



Simon Wiest

Continuous Integration mit Hudson

Grundlagen und Praxiswissen
für Einsteiger und Umsteiger

dpunkt.verlag

Continuous Integration mit Hudson



Dr. Simon Wiest ist Professor für Informatik im Fachbereich »Electronic Media« an der Hochschule der Medien, Stuttgart. In seiner selbstständigen Tätigkeit als freier Berater für Continuous Integration begleitete er Softwareteams zwischen 5 und 1.200 Entwicklern. Seit 2007 ist er Committer und Evangelist im Hudson-Projekt. Seine Beiträge wurden mit einem Sun Microsystems Community Innovation Award ausgezeichnet. Er spricht regelmäßig auf Fachkonferenzen, in Firmen und Java User Groups. Dr. Wiest ist zu erreichen unter www.simonwiest.de.

Simon Wiest

Continuous Integration mit Hudson

**Grundlagen und Praxiswissen
für Einsteiger und Umsteiger**

Simon Wiest
simon.wiest@simonwiest.de

Lektorat: René Schönenfeldt
Copy-Editing: Annette Schwarz, Ditzingen
Herstellung: Nadine Thiele
Umschlaggestaltung: Helmut Kraus, www.exclam.de
Druck und Bindung: Media-Print Informationstechnologie, Paderborn

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie;
detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

ISBN:
Buch 978-3-89864-690-1
PDF 978-3-89864-887-5
ePub 978-3-89864-852-3

1. Auflage 2011
Copyright © 2011 dpunkt.verlag GmbH
Ringstraße 19 B

69115 Heidelberg
Die vorliegende Publikation ist urheberrechtlich geschützt. Alle Rechte vorbehalten. Die Verwendung
der Texte und Abbildungen, auch auszugsweise, ist ohne die schriftliche Zustimmung des Verlags
urheberrechtswidrig und daher strafbar. Dies gilt insbesondere für die Vervielfältigung, Übersetzung
oder die Verwendung in elektronischen Systemen.
Es wird darauf hingewiesen, dass die im Buch verwendeten Soft- und Hardware-Bezeichnungen sowie
Markennamen und Produktbezeichnungen der jeweiligen Firmen im Allgemeinen warenzeichen-,
marken- oder patentrechtlichem Schutz unterliegen.
Alle Angaben und Programme in diesem Buch wurden mit größter Sorgfalt kontrolliert. Weder Autor
noch Verlag können jedoch für Schäden haftbar gemacht werden, die in Zusammenhang mit der
Verwendung dieses Buches stehen.

5 4 3 2 1 0

Geleitwort von Kohsuke Kawaguchi

One of the challenges in an open-source project is to come up with a comprehensive documentation. In the face of incoming bug reports and new feature requests, it always ended up postponed. So for the longest time, we relied on the community contributed Wiki and mailing list archives as a substitute for a documentation. The problem with those resources is that they are neither linear nor particularly well organized. Those are often specific questions and answers, troubleshooting guides, and other instructions, but it just doesn't work well with those who are new to continuous integration or Hudson to get a broad view on what it has to offer. I've done a few smaller articles in Japan toward this goal in the past, but it only made me more convinced that this is a serious undertaking.

So I was very happy when I learned that Simon was writing a book. In this book, he discusses the bread and butter functionalities of Hudson in details. Those of you who are new to Hudson and those of you who are using Hudson for a while are both likely to find this book useful. I should also point out that the part in which he discusses the organizational implications of Hudson is novel, and should come in very handy for those who are approaching CI in a more careful manner.

Simon is a longtime member of the Hudson community, and quite instrumental in the localization work. I don't remember exactly when he became a part of the community – German translation wasn't the first translation of Hudson, although it was pretty close. But the first real impression that he made on me was when he came up with a tool that tracks the progress of localizations in various languages, complete with a list of translations that became out of date, by looking at the version control history of the source files. I just went »Wow!«. So it was only natural that he won a Glassfish Community Award in 2008 in recognition of these valuable contributions. And finally now that this book is finished, I'm hoping he'd have time to wrap that up and make it generally available.

I'd be also amiss if I talked about Simon without mentioning his own Hudson extreme feedback devices, which is a triplet of plastic bears that glow and flash as your build status changes. This was done via Ethernet-controlled power sockets, and was one of the early feedback devices around Hudson. As a builder of one of those XFDs myself, I think there's something with bright LEDs that attracts geeks, but I digressed.

I hope you enjoy his book.

Kohsuke Kawaguchi
Creator of Hudson

Inhaltsverzeichnis

1	Einleitung	1
1.1	Kennen Sie die »Integrationshölle«?	1
1.2	Warum Continuous Integration (CI)?	3
1.3	Warum Hudson?	4
1.4	Warum dieses Buch?	6
1.5	Wer dieses Buch lesen sollte	7
1.6	Wie man dieses Buch lesen sollte	9
1.7	Konventionen	10
1.8	Website zum Buch	10
1.9	Danksagungen	10
1.10	Zusammenfassung	11
2	CI in 20 Minuten	13
2.1	Was ist CI?	13
2.2	Was ist CI nicht?	16
2.3	Software entwickeln ohne CI	18
2.4	Software entwickeln mit CI	19
2.5	Zusammenfassung	20
3	Welche Vorteile bringt CI?	23
3.1	Reduzierte Risiken	23
3.2	Verbesserte Produktqualität	24
3.3	Allzeit auslieferbare Produkte	25
3.4	Gesteigerte Effizienz	26
3.5	Dokumentierter Build-Prozess	27
3.6	Höhere Motivation	28
3.7	Verbesserter Informationsfluss	29

3.8	Unterstützte Prozessverbesserung	29
3.9	Zusammenfassung	29
4	Die CI-Praktiken	31
4.1	Gemeinsame Codebasis	31
4.2	Automatisierter Build	32
4.3	Häufige Integration	34
4.4	Selbsttestender Build	35
4.4.1	Compiler	35
4.4.2	Unit-Tests	35
4.4.3	Komponententests	36
4.4.4	Systemtests	36
4.4.5	Inspektionen	37
4.5	Builds (und Tests) nach jeder Änderung	38
4.6	Schnelle Build-Zyklen	38
4.7	Tests in gespiegelter Produktionsumgebung	39
4.8	Einfacher Zugriff auf Build-Ergebnisse	40
4.9	Automatisierte Berichte	41
4.10	Automatisierte Ausbringung (Deployment)	42
4.11	Zusammenfassung	43
5	Hudson im Überblick	45
5.1	Die Hudson-Story	45
5.2	Architektur und Konzepte	46
5.2.1	Systemlandschaft	47
5.2.2	Datenmodell	50
5.2.3	Benutzerschnittstellen	55
5.3	Die Top-10-Highlights	60
5.3.1	Schnelle Installation	60
5.3.2	Effiziente Konfiguration	60
5.3.3	Unterstützung zahlreicher Build-Werkzeuge	61
5.3.4	Anbindung von Versionsmanagementsystemen ..	61
5.3.5	Testberichte	62
5.3.6	Benachrichtigungen	63
5.3.7	Remoting-Schnittstelle	64
5.3.8	Abhängigkeiten zwischen Jobs	64
5.3.9	Multikonfigurationsbuilds (»Matrix-Builds«) ..	65
5.3.10	Verteilte Builds	66
5.3.11	Plugins	67

5.4	Hudson im Vergleich zu Mitbewerbern	67
5.4.1	Proprietäre Eigenentwicklungen	68
5.4.2	CruiseControl	69
5.4.3	ThoughtWorks Cruise	72
5.4.4	Atlassian Bamboo	76
5.4.5	JetBrains TeamCity	81
5.5	Zusammenfassung	86
6	Installieren und Einrichten	87
6.1	Schnellstart in 60 Sekunden	87
6.2	Systemkonfiguration	89
6.3	Fortgeschrittene Installation	90
6.3.1	Systemvoraussetzungen	91
6.3.2	Ablaufumgebungen	92
6.3.3	Datei-Layout	93
6.3.4	Installationsmethoden	95
6.3.5	Starten und Stoppen	96
6.3.6	Aktualisieren	99
6.3.7	Backups	100
6.3.8	Deinstallation	103
6.3.9	Plugins	103
6.3.10	Tipps aus der Praxis	104
6.4	Zusammenfassung	106
7	Hudson im täglichen Einsatz	107
7.1	Ihr erster Hudson-Job im Schnelldurchlauf	108
7.1.1	Den ersten Job einrichten	108
7.1.2	Statusanzeigen auf der Übersichtsseite	112
7.2	Jobtypen	114
7.2.1	Gemeinsamkeiten	114
7.2.2	Free-Style-Projekt	116
7.2.3	Maven-2-Projekt	117
7.2.4	Multikonfigurationsprojekt (»Matrix-Builds«) ..	118
7.2.5	Überwachung eines externen Jobs	118
7.3	Versionsmanagement integrieren	120
7.3.1	Konfiguration eines Projekts	120
7.3.2	Repository-Browser anbinden	124
7.4	Auslösen von Builds	126
7.4.1	Manuell (build trigger)	127
7.4.2	Zeitgesteuert	127
7.4.3	Bei Änderungen im Versionsmanagementsystem	128
7.4.4	Abhängigkeiten zu anderen Projekten	129
7.4.5	Auslösung durch Plugins	131

7.5	Testwerkzeuge integrieren	132
7.5.1	JUnit, TestNG	132
7.5.2	Weitere Test-Frameworks der xUnit-Familie . . .	135
7.6	Benachrichtigungen verschicken	136
7.6.1	E-Mail	136
7.6.2	Email-ext (Plugin)	138
7.6.3	RSS-Feeds	140
7.6.4	Hudson Build Monitor (Firefox Add-on)	141
7.6.5	Twitter (Plugin)	141
7.6.6	hudson-eclipse (Eclipse-Plugin)	142
7.6.7	Informationsradiatoren	143
7.6.8	eXtreme-Feedback-Geräte anbinden	144
7.7	Dokumentationswerkzeuge integrieren	146
7.7.1	Javadoc	146
7.7.2	Maven Site	148
7.7.3	Einbindung beliebiger HTML-Berichte	148
7.8	Analysewerkzeuge integrieren	151
7.8.1	Checkstyle, PMD, FindBugs	151
7.8.2	Warnings (Plugin)	156
7.8.3	Task Scanner (Plugin)	156
7.8.4	Cobertura (Plugin)	158
7.8.5	Sonar (Plugin)	161
7.9	Issue-Tracker integrieren	163
7.9.1	Atlassian JIRA	164
7.9.2	Mantis	168
7.10	Zusammenfassung	169
8	Hudson für Fortgeschrittene	171
8.1	Parametrisierte Builds	171
8.1.1	Definition von Parametern	172
8.1.2	Typische Anwendungsfälle	173
8.2	Ansichten (views) und Dashboards	175
8.2.1	Benutzerdefinierte Listenansichten	175
8.2.2	Projekte filtern und gruppieren	176
8.2.3	Verschachtelte Ansichten	178
8.2.4	Hudson als Ihr Build-Portal	178
8.3	Parallelisierung der Build-Aktivitäten	179
8.3.1	Abhängigkeiten zwischen Projekten	181
8.3.2	Nachgelagerte Builds zusammenführen	185
8.3.3	Exklusiver Zugriff auf Ressourcen	186
8.3.4	Parameterübergabe an nachgelagerte Builds	187
8.3.5	Abbruch hängender Builds	188

