch beim Test Reordering die Frage, nach welchen Kriterien die Tests vor der Ausführung sortiert werden. Zudem muss bei dieser Vorgehensweise sicher gestellt sein, dass die auszuführenden Tests keine Abhängigkeit bezüglich ihrer Ausführungsreihenfolge haben. Folgende Möglichkeiten kommen in Betracht:

- Most recent failures first: Diejenigen Tests, die in den letzten Testläufen fehlgeschlagen sind, werden möglichst früh erneut ausgeführt. Dieses Vorgehen findet häufig auch im „normalen“ Entwickleralltag statt. Bestimmte Tests schlagen fehl, der Entwickler nimmt Quellcodeänderungen vor und führt mittels Entwicklungsumgebung speziell die Tests aus, die im letzten Lauf erfolglos waren.
- Most frequent failures first: Die Tests, die in der Vergangenheit am häufigsten fehlgeschlagen sind und die somit besonders gute Dienste leisten, werden möglichst zu Beginn ausgeführt.
- Quick tests first: Tests, die eine relativ kurze Laufzeit besitzen, werden eher an den Anfang gestellt als Langläufer – in der Hoffnung, etwaige Fehlschläge möglichst früh zu entdecken.
- Random: Tests werden in zufälliger Reihenfolge ausgeführt, was dabei hilft, die bereits zuvor erwähnten unerwünschten Abhängigkeiten zwischen Tests zu entdecken.

Die hier genannten Kriterien müssen sich dabei nicht wechselseitig ausschließen, sondern können vielmehr gewichtet miteinander kombiniert werden.

Alle Theorie ist grau

Sowohl Test Selection als auch Test Reordering sind in der aktuellen Java-Build- und Testtool-Landschaft kaum zu finden. Dies ist verwunderlich, da beide bereits seit Längerem ausführlich beschrieben sind [DoHy05].

Wie bereits zuvor erwähnt, bieten sich Tools zur Messung von Testabdeckung von Hause aus zur Testoptimierung an, da in ihnen eine Verbindung zwischen Quell- und Testcode bestimmt und ausgewertet wird. So findet sich zum Beispiel im Tool „Clover“ der Firma „Atlassian“ ein sogenanntes Test-Optimierungs-Feature, welches die beiden zuvor geschilderten Ansätze von Test Selection und Reordering verknüpft [Clover,CloverTO,Hum08].

Das Testframework „JUnit“ wiederum bietet ab Version 4.6 einen neuen experimentellen Kern namens `MaxCore`, der in der Lage ist, basierend auf bestimmten Regeln Test Reordering durchzuführen [JUnit46,MaxCore]. Gleiches gilt für den CI-Server „Team City“, der über ein Feature namens „Risk Test Reordering“ verfügt [TeamCity].

Es bleibt somit abzuwarten, ob und wann sich diese vielversprechenden Ansätze in einer noch breiteren Tool-Unterstützung niederschlagen.

Fazit

Der Artikel hat gezeigt, wie sich mithilfe von Gateway-Build, Branches und Gatekeepern eine Stabilisierung in der Entwicklung im Hinblick auf die jeweilige Hauptentwicklungslinie im VCS erreichen lässt. Diese Vorgehensweisen haben jedoch zur Folge, dass die Laufzeiten von Tests dadurch noch relevanter als bisher werden. Test Selection und Test Reordering als Verfahren zur Testoptimierung machen hierbei einen sehr viel versprechenden Eindruck, allerdings ist die Tool-Unterstützung im Java-Umfeld erst noch im Entstehen.

Literatur und Links

[BazaarVCS] Workflows – Bazaar Version Control, `http://bazaar-vcs.org/Workflows`

[Clover] Code Coverage for Java – Clover, `http://www.atlassian.com/software/clover/`

[CloverTO] About Test Optimization, `http://confluence.atlassian.com/display/CLOVER/About+Test+Optimization`

[Cobertura] Java-Werkzeug Cobertura zur Messung der Codeabdeckung, `http://cobertura.sourceforge.net/`

[DoHy05] Hyunsook Do u. a., Prioritizing JUnit Test Cases: An Empirical Assessment and Cost-Benefits Analysis, 2.8.2005, in: Empirical Software Engineering, 2006, Band 11, Heft 1, `http://www.cse.unl.edu/~grother/papers/esej06.pdf`

[Duv07] P. Duvall, Automation for the people: Continuous Integration anti-patterns, 4.12.2007, `http://www.ibm.com/developerworks/java/library/j-ap11297/`

[Fow09] M. Fowlers WikiBlog: FeatureBranch, 3.9.2009, `http://martinfowler.com/bliki/FeatureBranch.html`

[Hum08] B. Humphrey, Stop testing so much!, 5.9.2008, `http://blogs.atlassian.com/developer/2008/11/stop_testing_so_much.html`

[JUnit46] JUnit 4.6 final release notes, `http://sourceforge.net/project/shownotes.php?release_id=675664&group_id=15278`

[PTCHudsonWiki] Designing pre-tested commit, `http://wiki.hudson-ci.org//x/2wNAAg`

[PTCJetBrains] JetBrains :: Development Academy | Concepts | Pre-tested Commit, `http://www.jetbrains.com/devnet/academy/concepts/pretested_commit.html`

[MaxCore] Klasse MaxCore (JUnit API), `http://kentbeck.github.com/junit/javadoc/latest/org/junit/experimental/max/MaxCore.html`

[San08] P. Santos, SCM: Continuous vs. Controlled Integration, Dr. Drobb's, 24.1.2008, `http://www.ddj.com/architect/205917960`

[TeamCity] Running Risk Group Tests First, TeamCity Documentation, `http://confluence.jetbrains.net/display/TCD6/Running+Risk+Group+Tests+First`

Björn Feustel ist als Trainer, Entwickler und Berater bei OIO tätig, einem Kompetenzzentrum für Java- und XML-Technologien. Seine Schwerpunkte liegen im Bereich Test- und Build-Management, Softwarearchitekturen sowie verteilte Systeme.
E-Mail: bjoern.feustel@oio.de

Steffen Schluff ist als Leiter der Softwarefactory bei OIO tätig. Seine Schwerpunkte liegen in den Bereichen Enterprise Java, XML und Open Source Tooling.
E-Mail: steffen.schluff@oio.de