8.4	Multikonfigurationsprojekte	189
8.4.1	Anlegen eines Multikonfigurationsprojekts	190
8.4.2	Verteilte Matrix-Builds	192
8.4.3	Zusammenfassung der Build-Ergebnisse	193
8.5	Verteilte Builds	194
8.5.1	Warum verteilen?	194
8.5.2	Warum nicht verteilen?	194
8.5.3	Slave-Knoten einrichten	195
8.5.4	Verteilen der Build-Umgebung	198
8.5.5	Überwachen des CI-Clusters	200
8.5.6	Virtualisierung	201
8.5.7	Bauen in der Wolke	206
8.6	Hudson absichern	212
8.6.1	Authentifizierung	213
8.6.2	Autorisierung	217
8.6.3	Auditierung	218
8.7	Weitere nützliche Plugins	220
8.7.1	Claim	220
8.7.2	Configuration Slicing	221
8.7.3	Build-Promotion	223
8.7.4	Description Setter	225
8.7.5	Green Balls	226
8.7.6	Locale	227
8.7.7	Continuous Integration Game	227
8.8	Zusammenfassung	228
9	Hudson erweitern	229
9.1	Erste Schritte	230
9.1.1	Schnellstart: Ihr erstes Plugin in 60 Sekunden ..	230
9.1.2	Anatomie eines Plugins	234
9.1.3	Einrichten einer Entwicklungsumgebung	240
9.1.4	Zusammenfassung	245
9.2	Hudsons Plugin-Konzept	245
9.2.1	Hudsons Technologie-Stapel	245
9.2.2	Was muss ein Plugin-Entwickler wissen?	248
9.2.3	Hudsons Erweiterungspunkte (Extension Points)	250

9.3	Beispiel: Das Plugin »Artifact Size«	263
9.3.1	Die Aufgabe	263
9.3.2	Erweiterungspunkt »ListViewColumn«	264
9.3.3	Erweiterungspunkte »Recorder« und »Action«	269
9.3.4	Erweiterungspunkt »CLICommand«	272
9.3.5	Bereitstellen der Artefaktgröße per XML-API . .	275
9.3.6	Online-Hilfetexte	276
9.3.7	Lokalisierung und Internationalisierung	277
9.4	Zusammenfassung	282
10	Aufwand einer CI-Einführung	283
10.1	Erstaufwand für Vollautomatisierung des Builds	283
10.2	Erstaufwand für Erstellung von fehlenden Tests	284
10.3	Optimieren des Build-Prozesses für CI	285
10.4	Hardwareressourcen für ein CI-System	285
10.5	Installation des CI-Systems	286
10.6	Administration des CI-Systems	286
10.7	Schulung	287
10.8	Zusammenfassung	288
11	Tipps zur Einführung von CI	289
11.1	Mit einem Pilotprojekt starten	289
11.2	Build-Zeiten kurz halten	290
11.3	Codeanalyse: Sanft starten, konstant steigern	290
11.4	Nur messen, was auch beachtet wird	291
11.5	Erscheinen neuer Plugins verfolgen	291
11.6	Den Spaß nicht vergessen	292
12	Fazit und Ausblick	293
12.1	Continuous Integration – wie geht's weiter?	293
12.2	Hudson – wie geht's weiter?	294
12.3	Zusammenfassung	296
	Literatur und Quellen	297
	Index	299

1 Einleitung

- Was ist Continuous Integration? Und was bringt sie Ihnen?
- Wer ist Hudson und was kann er für Sie tun?
- Wer sollte dieses Buch lesen? Und wie?

1.1 Kennen Sie die »Integrationshölle«?

»Oh ja!«, sehe ich den einen oder anderen Leser leidvoll nicken. Falls Ihnen der Begriff nicht geläufig sein sollte – die Geschichte beginnt meistens so:

Sie arbeiten in einem jener »führenden Softwarehäuser« an der Entwicklung des Flaggschiffproduktes mit. Am kommenden Tag soll Ihre Abteilung die längst überfällige neue Version Ihrer Software ausliefern – vom Vertrieb seit Monaten verlangt und vom Marketing noch länger beworben. Ihr Chef hat nachtleuchtende Silikonarmbändchen mit dem Aufdruck *»Failure is not an option«* verteilen lassen.

Die Entwickler werfen also eilig auf dem Abteilungsserver ihre Codeänderungen ab, die sie in den letzten Wochen auf ihren Rechnern erbrütet haben. Doch schon bald wird klar: Die Integration wird auch dieses Mal wieder direkt im Fegefeuer stattfinden!

Erst nach vielen Korrekturen kompiliert das Projekt. Die (spärlich) vorhandenen Tests schlagen fehl, manchmal aber auch nicht. Adrenalin trifft auf Testosteron. Es wird laut in der Abteilung: »Bei mir läuft's aber« – »Ich habe dort nichts geändert!« – »Ohne vernünftiges Refactoring sage ich gar nichts mehr!«

Am späten Abend laufen die Tests endlich durch. Kein Wunder, denn störende Programmteile wurden beherzt auskommentiert. Man muss ja auch etwas für das nächste Service Pack übrig lassen, oder nicht?

Kurz vor Mitternacht übernimmt »der Meister« das Kommando. Als langjähriger Mitarbeiter mit Kopfmonopol kennt er als Einziger die geheimen Schritte, die notwendig sind, um eine Distribution zu

*Softwareintegration
im Fegefeuer*

erstellen – und ja, nur er hat die notwendigen Werkzeuge dazu auf seinem Rechner installiert. Abblende nach Schwarz.

Am nächsten Tag wird die Software tatsächlich ausgeliefert. Das Team hat Bauchschmerzen, denn keiner weiß wirklich genau, welche Änderungen in dieses Release eingeflossen sind. Die alten Hasen haben schlauerweise schon mal vierzehn Tage Urlaub eingereicht. Die jüngeren Kollegen stellen sich unter den Türsturz und halten die Luft an. Hoffentlich nicht zu lange, denn das Marketing bewirbt bereits die nächste Version: »*the next big thing*«...

Neue Hoffnung

Während Sie also im Wartezimmer Ihres Betriebspsychiaters über die Ereignisse der letzten Tage nachsinnen, erleben Sie einen wilden Wunschtraum: Wäre es nicht fantastisch, wenn ...

- jemand *rund um die Uhr* alle Codeänderungen an Ihrem Produkt im Auge behielte?
- Jemand, der das komplette Produkt *selbstständig integrieren* könnte, bis hin zur Distribution? Am besten nach jeder Änderung? Ohne gelangweilt, genervt oder nachlässig zu werden?
- Jemand, der Ihr Produkt *gründlich testen* und bei neuen Problemen sofort *Alarm schlagen* würde?
- Jemand, der den gesamten *zeitlichen Verlauf* Ihres Projekts kennen würde, von der ersten Codezeile an? Der Ihnen mit gebrauchsfertigen *Visualisierungen* und *Berichten* helfen würde, Trends und wiederkehrende Fehlermuster aufzuspüren?

Mit anderen Worten: Wäre es nicht schön, einen persönlichen Butler zu haben, der sich kontinuierlich um die Integration Ihres Produktes kümmert?

Die gute Nachricht: Mit dem Continuous-Integration-System Hudson steht Ihnen genau ein solcher »Build-Butler« zur Verfügung. Er kann innerhalb von Minuten seine Dienstwohnung auf Ihrem Server beziehen, kommt mit den allerbesten Referenzen und arbeitet für Sie kostenlos. Interessiert?



Abb. 1-1
Hudson – Ihr Build-Butler

1.2 Warum Continuous Integration (CI)?

Continuous Integration ist ein hervorragendes Werkzeug, um in Softwareentwicklungsprojekten Risiken zu minimieren und Qualität zu steigern. Wie das?

Risiken minimieren,
Qualität steigern

Viele Entwicklungsprojekte geraten in Gefahr, weil die Integration, also die »Endmontage« aller Programmkomponenten, erst im allerletzten Moment angegangen wird. Leider ist dies der denkbar schlechteste Zeitpunkt für unliebsame Überraschungen. Die eingangs geschilderte »Integrationshölle« ist in vielen Softwareprojekten traurige Realität. CI propagiert häufigere Integrationen als den Ausweg aus dieser Misere. Nur: Wie soll ein Team gleich mehrmals am Tag integrieren, wenn schon die bisherige Frequenz »Einmal im Quartal« kaum zu bewältigen war?

Unverzichtbare Grundlage dafür ist die vollständige Automatisierung des Integrationsprozesses, der auch Tests, Dokumentation, Bereitstellung usw. eines Produktes umfassen kann. Viele Teams setzen dazu bereits Werkzeuge wie Ant, Maven oder Eigenentwicklungen ein. Im zweiten Schritt sorgt dann ein CI-Server für die regelmäßige Ausführung dieses Integrationsprozesses, etwa nach jeder Codeänderung oder einmal pro Nacht (*nightly build*).

Häufige Integration vermeidet nicht nur das Durchleiden der »Integrationshölle« (oft verschlimmert gefolgt von ihrem hässlichen Cousin »Big Bang Testing«). Sie bietet sogar noch eine ganze Reihe weiterer wichtiger Vorteile:

Vorteile von CI

- *Der Integrationsaufwand sinkt*, weil bei frühzeitiger Warnung kleinere Korrekturen ausreichen, um ein auseinanderlaufendes Projekt wieder auf gemeinsamen Kurs zu bringen. Umfangreiche Nacharbeiten unter hohem Zeitdruck vor dem Fertigstellungstermin entfallen.
- *Die Fehlersuche wird vereinfacht*, weil häufigere Integrationen geringere Veränderungen zur vorausgegangenen Integration bedeuten. Das ist wünschenswert, denn je weniger geändert wird, desto weniger Stellen kommen als Fehlerquelle in Frage und müssen untersucht werden.
- *Die Teammoral steigt*, weil häufigere Integrationen auch mehr Rückmeldungen an die Entwickler bedeuten. Erfolgreiche Änderungen und neue Funktionen werden zeitnah vom CI-System positiv bestätigt. Stellen Sie sich einfach einen Kollegen vor, der Ihnen nach jeder Änderung wenige Minuten später anerkennend auf die Schulter klopft. Zum anderen lässt sich – wie durch ein Fangnetz gesichert – mutiger entwickeln, beherzter refaktorisieren und gefahrlos Neues ausprobieren.
- *Manuelle Routineaufgaben* entfallen durch Automatisierung. Die Erzeugung lauffähiger Zwischenstände und interner Demoversisionen (»Kann man schon was sehen?«) ist somit kein lästiger Zeitfresser mehr, sondern läuft unaufgeregt nebenher. Pannen beim Integrieren und Bereitstellen eines Produktes werden vermieden, weil dieser Ablauf nicht mehr alle paar Monate, sondern mehrmals am Tag durchgeführt wird.

Was hält Sie also noch davon ab, diese Vorteile Ihrem Team zugänglich zu machen? Sie benötigen lediglich eine motivierte Person, welche die CI-Einführung in die Hand nimmt (das sind Sie), technisches und organisatorisches Know-how (dieses Buch) und ein gutes CI-System (nächster Absatz).

1.3 Warum Hudson?

Der Markt bietet inzwischen zahlreiche CI-Systeme an. Eine Vergleichsmatrix mit den dreißig bekanntesten Vertretern finden Sie unter [ThoughtWorks10]. Spricht man mit Entwicklern in Firmen und auf Konferenzen, lässt sich jedoch eine Konzentration auf wenige Anbieter beobachten. Vor allem Hudson erfreut sich hier großer Beliebtheit. Warum entscheiden sich Unternehmen wie eBay, Yahoo, Oracle, SAP oder die Allianz für Hudson?

Vorneweg: Es liegt *nicht* an der kostenlosen Verfügbarkeit. Selbstverständlich macht dieser Umstand den Einstieg leichter. Aber als alleinige Erklärung für Hudsons Erfolg greift dies zu kurz, denn:

- Es existieren zahlreiche kostenlose Alternativen. Zudem bieten inzwischen viele kommerzielle Hersteller eingeschränkte »Einstiegerversionen« ebenfalls kostenfrei an.
- Selbst kostenpflichtige CI-Systeme dürften sich bereits nach wenigen Monaten amortisiert haben, etwa durch Produktivitätsgewinne.

Viele Anwender stufen stattdessen Hudsons einzigartige Kombination folgender Vorteile als ungleich wichtiger ein:

Vorteile von Hudson

- *Schnelle Resultate:*
Hudson ist zügig installiert und eingerichtet. In Kapitel 5 werden Sie eine produktionstaugliche Instanz in 60 Sekunden aufsetzen. Sind Ihre Build-Prozesse schon mit Ant oder Maven automatisiert, können Sie in vielen Fällen bereits nach wenigen Minuten von CI mit Hudson profitieren.
- *Unkomplizierter Betrieb:*
Die meiste Zeit arbeitet Hudson, wie man es von einem erstklassigen Butler erwartet: diskret wachsam im Hintergrund, genügsam im Ressourcenverbrauch und geschliffen in den Umgangsformen. Die webbasierte Benutzeroberfläche lässt auch Gelegenheitsadministratoren die wichtigsten Handgriffe ohne Handbuch erledigen.
- *Plugin-Architektur:*
Trotz etablierter »best practices« und aller Standardisierung werden Sie in jedem Team leicht unterschiedliche Werkzeugketten finden. Dank seiner Plugin-Architektur ist Hudson ein wahres »Infrastruktur-Chamäleon«. Es unterstützt unterschiedlichste Versionsverwaltungen, Build-Werkzeuge, Compiler, Testframeworks, Nachrichtenkanäle usw. Über 200 Plugins stehen bereits zur Verfügung, und wöchentlich kommen neue hinzu. Da diese Plugins ebenfalls unter der liberalen MIT-Open-Source-Lizenz vorliegen, sind sie exzellente Ausgangspunkte für Eigenentwicklungen, etwa um hausseigene, proprietäre Systeme anzubinden.
- *Große, hilfsbereite Anwender- und Entwicklergemeinde:*
Mit über 180 Committern (15 an der Kernanwendung, die weiteren schreiben Plugins), dürfte Hudson eines der CI-Systeme mit der breitesten Entwicklerbasis sein. Bei einer strategischen Entscheidung – vor allem für den Einsatz eines Open-Source-Produkts – ist dies ein nicht unerheblicher Aspekt. Rund 18.000 ausgetauschte

Nachrichten auf Hudsons Mailinglisten im Jahr 2009 unterstreichen zusätzlich die Vitalität des Projekts.

Der Autor dieses Buches ist aktiver Evangelist und Committer im Hudson-Projekt. Sie dürfen also eine engagierte Darstellung der Vorteile dieses Produktes erwarten. Trotzdem mag Hudson nicht für alle Teams optimale CI-Server sein. Insbesondere die kommerziellen Werkzeuge locken mit einer engeren Verzahnung in Produkte desselben Herstellers (z. B. IntelliJ TeamCity mit IDEA, Atlassian Bamboo mit JIRA, Microsoft Team Foundation Server mit Visual Studio). Am Ende von Kapitel 5 werden wir daher auch auf gängige Alternativen zu Hudson eingehen.

Statt sich nun aber monatelang auf die Suche nach dem idealen CI-System zu begeben, ist viel wichtiger, überhaupt erst einmal häufigere Integrationen einzuführen und eine gelebte CI-Kultur aufzubauen. Ob Hudson dabei nur übergangsweise als Türöffner fungiert oder zum ständigen Begleiter wird, ist nachrangig.

1.4 Warum dieses Buch?

Seit 2008 halte ich regelmäßig Vorträge zu Hudson auf Fachkonferenzen, bei Firmen und in Java User Groups. Ebenso regelmäßig kommt im Anschluss an diese Vorträge die Frage nach einer »richtigen Dokumentation« für Hudson auf.

Das sollte Sie stutzig machen: Seit wann wollen Anwender Handbücher lesen? Ausgerechnet für eine Software, die sich intuitive Bedienung und Benutzerfreundlichkeit auf die Fahnen geschrieben hat? Selbstverständlich werden die wichtigsten Funktionen und Plugins auf der Hudson-Homepage¹ erklärt. Es gibt hilfreiche Blog-Einträge im Internet und zwei vitale Mailinglisten für Anwender und Entwickler. Und wer es ganz genau wissen möchte, kann jederzeit den Quelltext inspizieren². Warum also ein Hudson-Buch?

Viele Anwender hatten das Zusammenpuzzeln von Informationshäppchen aus dem Internet leid und wünschten sich nicht nur Beschreibungen zu einzelnen Hudson-Funktionen, sondern auch konzeptionelles Hintergrundwissen zur CI sowie Erfahrungsberichte aus erfolgreichen CI-Einführungen. Erstaunlicherweise ist das Buchangebot für CI-Titel sehr überschaubar, z. B. [Duval07]. Andere Werke stellen das Thema vor, können aber aus Platzgründen nur sehr bedingt in

1. <http://hudson-ci.org>

2. Sie müssen dazu den Quelltext nicht herunterladen. Unter <http://fisheye.hudson-ci.org> können Sie diesen auch bequem im Webbrowser einsehen.

die Tiefe gehen [Hüttermann10, Larman10, Smart08, Hüttermann07, Clark04]. Titel explizit zu Hudson sucht man komplett vergeblich. Selbst auf dem internationalen Buchmarkt existieren momentan nur Veröffentlichungen in der Entstehungsphase, etwa das lesenswerte, aber bisher nur in Englisch verfügbare Open-Source-Buch von J.F. Smart [Smart10].

Umso mehr freute mich die Anfrage des dpunkt.verlages im Sommer 2009, »ob denn der deutsche Buchmarkt für ein Hudson-Buch reif wäre?«. Aus dieser Anfrage entwickelte sich das Ergebnis, das Sie nun in den Händen halten. Es soll Sie in folgenden Aspekten unterstützen:

■ *CI-Grundlagen:*

Eine umfassende Einführung in die Konzepte, Vorteile und Herausforderungen kontinuierlicher Integration. Durch die produktneutrale Darstellung bleibt dieser Teil unabhängig von ständig neu erscheinenden Systemen und Versionen relevant.

■ *Hudson für Einsteiger:*

Ein schneller Start mit Hudson, von der Installation über die Konfiguration bis hin zur Anwendung im Alltag

■ *Hudson für Fortgeschrittene:*

Anregungen zu Themen wie Build-Zeit-Optimierung, verteiltem Bauen oder der Entwicklung eigener Plugins

■ *CI einführen:*

Praxistipps zur erfolgreichen CI-Einführung in Ihrer Organisation, mit Fallstudien aus unterschiedlichen Unternehmensgrößen

Darüber hinaus soll dieses Buch nicht nur Ihren Kopf, sondern auch Ihren Bauch ansprechen und Ihnen Lust auf kontinuierliches Integrieren machen. Wie oft in Softwaretechnik stellt auch bei der CI der Mensch einen maßgeblichen Erfolgsfaktor dar. Wissen und Werkzeuge sind nutzlos, wenn CI nicht wirklich *gelebt* wird. In diesem Sinne wünsche ich Ihnen nicht nur großen Wissenszuwachs bei der Lektüre, sondern im Idealfall auch gute Unterhaltung.

1.5 Wer dieses Buch lesen sollte

Dieses Buch wurde primär für Softwareentwickler und -architekten konzipiert, dürfte aber auch IT-Projektleitern und Informatikstudierenden nützen:

Was bringt Ihnen dieses Buch?

■ *Entwickler und Architekten:*

In diesem Buch lernen Sie Werkzeuge und Verfahren kennen, die Sie von lästigen Routineaufgaben entlasten und Ihnen mehr Zeit

für die kreativen und kniffligen Aufgaben der Softwareerstellung lassen. Sie werden sich mit Hudson ein zuverlässiges Frühwarnsystem einrichten, das Sie Probleme beheben lässt – längst bevor sie bei den Kollegen der Qualitätssicherung, Ihren Chefs oder Kunden aufschlagen.

■ *IT-Projektleiter:*

Softwareentwicklung steckt voller Überraschungen. Ihr Releaseprozess sollte keine davon sein. Sie werden Werkzeuge und Verfahren kennenlernen, die Ihnen helfen, Risiken zu minimieren und Qualität zu steigern. Sie werden CI erfolgreich in Ihrer Organisation einführen, weil Sie mit diesem Buch auf bewährten Vorgehensweisen und dem bereits bezahlten Lehrgeld anderer Teams aufbauen können.

■ *Informatikstudierende:*

In diesem Buch lernen Sie Werkzeuge und Verfahren kennen, deren Wert zwar im Prinzip unbestritten, aber längst noch nicht Standard in allen Entwicklungsteams ist. Ob als Praktikant oder baldiger Berufseinsteiger: Mit Ihrem CI-Wissen haben Sie eine Trumpfkarte im Ärmel, die Ihnen auch als »Junior« den Respekt Ihrer Kollegen einbringen kann. Hudsons geringer Ressourcenbedarf und seine hohe Anpassungsfähigkeit erlauben problemlos eine Einführung »nur mal als Experiment«, ohne gleich eine bestehende Infrastruktur komplett auf den Kopf stellen zu müssen.

Benötigtes Vorwissen

Idealerweise haben Sie bereits Erfahrung in Softwareprojekten sammeln können und die Höhen und Tiefen einer Produktentwicklung mit erlebt. Die meisten Beispiele in diesem Buch verwenden als Technologiestapel Java, Maven und Subversion. Sie benötigen lediglich Grundkenntnisse in diesen Technologien, um dem vorliegenden Buch folgen zu können. Trotzdem sei Ihnen an dieser Stelle eine hervorragende Einführung in das Konfigurationsmanagement ans Herz gelegt [Popp09]. Der optimale Einsatz eines Versionsmanagementsystems und die Erstellung von Build-Skripten mit Ant oder Maven sind Gebiete, die problemlos den Rahmen dieses vorliegenden Werkes sprengen würden.

*Windows, Linux oder
Mac OS X?*

Hudson ist in Java implementiert und daher auf vielen Plattformen verfügbar, darunter auch Windows, Linux, Mac OS X. Rund zwei Drittel aller Hudson-Server werden unter Linux betrieben. Auf den Entwicklerarbeitsplätzen dominiert hingegen meist Windows. Die Beispiele in diesem Buch sind daher aus der Perspektive eines Entwicklers mit einem Windows-System gewählt. Durch Javas Plattformunabhängigkeit ist dies aber nur an wenigen Stellen von Bedeutung. Sie werden im Buch auf diese wichtigen Unterschiede jeweils hingewiesen werden. Auch softwareentwickelnde Mac-OS-X-Anwender dürften keine Probleme haben, sich an den Windows-Beispielen zu orientieren.

1.6 Wie man dieses Buch lesen sollte

Dieses Buch ist so strukturiert, dass Sie es linear von vorne bis nach hinten lesen können und dabei von den Grundlagen zu immer fortgeschritteneren Themen geführt werden.

Sollten Sie jedoch wenig Zeit haben, möchte ich Ihnen untenstehende Lesepfade mit passenden Abkürzungen empfehlen (Abb. 1–2). Die ausgelassenen Kapitel können Sie dann später, je nach persönlichem Bedarf, in Angriff nehmen.

Wenn Sie wenig Zeit haben...

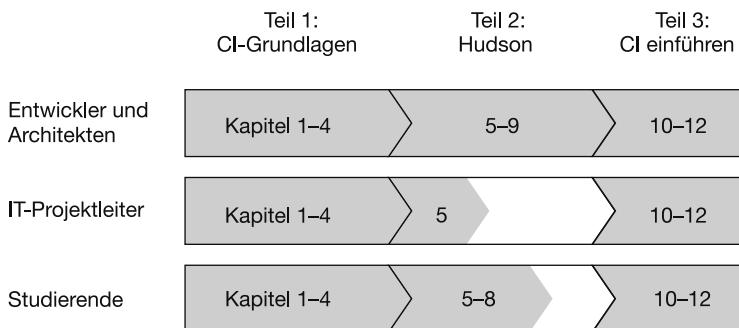


Abb. 1–2
Empfohlene Lesepfade für die Zielgruppen dieses Buches

■ *Entwickler und Architekten:*

Beginnen Sie mit den CI-Grundlagen in Teil 1 und lernen Sie Hudson in Teil 2 komplett kennen. Überfliegen Sie abschließend Teil 3, um einen Eindruck von den organisatorischen Aspekten einer CI-Einführung zu bekommen.

■ *IT-Projektleiter:*

Beginnen Sie mit den CI-Grundlagen in Teil 1 und lernen Sie Hudson in Teil 2 steckbriefartig kennen. Da Sie Hudson hauptsächlich als Beobachter verwenden werden, überspringen Sie die technischen Details. Die organisatorischen Aspekte in Teil 3 sind für Sie relevanter. Sie sollten diesen Teil daher komplett lesen.

■ *Studierende:*

Beginnen Sie mit den CI-Grundlagen in Teil 1. Lernen Sie danach Hudson in Teil 2 detailliert kennen, sparen sich jedoch die ganz fortgeschrittenen Themen wie etwa Plugin-Entwicklung für später auf. Überfliegen Sie abschließend Teil 3, um einen Eindruck von den organisatorischen Aspekten einer CI-Einführung zu bekommen.

1.7 Konventionen

*Deutsche und englische
Fachbegriffe*

In diesem Buch finden Sie überwiegend deutsche Fachbegriffe. Sollte keine allgemein akzeptierte deutsche Übersetzung existieren oder der Bezug zur Dokumentation eines Werkzeuges durch die Übersetzung erschwert werden, wurden die englischen Begriffe belassen. Deutschen Übersetzungen ist bei der ersten Verwendung der englische Originalbegriff in Klammern (*brackets*) nachgestellt.

Typografie

Der *kursive Schriftschnitt* hebt wichtige Stellen hervor. Texte, die der Benutzeroberfläche Hudsons entnommen wurden (z.B. *Hudson → Hudson verwalten*), sind ebenfalls kursiv gestellt.

Ausschnitte aus Quelltexten oder der Kommandozeile sind in der Schriftart *LetterGothic* dargestellt. Passen Quelltexte oder Befehle nicht in eine Zeile, wurden Sie mithilfe des Zeichens ⇒ umbrochen:

Dieser Text sollte in einer langen, langen, langen ⇒
Zeile eingegeben werden.

1.8 Website zum Buch

Auf der Website zum Buch, www.ci-mit-hudson.de, finden Sie Fehlerkorrekturen, ein Literaturverzeichnis mit »anklickbaren« Links sowie die Quelltexte zur Plugin-Entwicklung aus Kapitel 9.

1.9 Danksagungen

Mein herzlicher Dank gilt zunächst meinem Lektor René Schönenfeldt. Er ermunterte mich nicht nur erfolgreich zu diesem Buchprojekt, sondern verlor auch bei grotesken Terminverspätungen meinerseits nie die Fassung.

Großer Respekt gebührt außerdem den aufmerksamen und kritischen Lesern der Betaversionen dieses Buches: Ullrich Hafner, Michael Hüttermann, Jan Matérne, Hans-Peter Seitz, Martin Stransfeldt, Jürgen Walter, Lorenz Wiest sowie den unbestechlichen anonymen Gutachtern! Mario Ernst danke ich für seine Recherche zum Thema »Blau/Grün« im japanischen Kulturreis.

Am meisten stehe ich jedoch in der Schuld meiner Familie, allen voran meiner Frau Evelyn und meiner Tochter Floriane, die in den letzten Monaten viel zu oft auf Ehemann und Papa verzichten mussten. Ohne Euer Verständnis gäbe es dieses Buch nicht. Danke.

Simon Wiest
Gomaringen, im Oktober 2010

1.10 Zusammenfassung

Dieses Kapitel eröffnete Ihnen einen ersten Einblick in die Welt der CI: Sie haben erfahren, welche Probleme CI lösen möchte, und Sie haben Hudson als verbreitetes CI-System kennengelernt. Sie wissen, was Sie als Entwickler, IT-Projektleiter oder Studierender in diesem Buch erwartet, und kennen Ihren optimalen Lesepfad.

Im nächsten Kapitel definieren wir genauer, was Continuous Integration ist – und was nicht. Anschließend statteten wir »unserem« Entwicklungsteam vom Beginn dieses Kapitels zwei Besuche ab, jeweils vor und nach der CI-Einführung. Ähnlichkeiten mit Ihrem Alltag sind nicht ausgeschlossen ...

2 CI in 20 Minuten

Erinnern Sie sich noch an die Integrationshölle, die das Softwareteam zu Beginn des vorausgegangenen Kapitels durchschreiten musste? Wie Sie richtig vermuten, könnte CI viele der beschriebenen Probleme lösen.

In diesem Kapitel lernen Sie daher zunächst das CI-Konzept in einer knappen Definition kennen: Was also *ist* CI? Wir fragen aber auch: Was ist CI *nicht*? Anschließend kehren wir zu »unserem« Softwareteam in einem Vorher-Nachher-Vergleich zurück und überprüfen, wie CI dieses Team im Alltag unterstützen kann.

Nach der Lektüre dieses Kapitels kennen Sie also die Grundzüge der CI und haben einen ersten Eindruck von deren Vorteilen, aber auch von den notwendigen Voraussetzungen. Sie schaffen sich damit eine gute Grundlage für die darauffolgenden zwei Kapitel, die dann auf Vorteile und Voraussetzungen im Detail eingehen werden.

2.1 Was ist CI?

Zahlreiche Softwareteams dürften unabhängig voneinander die Idee des kontinuierlichen Integrierens »erfunden« haben. Populär geworden ist sie unter dem Namen »Continuous Integration« jedoch erst als eine der Praktiken der Extremprogrammierung (*eXtreme Programming*, kurz: XP), wie beispielsweise durch Kent Beck beschrieben [Beck99].

Begriffsbildend dürfte letztendlich Martin Fowlers Artikel »Continuous Integration« gewesen sein, in dem er das Kernkonzept folgendermaßen beschreibt:

»Die Kontinuierliche Integration ist eine Softwareentwicklungspraktik, bei der Teammitglieder ihre Arbeit häufig integrieren. Üblicherweise integriert jede Person im Team mindestens einmal täglich was zu mehreren Integrationen am Tag führt. Jede Integration wird durch einen vollautomatisierten Build (und Test) geprüft, um Fehler so

CI nach Martin Fowler

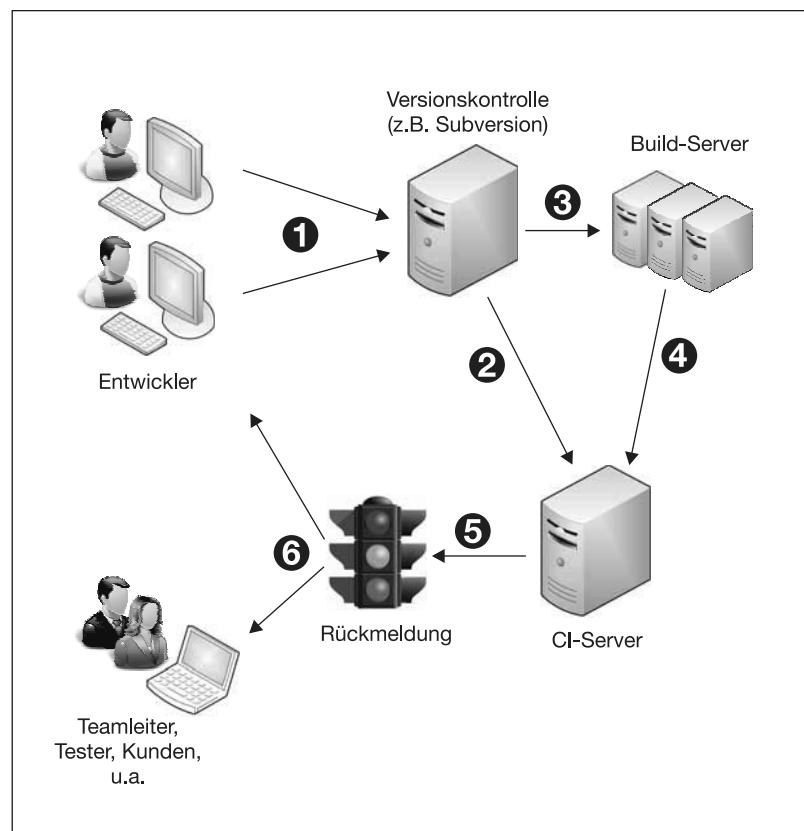
schnell wie möglich aufzudecken» ([Fowler00], aus dem englischen Original übersetzt).

Im Kern geht es also darum, die Integration aller Codeänderungen nicht mehr als eine einmalige Phase am Ende eines Entwicklungsprojekts zu betrachten. Vielmehr soll diese »Endmontage« einer Software viel häufiger erfolgen, typischerweise sogar mehrfach am Tag. Im Idealfall wird die Integration damit zum »Nicht-Ereignis«, da sie ohnehin ständig stattfindet. Vorteil: Wer mehrmals täglich den Ernstfall probt, hat auch keine Angst mehr davor.

Der Entwicklungsprozess mit CI

Wie sieht also der Entwicklungsprozess in einem minimalen CI-System aus? Betrachten wir dazu den Ablauf der kontinuierlichen Integration (CI) in Abbildung 2-1:

Abb. 2-1
Ablauf der
kontinuierlichen
Integration (CI)



Neuer Code entsteht zunächst auf den Rechnern der Entwickler. Nach eingehender lokaler Überprüfung durch den Entwickler werden alle Codeänderungen an ein zentrales Versionskontrollsystem übertragen (1). Dieser Vorgang wird vom CI-Server bemerkt (2), der daraufhin

einen neuen Build auf Build-Servern veranlasst (3). Nach Abschluss des Builds werden die Ergebnisse (Programme, Testberichte, Dokumentationen usw.) auf den CI-Server zur Auswertung und Archivierung übertragen (4). Über geeignete Kommunikationskanäle (E-Mail, RSS, Issue-Tracker usw.) meldet der CI-Server den Ausgang des Builds an die Entwickler zurück (6). Diese bekommen so eine zeitnahe positive Bestätigung ihrer Arbeitsergebnisse – oder aber einen Hinweis auf neu aufgetretene Fehler. In diesem Falle kann der Entwickler sofort mit der Verbesserung seines Codes beginnen: Der CI-Zyklus beginnt von vorne. Darüber hinaus können auch andere interessierte Parteien den aktuellen Stand der Softwareentwicklung verfolgen, z.B. Teamleiter, Tester, Kunden usw.

CI-Systeme haben in der Praxis zahlreiche Schnittstellen zu weiteren IT-Systemen (z.B. zentrales Benutzerverzeichnis, Test- und Produktionsserver, Software-Repositories). In kleineren Arbeitsgruppen werden die benötigten Dienste oftmals auf derselben Hardware ausgeführt, in großen Unternehmen werden hingegen eigens dafür vorgesehene Server eingesetzt. Das Funktionsprinzip ist jedoch das gleiche und setzt hinsichtlich des Entwicklungsprozesses gewisse Rahmenbedingungen voraus. Genau betrachtet können diese fast vollständig aus der Anforderung des häufigen Bauens (und Testens) abgeleitet werden. Fowler zählt in seinem Artikel folgende zehn Praktiken als Voraussetzung für wirkungsvolles CI auf:

Voraussetzungen für CI

1. Gemeinsame Codebasis:

Alle Daten, die in ein Produkt eingehen, werden an einem gemeinsamen Ort verwaltet, typischerweise in einem Versionskontrollsystem.

2. Automatisierter Build:

Das Produkt muss vollautomatisch aus seinen Grundbestandteilen übersetzt und zusammengebaut werden können.

3. Selbsttestender Build:

Entstandene Produkte werden während des Builds automatisch auf korrekte Funktionsweise überprüft.

4. Häufige Integration:

Entwickler integrieren ihre Arbeitsergebnisse mindestens einmal pro Tag. Anders ausgedrückt: Sie checken mindestens einmal am Tag ihren Stand in das Versionskontrollsystem ein.

5. Builds (und Tests) nach jeder Änderung:

Nach jeder Änderung wird vollautomatisch gebaut und getestet. Sind die Änderungen klein, lassen sich so neue Fehler sehr schnell in den korrespondierenden Änderungen auffinden.

6. *Schnelle Build-Zyklen:*

Zwischen dem Einchecken einer Änderung und der Rückmeldung durch das CI-System sollte möglichst wenig Zeit vergehen. Idealerweise liegt diese Dauer im Minutenbereich.

7. *Tests in gespiegelter Produktionsumgebung:*

Tests sollten in einer möglichst realitätsnahen Umgebung stattfinden. In der Konsequenz muss also auch der Aufbau dieser Testumgebungen automatisiert werden.

8. *Einfacher Zugriff auf Build-Ergebnisse:*

Der letzte Stand des Produkts sollte für alle Beteiligten einfach zugänglich sein. Dies gilt nicht nur für Entwickler, sondern auch für Tester, Kundenberater, Projektleiter usw.

9. *Automatisierte Berichte:*

Die Ergebnisse der Builds müssen vollautomatisch »gebrauchsferdig« aufbereitet werden. Zum einen beinhaltet dies das Verschicken von Benachrichtigungen, zum anderen auch die Visualisierung von archivierten Informationen aus vergangenen Builds.

10. *Automatisierte Verteilung:*

Ein CI-System sollte nicht nur den Softwareerstellungsprozess automatisieren, sondern auch die Verteilung zu den jeweiligen Anwendern bzw. das Ausbringen auf Test-, Demonstrations- und Produktionsserver.

Wenn Sie sich die Liste der zehn Praktiken betrachten, fragen Sie sich vermutlich, ob hier nicht zu viel vorausgesetzt wird? Muss nicht zu viel umgestellt werden? Lohnt sich dieser Aufwand?

Um es vorwegzunehmen: Es lohnt sich auf jeden Fall! Jede dieser Praktiken ist bereits für sich isoliert gesehen wertvoll und wird vielleicht sogar bereits in Ihrer Arbeitsgruppe praktiziert. Erst in der Summe hingegen ermöglichen sie einen runden Entwicklungsprozess, der auf hoher Frequenz kontinuierlich wechselt zwischen Entwickeln, Einchecken, Bauen, Testen, Berichten und erneut Entwickeln, Einchecken, Bauen, ...

2.2 Was ist CI *nicht*?

Wie wir im vorausgegangenen Abschnitt gesehen haben, beinhaltet CI eine weitestgehende Automatisierung typischer Arbeitsschritte der Softwareentwicklung, z. B. Kompilieren, Testen, Archivieren, Verteilen, Berichten (denken muss man glücklicherweise noch selber). Kein Wunder also, dass kaum ein anderer Dienst mit so vielen anderen IT-Systemen in Kontakt steht wie ein CI-Server: vom Versionskontrollsyst-

tem bis zum E-Mail-Server, von der Benutzerverwaltung bis zum Produktionsserver.

Im Zusammenspiel dieser IT-Systeme übernimmt der CI-Server sozusagen die Rolle des Dirigenten im »Build-Orchester«. Sicher – man kann auch ohne CI brauchbare Software erstellen. Und genauso würden die Berliner Philharmoniker auch ohne Dirigent locker vom Blatt musizieren können. Ungleich besser wird das Ergebnis jedoch unter der Leitung eines Sir Simon Rattle. Rattle selbst greift dabei weder zur Bratsche noch zur Oboe. Stattdessen überlässt er dies den jeweiligen Virtuosen. Trotzdem ist sein »Mehrwert« so groß, dass er als Dirigent auf Konzertplakaten zuerst aufgeführt wird. Analog gilt für Continuous Integration:

Das Build-Orchester

- CI ist *keine Programmiersprache* wie C/C++, Python oder Java. Sie steuert aber die entsprechenden Compiler, um Quelltexte zu übersetzen.
- CI ist *kein Build-Werkzeug* wie Make, Ant oder Maven. Sie ruft aber typischerweise solche Werkzeuge auf.
- CI ist *kein Versionskontrollsystem* wie Subversion, CVS, Git oder Perforce. Sie kommuniziert aber mit ihnen, um von Codeänderungen der Entwickler zu erfahren.
- CI ist *kein Test-Framework* wie JUnit, TestNG oder Selenium. Sie ruft aber solche Frameworks auf, um übersetzte Programme zu testen.
- CI ist *kein Werkzeug für statische Codeanalyse* wie PMD, Checkstyle oder FindBugs. Sehr wohl aber werden diese Werkzeuge in einem CI-Build aufgerufen und deren Ergebnisse ausgewertet.
- CI ist *kein Repository* für erzeugte Artefakte wie Nexus oder Artifactory. Sie kann aber Produkte eines CI-Builds dort ablegen und so anderen IT-Systemen und Entwicklern bereitstellen.
- CI ist *kein einzelnes Produkt*, sondern integriert vielmehr eine Vielzahl von Technologien für die jeweiligen Arbeitsschritte der Softwareerstellung (Compiler, Build-Werkzeuge, Testverfahren usw.)
- CI ist *keine markengeschützte Methode*, für die sich Zertifikate sammeln lassen. In der agilen Softwareentwicklung ist CI jedoch nicht wegzudenken und auch in anderen Methoden sinnvoll einsetzbar.

Und: CI ist kein Allheilmittel gegen schlechten Code, fehlerhafte Software und enge Projektzeitpläne. Sie hilft aber, Probleme schneller zu entdecken und früher zu beheben. Dies spart Zeit in nachgelagerten Phasen und vergrößert den zeitlichen Puffer für Unvorhergesehenes (»Alle mal herhören: Unser Kunde hat gerade angerufen ...«).

Nachdem wir uns vom CI-Ablauf und der eingesetzten Technik ein Bild gemacht haben, wollen wir in den nächsten zwei Abschnitten CI an einem Vorher-Nacher-Beispiel betrachten.

2.3 Software entwickeln ohne CI

Zu Beginn des vorangegangenen Kapitels hatten wir ein Softwareteam in der heißen Phase der Release-Integration besucht. Erinnern Sie sich noch an die Tiefpunkte, mit denen dieses Team ohne Continuous Integration zu kämpfen hatte? Hier ein »Worst of«:

*Integration als
besondere Aktivität*

»Am kommenden Tag soll Ihre Abteilung die längst überfällige neue Version Ihrer Software ausliefern ...« – Die Integration wurde bis auf den letzten Drücker hinausgeschoben, weil sie nicht als Teil des Entwicklungsprozesses betrachtet oder im Eifer der Codierung verdrängt wurde (»Der Kunde bezahlt uns fürs Programmieren und nicht fürs Dateien zusammenkopieren.«).

Späte Integration

»Die Entwickler werfen also eilig auf dem Abteilungsserver ihre Codeänderungen ab, die sie in den letzten Wochen auf ihren Rechnern erbrüitet haben.« – Code wurde über Tage und Wochen in Silos lokal entwickelt. Schnittstellen und Abhängigkeiten diffundieren dabei zwangsläufig auseinander, wenn nicht immenser Abstimmungsaufwand betrieben wird (»Oh je – alle in den Teamraum! Wir müssen ganz dringend noch ein Meeting machen.«).

*Tests als
»Code zweiter Klasse«*

»Die (spärlich) vorhandenen Tests schlagen fehl, manchmal aber auch nicht.« – Die Tests deckten nur geringe Teile des zu prüfenden Codes ab. Zusätzlich trugen unzuverlässige Tests das Übrige bei, um das Vertrauen in Testberichte zu erschüttern. Sicherer Alarmsignal hierfür sind Dialoge nach folgendem Schema: Projektleiter: »Ist das Feature komplett?« Entwickler: »Die automatischen Tests laufen eigentlich durch, aber die Tester müssten sich das trotzdem nochmals gründlich anschauen ...«

*Keine Builds auf
neutralem Grund*

»Bei mir läuft's aber! – Ich habe dort nichts geändert!« – Jeder Entwickler arbeitet auf seinem Rechner in seinem persönlichen lokalen Ökosystem aus Werkzeugen, Einstellungen und ausgecheckten Quelltexten. Kein Wunder, dass auf unterschiedlichen Rechnern unterschiedliche Ergebnisse beobachtet werden können. Wer aber hat Recht, wenn es keine verbindliche und neutrale Instanz gibt, die hier entscheidet?

*Lange Build- und
Test-Zyklen*

»Am späten Abend laufen die Tests endlich durch.« – Lange Build-Zeiten bedeuten weniger Chancen pro Tag, um geänderten Code zu bauen und zu testen. Das Resultat ist Programmierung im Stile von »Commit and Run«: Kurz vor Arbeitsende einchecken, dann schnell den Arbeitsplatz verlassen und am nächsten Morgen in den Trümmern

nachschauen, wie der nächtliche Build mit den Änderungen klargemkommen ist. So griffig dieses Konzept sein mag, so stark limitiert es die Anzahl an Rückmeldungen zum Entwickler – in diesem Fall auf eine pro Tag.

»*Als langjähriger Mitarbeiter mit Kopfmonopol kennt der Meister als Einziger die geheimen Schritte, die notwendig sind, um eine Distribution zu erstellen – und ja, nur er hat die notwendigen Werkzeuge dazu auf seinem Rechner installiert.*« – Teile des Erstellungsprozesses sind nicht dokumentiert, werden manuell ausgeführt und können nur in bestimmten Rechnerumgebungen stattfinden. Folgerichtig tauchen im Ablauf der Softwareerstellung immer wieder Flüchtigkeitsfehler durch menschliche Nachlässigkeit auf. Im besten Falle sind sie nur lästig, im schlimmsten Falle aber fatal – denn durch die Abhängigkeit von wenigen Köpfen und Rechnern hat man gleich mehrere K.o.-Punkte (*single points of failure*) geschaffen.

»*Das Team hat Bauchschmerzen, denn keiner weiß wirklich genau, welche Änderungen in dieses Release eingeflossen sind.*« – Von Hand erzeugte Dokumentation kann selten mit dem Tempo der Änderungen in einem Softwareprojekt mithalten. Im Ergebnis erhält man Dokumentation, die erfreulicherweise zu 80% auf dem neuesten Stand ist. Leider weiß man hingegen nicht, welche 20% der Dokumentation inzwischen veraltet sind ...

Geringe Automatisierung
und Dokumentation
kritischer Schritte

Geringe Automatisierung
der Dokumentation

2.4 Software entwickeln mit CI

Besuchen wir »unser« Softwareteam ein paar Monate später. Der hohe Leidensdruck in den Integrationsphasen führte zur Installation eines CI-Servers, der nun stetig das Zusammenspiel aller beteiligten Personen und IT-Systeme koordiniert.

Seit der Vollautomatisierung der Integration erstellt der CI-Server inzwischen vollautomatisch aus den Quelltexten des Versionsmanagementsystems das fertige Produkt – genau so, wie es der Kunde letztendlich erhält: Ein Installationsprogramm, eine Liste mit den Änderungen seit der letzten Version, eine aktuelle Dokumentation der externen Schnittstellen sowie ein Prüfprotokoll des vorgeschriebenen Testlaufs. Die Integration ist zum »Nicht-Ereignis« geworden.

Die Entwickler wurden in der Folge geradezu süchtig nach dem »schnellen Erfolgserlebnis« und checken nun ihre Änderungen nach eingehender lokaler Prüfung sehr viel häufiger ein.

Durch zusätzliche Tests konnte die Codeabdeckung erhöht werden. Vor allem aber wurden unzuverlässige Tests überarbeitet, so dass deren Ausgang nun nicht mehr von veränderlichen Einflüssen wie

Integration als
»Nicht-Ereignis«

Häufige Integration,
häufiges Erfolgserlebnis

Tests als »Code erster
Klasse«

*Builds auf neutralem
Grund*

Netzwerkbandbreite oder der momentanen Auslastung des E-Mail-Servers abhängt. Und weil auf Testberichte wieder Verlass ist, erhalten fehlgeschlagene Tests sofortige Aufmerksamkeit.

*Kurze Build- und
Test-Zyklen*

Immer noch kommt es vor, dass Entwickler auf unterschiedlichen Rechnern zu unterschiedlichen Ergebnissen kommen. Entscheidend ist aber nun, was das CI-System nach Ausführung der Tests befindet. Somit gibt es endlich einen neutralen Schiedsrichter, der – frei von Skrupel und Häme – aufgetretene Probleme und fehlgeschlagene Tests mitteilt.

*Vollautomatisierung
kritischer Schritte*

Die Build-Zeit konnte durch geschickte Verteilung und Parallelisierung auf mehrere Rechner auf unter 15 Minuten gedrückt werden. Dadurch wurde der Taktenschlag ganz erheblich erhöht und langes »Codieren im Nebel« abgeschafft.

*Automatische Erstellung
der Dokumentation*

Durch die Automatisierung kann inzwischen selbst der Praktikant eine aktuelle Produktversion zu Demonstrationszwecken für den Vertrieb erstellen. Die erfahrenen Entwickler haben damit auch endlich wieder mehr Zeit, sich um *wirklich* knifflige Probleme zu kümmern oder den Junior-Entwicklern Tricks aus ihrem Erfahrungsschatz beizubringen.

Die Schnittstellendokumentation wird endlich nicht mehr von Hand in einer Textverarbeitung geführt, sondern in jedem Build aufs Neue vollautomatisch aus den Quelltexten erzeugt. Seitdem hatte es wesentlich weniger Anrufe erboster Kunden gegeben, die – streng der *veralteten* Dokumentation folgend – Fehlermeldungen beim Aufruf nicht mehr vorhandener Funktionen erhalten hatten.

Gruppenstolz

Im Abteilungsflur fallen drei niedliche Lampen in Form von Gummibären auf. Sie sind in den Ampelfarben Rot, Gelb und Grün aufgestellt und zeigen rund um die Uhr die Testergebnisse des letzten Projekt-Builds an. Zurzeit leuchtet der grüne Bär, was selbst für Branchenfremde als gutes Zeichen interpretierbar ist. Ein dankbarer Entwickler huscht vorbei und stellt vor dem grünen Bären ein kleines Schälchen mit Süßigkeiten ab ...

2.5 Zusammenfassung

In diesem Kapitel haben Sie in kompakter Form den Ablauf und die erforderlichen Rahmenbedingungen für effektive CI kennengelernt. Ein CI-Server koordiniert dabei als Dirigent des »Build-Orchesters« das Zusammenspiel der beteiligten Virtuosen wie Compiler, Build-Werkzeug, Versionskontrollsysteem, Test-Framework usw. Im Vorher-Nachher-Vergleich konnten Sie bereits erahnen, welches Potenzial die Einführung von CI in der Softwareentwicklung freisetzen kann.

Vielleicht können Sie es inzwischen kaum erwarten, CI auch in Ihrer Arbeitsgruppe einzuführen. Was wird also benötigt? Im folgenden Kapitel werden wir die erforderlichen Rahmenbedingungen nochmals im Detail beleuchten.

3 Welche Vorteile bringt CI?

In den vorausgegangenen Kapiteln haben wir bereits einige Vorteile der CI kennengelernt. In diesem Kapitel komplettieren wir die Aufzählung dieser Vorteile durch eine systematische und ausführliche Betrachtung. Dies dürfte besonders Projektleiter ansprechen, da es hier weniger um die technischen Fragen geht, sondern um den geschaffenen wirtschaftlichen Mehrwert.

Die zwei offensichtlichen Vorteile sind dabei sicher »Reduzierte Risiken« und »Verbesserte Produktqualität«. Wir werden aber auch noch weitere wünschenswerte finden, die vielleicht erst auf den zweiten Blick erkennbar sind.

3.1 Reduzierte Risiken

Verringertes Risiko dürfte der wichtigste Grund sein, warum CI in keiner professionellen Softwarewerkzeugkette fehlen sollte. Durch häufige, reproduzierbare und vollautomatische Integrationen vermeidet man große Überraschungen am Ende eines Softwareprojekts. Martin Fowler beobachtet dies sehr treffend in seinem Artikel über Continuous Integration [Fowler00]: Eine späte Integration, die zum Ende eines Softwareprojekts ausgeführt wird, hat nach Fowler (mindestens) drei unerwünschte Eigenschaften:

*Späte Integration – die
große Unbekannte*

- Vor der Integration ist unklar, wie lange sie dauern wird.
- Während der Integration weiß man nicht, wie viel man noch vor sich hat (oder es sind konstant die berühmten fehlenden 20%).
- Die Integration findet am Projektende statt – wenn also die meisten Projekte den Zeitplan bereits überzogen haben und die Neuen bereits blank liegen.

So betrachtet, kann man sich eine späte Integration als einen dunklen Tunnel unbekannter Länge vorstellen, durch den ein Projekt im Moment der höchsten Anspannung geschoben werden muss. Dass dies

für alle Beteiligten – und auch für das Produkt selbst – nicht förderlich sein kann, leuchtet ein. Durch eine kontinuierliche, begleitende Integration hingegen wird diese zu einem »Nicht-Ereignis«. Die große Unbekannte am Ende eines Projekts wird dadurch eliminiert.

3.2 Verbesserte Produktqualität

*Öfter testen,
weniger Fehler*

Die Qualität des Softwareprodukts wird durch CI ganz erheblich gesteigert, denn es rutschen weniger Fehler in das Endprodukt durch. Der Grund liegt darin, dass ein gutes CI-System früher, öfter, gründlicher und umfangreicher testet, als dies bei einer manuellen Vorgehensweise möglich ist:

- *Früher*, weil automatische Tests bereits nach jeder kleinen Änderung durchgeführt werden und nicht erst am Ende eines Projekts. Dies verschafft Entwicklern mehr Zeit zur Reaktion. Außerdem hat sich vielfach gezeigt: Je früher ein Fehler erkannt wird, desto kostengünstiger ist seine Behebung.
- *Öfter*, weil jede Änderung im Versionsmanagement einen neuen Build und somit neue Testläufe auslöst.
- *Gründlicher*, weil das Produkt stets denselben Testparcours bestehen muss. »Clevere Abkürzungen« gibt es in diesem Ablauf nicht (»In diesem Modul haben wir ja nichts wirklich Wichtiges verändert – das müssen wir also nicht schon wieder durchklicken..«).
- *Umfangreicher*, weil durch die Automatisierung der Tests eine Codeabdeckung erreicht werden kann, die bei manueller Arbeitsweise wirtschaftlich schwer vertretbar wäre.

Broken-Windows-Theorie

Ein etwas subtilerer Punkt ist die veränderte Wahrnehmung neu auftretener Probleme: In einem guten CI-System ist es der Normalzustand, dass Produkte jederzeit alle Qualitätsziele erreichen. Selbst wenn der angepeilte Funktionsumfang noch nicht komplett erreicht wurde, lässt sich das Produkt jederzeit bauen, laufen alle Tests erfolgreich durch und werden alle geforderten Codemetriken erfüllt. Vor dem makellosen Hintergrund eines fehlerfreien Builds sind neu auftretende Probleme wesentlich besser sichtbar, als wenn zu einer großen Anzahl bereits bestehender Probleme lediglich weitere hinzukommen. Dieses Phänomen wird in der Psychologie in der Broken-Windows-Theorie untersucht. Sie beschreibt, wie eine an sich harmlose Tatsache, etwa ein eingeschlagenes Fenster in einem leerstehenden Haus, zur vollen Demolierung des Gebäudes führen kann. Ihre Alarmglocken sollten also bei dem folgenden Satz läuten: »Ach ja, der Compiler wirft immer rund 300 Warnungen aus – das ist normal.« Zudem neigen Feh-

ler, die in Rudeln auftreten, zu unangenehmen Abhängigkeits- und Verdeckungseffekten. Dies macht deren Behebung aus wirtschaftlicher und psychologischer Sicht zusätzlich frustrierender und erhöht somit die Chance des »Durchrutschens« ins Endprodukt. Die Behebung eines neu aufgetretenen Fehlers sollte daher Vorrang vor der Weiterentwicklung neuer Funktionsmerkmale haben, um ein ungutes Aufstauen von Problemen zu vermeiden.

Häufiges Bauen und Testen erleichtert aber auch die Behebung vereinzelt auftretender Probleme. Wird öfter gebaut, sind die Veränderungen in den Quelltexten zwischen zwei Builds weniger umfangreich – mit anderen Worten: Es gibt weniger Stellen, an denen etwas schiefgehen kann. Da diese »Deltas« sich vom Versionskontrollsystem anzeigen lassen, beschleunigt dies die Fehler suche enorm.

Selbstverständlich ist ein angemessener Umfang an automatischen Test die unabdingbare Voraussetzung, um die beschriebenen Vorteile zu erzielen: Es können nur Schwachstellen gefunden werden, nach denen auch getestet wird. Immerhin lässt sich durch Erhebung der Codeabdeckung messen, wo sich noch weiße Flecken in Ihrem Testplan befinden. Wir werden im nachfolgenden Kapitel auf die Besonderheiten und typischen Schwierigkeiten eingehen, die bei der Erstellung angemessener Tests zu beachten sind.

Schnellere Eingrenzung von Fehlerstellen

Testumfang ist entscheidend.

3.3 Allzeit auslieferbare Produkte

Eine der sichtbarsten Verbesserungen nach einer CI-Einführung dürfte die ständige Verfügbarkeit eines auslieferbaren Produktes sein. Wurde neben dem Bauen und Testen eines Produkts auch dessen Installation auf einem Demonstrationssystem automatisiert, kann der aktuelle Stand jederzeit auch von »nichttechnischem Personal« begutachtet werden, etwa von Projektleitern, Fachtestern, dem Management usw. CI ist damit die abschließende Antwort auf die sprichwörtliche Chef-Frage »Kann man schon was sehen?«

»Kann man schon was sehen?«

Die positiven Konsequenzen eines allzeit auslieferbaren Produktes sind vielfältig. Hier einige Beispiele:

Newer Möglichkeiten durch ein allzeit auslieferbares Produkt

- *Tester* können schnell mit aktualisierten Versionen versorgt werden und die korrekte Behebung von Problemen verifizieren. Offene Problemberichte können somit schneller abgeschlossen werden.
- *Vertriebsmitarbeiter* können sich ein Demonstrationssystem aufsetzen lassen, ohne dass dazu kostbare Entwicklerzeit abgezogen werden müsste. Die automatische Installation neuester Programmversionen kann auch – leicht modifiziert – zum Einspielen beliebiger Versionen und Datenstände wiederverwendet werden.

- *Trainer* können sich frühzeitig mit kommenden Funktionsmerkmalen vertraut machen und Schulungsmaterialien anpassen.
- *Kundendienstmitarbeiter* können sich neueste Versionen und Datenstände einrichten, um Probleme bei der Migration von Vorgängerversionen frühzeitig zu erkennen.
- *Projektleiter* können sich vom Stand des Projekts an einem laufenden System einen Eindruck verschaffen und das »Schaulaufen« in einer Anwendung für den nächsten Kundentermin üben.
- *Kunden* können den Fortschritt des Entwicklungsprojekts auf einem Kundensystem verfolgen. Dies ist sowohl in Form eines Demosystems realisierbar, das automatisch auf den neuesten Stand gebracht wird, als auch in Form eines Abnahmesystems, auf dem der Kunde die Gelegenheit zur Erprobung neuer Funktionsmerkmale erhält.

3.4 Gesteigerte Effizienz

CI setzt einen hohen Grad an Automatisierung voraus. Es mutet ironisch an, dass gerade Softwareentwickler, die für ihre Kunden hocheffiziente Werkzeuge zur Bearbeitung des Tagesgeschäfts erstellen, in ihrem eigenen Vorgehen den Automatisierungsgrad des Kunsthandwerks erreichen. Der Schuster hat hier oft tatsächlich die schlechtesten Sohlen!

Aufwand der Automatisierung lohnt auf lange Sicht immer.

Beim Umstieg auf einen CI-unterstützten Entwicklungsablauf stellt die vollständige Automatisierung der Build-Werkzeugkette meist die größte Hürde dar. Oft sind es gerade die letzten 10 Prozent, für die am meisten Aufwand getrieben werden muss. Doch die Mühe lohnt sich: Ein automatischer Build-Prozess ist schneller als jedes manuelle Vorgehen, bei dem Schritt für Schritt das Endprodukt zusammengestellt werden muss. Darüber hinaus werden Flüchtigkeitsfehler eliminiert und wertvolle Arbeitszeit für andere, anspruchsvolle Tätigkeiten freigesetzt. Selbst wenn die Ersparnis nur wenige Minuten pro Build beträgt, summiert sich diese im Laufe der Tage, Wochen und Monate auf erkleckliche Summen.

Produktiverer Start für neue Teammitglieder

Die Automatisierung spart aber auch Zeit an einer ganz anderen Stelle: Kommen neue Mitarbeiter zum Team hinzu, so müssen diese bei modernen Build-Werkzeugen nur noch sehr wenige manuelle Einstellungen vornehmen, um den ersten privaten Build auf ihrem Entwicklerrechner zu bauen. Die neuen Kollegen sind dadurch schneller produktiv ins Team integriert. Die »alten Hasen« hingegen müssen weniger Zeit damit zubringen, mühsam von Hand die verwendeten Bibliotheken, Werkzeuge und Einstellungen der Entwicklungsumgebung herauszusuchen.

Eine gelungene Automatisierung abstrahiert zudem den Build-Prozess von einer bestimmten Maschine. Ob also auf dem Arbeitsplatzrechner des Entwicklers, auf dem seines Kollegen oder einem »neutralen« CI-Server gebaut wird, spielt dann keine Rolle mehr: Es kommen immer dieselben Werkzeuge und Schritte zum Einsatz. Dies bildet die technische Grundlage zu verteilten Builds, welche die verfügbaren Ressourcen (vor allem Rechenzeit) optimal ausnützen können.

*Bessere
Ressourcennutzung durch
Automatisierung*

3.5 Dokumentierter Build-Prozess

Die geforderte Automatisierung schafft gleichzeitig die Notwendigkeit, alle Build-Schritte explizit in Build-Skripten oder in der Konfiguration des CI-Servers festzuhalten. Dies ist kein Nachteil der Methode – im Gegenteil: Damit wird in vielen Fällen erstmalig eine durchgängige Dokumentation des Build-Prozesses geschaffen, und verbleibende Lücken werden geschlossen. Nicht selten nämlich trägt der Build-Manager bei der Endmontage eines Softwareprodukts hier und da ein bisschen »Zauberstaub« in Form von selbst geschriebenen Skripten auf (»Für diesen Schritt setzt Martin – glaube ich – eines seiner Python-Skripte zur Umwandlung von ISO-8859-1 nach EBCDIC ein. Leider ist er gerade krank, so dass wir den Hot-Fix für den Kunden heute nicht einbauen können.«). Werden diese Skripte nicht als wichtiger Teil der Werkzeugkette verstanden (und damit auch nicht versioniert, gesichert und dokumentiert), kann ein Festplattendefekt im schlimmsten Falle für ein technisches K.o. eines ganzen Entwicklungsteams sorgen¹.

*Automatisierung erfordert
vollständige
Dokumentation*

Eine Besonderheit dieser Art der Dokumentation ist, dass sie ausgeführt wird. Dies erleichtert zum einen die Einarbeitung in den Build-Prozess, weil man die ausgeführten Schritte in der Oberfläche eines CI-Servers mitverfolgen kann. Zum anderen ist die Dokumentation immer aktuell, weil man ja genau die Einstellungen vor sich hat, die den Build-Prozess steuern. Dies ist ein nicht zu unterschätzender Vorteil gegenüber externen Dokumentationsformen (»Da gab's mal eine Wiki-Seite dazu«, »Das könnte in einer Word-Datei irgendwo auf dem Netzlaufwerk stehen«, »Im Teamraum in der Stuttgarter Filiale hängt dazu ein Diagramm an der Pinnwand«).

*Ausführbare
Dokumentation ist immer
aktuell.*

Die »ausführbare Dokumentation« hält mehr als nur die Schrittfolge des Build-Prozesses fest. Da auch Qualitätsziele automatisch überprüfbar sein müssen, sind diese explizit zu benennen: Was wird getestet? Wie viele Tests dürfen ausgelassen werden? Auf welche Code-

*Auch Qualitätsziele
werden dokumentiert.*

1. Schwarzmalerei? Glauben Sie mir, es kommt öfter vor, als man denkt. Nur spricht man selten gerne darüber ...

konventionen hin wird geprüft? Welche Codeelemente müssen dokumentiert sein?

3.6 Höhere Motivation

Die positiven psychologischen Auswirkungen, die CI auf Entwicklerebene entfalten kann, werden gerne übersehen (Vermutung: Glückliche Entwickler produzieren besseren Code).

Häufigere Rückmeldung,

häufigere

Erfolgserlebnisse

Häufigere Integration bedeutet für den einzelnen Entwickler auch häufigere Rückmeldung. Im besten Falle sind das natürlich Erfolgserlebnisse, wenn nach wenigen Minuten die eigene Änderung vom CI als wirksam bestätigt wird und wieder ein Ticket geschlossen werden kann. Manche Teams gehen sogar so weit, im CI-System für geglückte Builds an die beteiligten Entwickler Punkte zu verteilen. In diesen Gruppen wird es zu einer Frage der Ehre, dass die »eigenen Builds« nicht fehlschlagen dürfen.

Doch selbst eine negative Rückmeldung ist einer fehlenden vorzuziehen. Sie lässt einen Entwickler nicht im Unklaren darüber, ob vielleicht ein paar Wochen später ein »dickes Ende« auf ihn zukommen könnte ... Die stetige Rückmeldung des CI-Systems erlaubt es also dem Entwickler, zuversichtlich und mit voller Kraft an seinen Aufgaben zu arbeiten. Er muss keine Reserven aufsparen wie ein Marathonläufer, der noch kurz vor dem Zieleinlauf eine hohe Steilwand vermuten muss.

Nachhaltiges Tempo

Das häufige Einchecken trainiert gleichzeitig, in kleinen, überschaubaren Schritten in einem nachhaltigen Tempo zu arbeiten, statt in stressreichen Schlussspurts umfangreiche Codeänderungen nachzuarbeiten, bis sich diese endlich mit den Arbeitsergebnissen der Kollegen integrieren lassen.

Mutiger arbeiten mit Fangnetz

Gleichzeitig ermuntert der CI-Server als »Fangnetz« zu mutigen Änderungen, weil man die Gewissheit hat, dass jede Änderung vollautomatisch umfangreichen Tests unterzogen wird (die natürlich existieren müssen!). Dies ist eine Voraussetzung für langfristig wartbaren Code. Nur wenn man sich traut, bestehenden Code immer wieder an die Realität anzupassen, bleibt er beherrschbar (»Wie kann ich eigentlich den Kontostand ermitteln?« »Ganz einfach: Mit `account.getBalanceDM()`. Aber aufpassen: Die Methode liefert inzwischen Euro zurück!«).

Neutrale Rückmeldungen

Viele Entwickler tun sich zudem leichter mit negativen Rückmeldungen, die sie von einem Rechner statt von einer Person erhalten. Der CI-Server behandelt alle gleich und nimmt auch keine lautstarken, spontanen Äußerungen krumm – ein Kollege aus Fleisch und Blut eher schon (»So ein ..., immer hackt er auf meinen Einrückungen herum!«).

3.7 Verbesserter Informationsfluss

»Wie weit sind wir?« – »Schätzungsweise 80%.« – »Waren wir das nicht schon vor zwei Wochen?« – »Ja, aber wir hatten bisher keine Zeit, nochmals nachzurechnen.«

Durch das vollautomatische und häufige Bauen und Testen einer Anwendung liegen in einem CI-System stets aktuelle und vollständige Informationen zum Stand eines Softwareprojekts vor. Gute CI-Systeme stellen dabei erhobene Daten in gebrauchsfertiger Weise dar, etwa grafisch visualisiert und mit angemessenen Such- und Filterfunktionen.

Dies kann die Entscheidungsfindung innerhalb eines Projekts unterstützen (»Warum wird unser Build immer langsamer?«), aber auch die Koordination mit abhängigen Projekten erleichtern (»Wo kann ich die allerneuste Version eurer Bibliothek herunterladen?«).

Aktuelle, vollständige,
gebrauchsfertige
Informationen –
an einer Stelle

3.8 Unterstützte Prozessverbesserung

Schließlich stellt ein CI-System eine sehr effiziente Möglichkeit dar, wichtige Metriken *über den Verlauf* eines Projekts und darüber hinaus auch über mehrere Projekte eines Unternehmens zu sammeln. Durch die zusätzliche Zeitachse lassen sich aus diesen Daten Strategien zur Verbesserung des Entwicklungsprozesses ableiten, etwa die gezielte Aufrüstung von Hardwarekomponenten, Schulungsbedarf für bestimmte Testwerkzeuge oder die Ausweitung erfolgreicher Build-Werkzeuge auf alle Projekte eines Unternehmens.

Bessere Produkte, aber
auch bessere Prozesse

CI kann somit nicht nur die Qualität eines einzelnen Softwareprodukts verbessern, sondern unterstützt auch die unternehmensweite Verbesserung des Entwicklungsprozesses durch Bereitstellung kritischer Informationen.

3.9 Zusammenfassung

In diesem Kapitel haben Sie acht wichtige Vorteile der Continuous Integration kennengelernt:

- reduzierte Risiken
- verbesserte Produktqualität
- allzeit auslieferbare Produkte
- gesteigerte Effizienz
- dokumentierte Build-Prozesse
- höhere Motivation von Entwicklern
- besserer Informationsfluss
- Unterstützung von Prozessverbesserungen

Klingen diese »Acht Schätze« aber nicht zu schön, um wahr zu sein? Welche Voraussetzungen werden denn benötigt, um diese Vorteile zu erzielen? Im folgenden Kapitel werden wir dazu die empfohlenen Praktiken betrachten, auf denen ein effektives CI-System basiert.

4 Die CI-Praktiken

Continuous Integration basiert auf einer Reihe von empfohlenen Praktiken. Wir haben diese Praktiken nach [Fowler00] bereits kurz in Kapitel 2 kennengelernt. Jede dieser Praktiken kann – sogar für sich alleine genommen – den Softwareentwicklungsprozess deutlich verbessern. Ein wirklich effektives CI-System wird hingegen auf keine der im Folgenden ausführlich betrachteten Praktiken verzichten wollen.

4.1 Gemeinsame Codebasis

Eine gemeinsame Codebasis, typischerweise in einem Versionskontrollsystem verwaltet, ist sicherlich die offensichtlichste CI-Praktik. Wenn häufig integriert werden soll, muss es ja auch ein Ziel geben, wohin die Änderungen integriert werden. »Das ist noch nichts Neues – wir arbeiten doch schon lange mit einem Versionskontrollsystem!«, fragen sich sicherlich viele Leser an dieser Stelle. In der Tat ist die bloße Existenz eines solchen Systems eine notwendige, aber nicht hinreichende Bedingung.

Entscheidend ist auch, ob wirklich alle benötigten Daten versiert werden, die für einen Build bis hin zum auslieferbaren Produkt benötigt werden. Programmquelltexte stellen hierbei in den seltensten Fällen das Problem dar, denn deren Versionierung leuchtet jedem Entwickler unmittelbar ein. Anders sieht es bei benötigten Bibliotheken, Kommandozeilenwerkzeugen, Konfigurationsdaten, Datenbankschemas oder den »kleinen Skripten, die mal ein Diplomand geschrieben hat«, aus. Oftmals ist Entwicklern gar nicht bewusst, dass diese Dateien ebenso viel zum auslieferbaren Produkt beitragen wie Programmquelltexte. Sei es, weil Build-Skripte nicht immer von Entwicklern geschrieben werden oder aber, weil sie nicht im Installationspaket des Produktes ausgeliefert werden.

Aber wie viel sollte man ins Versionskontrollsystem einchecken? Auch Compiler, Entwicklungsumgebungen, Datenbankserver, Betriebs-

*Versionskontrolle?
Aber das machen wir
doch schon.*

*Quelltexte sind nur
ein Teil des Produkts*

*Wie viel sollte versioniert
werden?*

systeme, BIOS-Abbilder? Manche Teams in sehr streng kontrollierten Branchen, etwa dem Finanzsektor oder der Medizintechnik, gehen tatsächlich so weit. In den meisten Fällen muss man es nicht so weit treiben. Ein guter Test ist folgendes Gedankenexperiment: Welche Informationen müssten Sie einem neuen Kollegen geben, damit er auf einem frisch installierten Rechner mit der betriebsüblichen Softwareausstattung einen Build erstellen kann? Der Idealfall wäre lediglich ein Pfad ins Versionskontrollsystem, ohne jegliche weiteren Zusätze wie »... und dann musst du noch Datei X vom Netzlaufwerk ins Verzeichnis Y kopieren – aber warte mal, ich habe eine aktuellere Datei bei mir auf dem Rechner ...«. Große und relativ stabile Komponenten wie Betriebssystem, Compiler oder Datenbanksysteme können hingegen vorausgesetzt werden – was aber in Build-Skripten dokumentiert sein muss. Verwendet Ihr Build beispielsweise einen bestimmten Compiler, so muss dieser unter einer definierten Stelle lokal installiert sein oder dessen Wurzelverzeichnis über eine definierte Umgebungsvariable erreichbar sein. Zur Sicherheit sollte das Build-Skript zu Beginn des Builds die Existenz dieser Verzeichnisse überprüfen und gegebenenfalls mit aussagekräftigen Fehlermeldungen abbrechen.

Auch rechner- oder benutzerspezifische Daten können versioniert werden.

Auf der anderen Seite sollten Sie Versionierung nicht nur auf diejenigen Dateien beschränken, die für alle Entwickler relevant sind. In vielen Fällen hat es sich als hilfreich erwiesen, auch Dateien zu versieren, die lediglich für eine bestimmte Rechnerumgebung oder einen bestimmten Benutzer von Interesse sind, beispielsweise Einstellungen einer Entwicklungsumgebung. Ein eigens dafür reserviertes Plätzchen im Versionskontrollsystem kann sowohl als rettendes Backup als auch als Inspiration für neue Kollegen oder zum Vergleich nach einer Konfigurationsänderung dienen.

4.2 Automatisierter Build

Eine häufige Integration – typischerweise mehrmals am Tag – lässt sich in der Praxis nur durch Automatisierung des Build-Prozesses leisten. Automatisierte Builds sind darüber hinaus nicht nur schneller, sondern auch besser reproduzierbar, weniger anfällig für Flüchtigkeitsfehler und verbrauchen weniger kostbare menschliche Arbeitszeit.

Kompilieren ist nur Teil des Builds.

Im vorausgegangenen Abschnitt sind wir darauf eingegangen, dass die gemeinsame Codebasis nicht nur die Quelltexte, sondern alle Daten beinhalten muss, die für einen vollautomatischen Build erforderlich sind. Analog gilt für den Build-Prozess, dass er nicht nur das Kompilieren des Quellcodes und ggf. das Paketieren zu einer Distribution automatisieren sollte. Vielmehr müssen auch Schritte wie Quali-

tätssicherung, Verteilung und gegebenenfalls Installation des Softwareprodukts berücksichtigt werden.

In gewachsenen Strukturen kommt meist eine Vielzahl an Build-Werkzeugen und Technologien zum Einsatz: Shell-Skripte, Perl, Ant, Maven, ein SQL-Client, eine interaktive Applikation mit grafischer Benutzeroberfläche usw. Deren kompromisslose, vollständige Automatisierung ist in vielen Fällen nicht selten die größte Aufgabe, vor der ein Team bei der Einführung von CI steht. Von Vorteil ist es daher, Build-Werkzeuge einzusetzen, die auf allen benötigten Plattformen verfügbar sind, und den Build-Prozess vom darunterliegenden Betriebssystem ausreichend zu abstrahieren. Im Java-Umfeld haben sich hier Ant und Maven fest etabliert, aber auch verbreitete Skriptsprachen (etwa Ruby, Python oder Groovy) können mit Erfolg eingesetzt werden. Achten Sie bei der Wahl Ihres zentralen Build-Werkzeugs auf die Möglichkeit, beliebige externe Kommandos einbinden zu können. Über diesen Weg können Sie einzelne Abschnitte des Build-Prozesses durch Aufruf bestehender Skripte realisieren, auch wenn diese in unterschiedlichen Technologien realisiert wurden. In einem zweiten Schritt können Sie die externen Skripte mit den Mitteln des gewählten zentralen Build-Werkzeugs nachbilden und somit die Anzahl der verwendeten Technologien nach und nach reduzieren.

Build-Skripte sollten Abhängigkeiten zu speziellen Umgebungen in eigenständige Konfigurationsdateien auslagern, z.B. lokale Dateipfade, lokale Benutzerkonten und Passwörter, Namen von Datenbankschemas, URLs, IP-Adressen usw. Verwenden Sie automatisierte Inspektionen, um solche Schwachstellen systematisch zu finden. Das Analysewerkzeug PMD kann beispielsweise »hart codierte« IP-Adressen in Java-Quelltexten aufspüren. Ausgelagerte Konfigurationsdateien sorgen nicht nur für eine saubere Trennung zwischen dem überall identischen Ablauf des Build-Prozesses und den lokalen Besonderheiten in der Konfiguration. Sie erlauben auch, den Build auf einfache Weise auf weiteren Maschinen auszuführen.

Achten Sie darauf, dass Ihr Build-Prozess angemessene Fortschrittsmeldungen ausgibt: Zum einen sollten Sie alle logischen Schritte (Kompilieren, Testen, Inspizieren, Packen, Verteilen usw.) im Build-Protokoll nachvollziehen können. Zum anderen hat es sich als sehr hilfreich erwiesen, wenn ein Build-Prozess mindestens alle 15–30 Sekunden ein Lebenszeichen in Form einer Ausgabe ins Build-Protokoll von sich gibt. Dadurch entdecken Sie feststeckende Builds schneller. Im Idealfall können Sie die Ausführlichkeit des Protokolls von außen steuern. Sie vermeiden damit sowohl zu knappe Protokolle, die Ihnen zu wenige Ansatzpunkte bei der Fehlersuche bieten, als auch zu

Konsolidierung der Build-Technologien

Konfigurationen aus dem Build-Skript auslagern

Angemessene Fortschrittsmeldungen

geschwätzige Protokolle, die für jeden neuen Build Speicherplatz im zweistelligen Megabyte-Bereich belegen. Hilfreich ist ebenfalls die Möglichkeit, sich Zeitstempel ins Protokoll ausgeben zu lassen. Bei der Optimierung der Build-Dauer lassen sich so langlaufende Schritte besser identifizieren.

Ein Build-Skript für
CI-Server und
Entwicklerrechner

Das Kompilieren eines umfangreichen Produkts kann lange dauern. Daher beherrschen gute Entwicklungsumgebungen Optimierungsmöglichkeiten wie inkrementelles Kompilieren oder *hot deployment* von geändertem Code in laufende Prozesse. Hier ist prinzipiell alles erlaubt, was die Effizienz des Entwicklers steigert. Verbindlich sind aber immer nur diejenigen Build-Ergebnisse, die auf dem CI-Server erzeugt wurden. Build-Skripte sollten daher so geschrieben sein, dass sie sowohl auf dem CI-Server als auch lokal auf dem Rechner des Entwicklers ausführbar sind. Disziplinierte Entwickler vermeiden so vor jedem Einchecken neuer Änderungen mit privaten, lokalen Builds bereits im Vorfeld fehlgeschlagene Builds im CI-System.

4.3 Häufige Integration

Die häufige Integration, welche regelmäßiges Einchecken von Codeänderungen voraussetzt, ist vor allem eine kulturelle Veränderung. Für viele Entwickler geht damit ein Umdenken hinsichtlich der Größe einer »Arbeitseinheit« einher. Arbeitspakete, die bisher in mehreren Tagen oder Wochen isolierten Entwickelns umgesetzt wurden, sind für diesen Ansatz zu grobkörnig. Stattdessen müssen diese Pakete vom Entwickler, vielleicht auch unter Anleitung eines Senior-Entwicklers oder Trainers, in kleinere Einheiten zu wenigen Stunden Arbeitszeit zerlegt werden.

»One check-in a day
keeps tickets away.«

Als Faustregel sollte ein Entwickler mindestens einmal am Tag seine lokale Arbeitskopie auf den momentanen Stand der gemeinsamen Codebasis aktualisieren, dann mit seinen Änderungen testen und – bei erfolgreichem privaten Build – ins Versionskontrollsystem einchecken. Dies sorgt nicht nur für ein garantierter tägliches Erfolgsergebnis, sondern hält auch die Chance gering, dass voneinander abhängiger Code allzu weit auseinanderdriftet.

Umgekehrt werden die Änderungen eines Entwicklers erst durch das Einchecken für alle anderen Beteiligten sichtbar. Geschieht dies häufig, kann auf Fehlentwicklungen viel früher und mit überschaubaren Kurskorrekturen reagiert werden. Das ist besser, als später in einem Kraftakt »den Tanker drehen« zu müssen.

4.4 Selbsttestender Build

Wie in Abschnitt 4.2 bereits angesprochen, beinhaltet ein vollautomatischer Build die komplette Kette der Produkterstellung bis zum auslieferbaren Produkt. Dies schließt eine automatische Qualitätssicherung mit ein, die vor allem Tests beinhaltet, aber auch die Überprüfung weiterer zu erreichender Qualitätsziele. Ein solcher Build begutachtet also seine eigenen Erzeugnisse. Man spricht daher von einem »selbsttestenden Build«. Dabei wird auf mehreren Ebenen geprüft, auf die wir im Folgenden eingehen werden.

Qualitätssicherung ist mehr als Testen.

4.4.1 Compiler

Die erste Hürde stellt die erfolgreiche Kompilierung der Quelltexte dar – sofern eine kompilierbare Programmiersprache zum Einsatz kommt. Andernfalls wäre an dieser Stelle auch eine Syntaxüberprüfung der Quelltexte denkbar. Je nach Qualität der Entwicklungsumgebungen mag diese Hürde mehr oder weniger oft scheitern. Im Java-Bereich werden Syntax- und Kompilierungsprobleme in der Regel bereits während des interaktiven Codierens entdeckt.

Umgang mit Compiler-Warnungen

In Programmiersprachen mit geringerer Werkzeugunterstützung stellt die Kompilierung hingegen immer noch eine wichtige, erste Überprüfung dar. Gerade in diesen Umgebungen ist es wichtig, den Umgang mit Compiler-Warnungen zu klären. Im Gegensatz zu Compiler-Fehlern, die den Kompilationsvorgang zwingend abbrechen, besteht bei Warnungen die bequeme Möglichkeit, dem einfachen Weg zu folgen und einfach wegzusehen. Hier sind Seniorentwickler und Projektleiter gefragt: Ein erfolgreicher Kompilationsvorgang sollte komplett frei von Warnungen sein. Daher muss für jedes Projekt festgelegt werden, welche Warnungen zu beachten und zu beheben sind – und welche ignoriert werden dürfen (bitte vorher gründlich überlegen). Letztere sollten durch entsprechende Parametrisierung des Compilers deaktiviert oder aus den Ausgaben des Compilers gefiltert werden. Nur so gehen die verbleibenden Warnungen nicht im Strom der zahlreichen »unkritischen« Warnungen unter und werden mit der gebotenen Aufmerksamkeit behandelt werden.

4.4.2 Unit-Tests

Der erfolgreichen Kompilierung schließen sich die Unit-Tests an, welche – der Name deutet es an – möglichst kleine Bestandteile eines Produkts in Isolation überprüfen. In Java-Projekten werden hier typischerweise einzelne Klassen getestet. Da Unit-Tests kleine Einheiten

Unit-Tests: Die Sprinter

prüfen und die Umgebung dieser Einheiten während des Testens lediglich durch Test-Doubles nachgebildet wird (z.B. mit Dummies, Fakes, Stubs oder Mocks), können effiziente Unit-Tests im Sekundenbereich durchgeführt werden. Oftmals werden sie daher automatisch bei jedem Kompilierungsvorgang mit ausgeführt – im Gegensatz zu den aufwendigeren Tests auf Komponenten- oder Systemebene.

Code muss testbar entwickelt werden.

Die Entwicklung guter Unit-Tests ist ein Thema, das eigene Bücher füllt und auf das hier aus Platzgründen nicht weiter eingegangen werden kann. Wichtig ist jedoch, zu verstehen, dass sich nicht alle Quelltexte gleich gut mit Unit-Tests prüfen lassen. Vielmehr muss Code bereits gut testbar entwickelt werden. Was sich als Einschränkung anhört, wirkt sich in der Regel aber als großer Vorteil aus, vor allem hinsichtlich der Wartbarkeit. Umgekehrt bedeutet dies leider, dass sich bestehender Code in vielen Fällen nur sehr schwierig nachträglich mit Unit-Tests absichern lässt. Diese Projekte vertrauen dann auf eine Kombination aus wachsamem Compiler (siehe dazu die Hinweise zu Compiler-Warnungen im vorausgegangenen Abschnitt) und ausreichender Codeabdeckung durch Tests auf höheren Ebenen.

4.4.3 Komponententests

In Komponententests werden zusammengehörende Codebereiche, also etwa mehrere Java-Klassen, als Modul betrachtet und entlang der Modulschnittstellen zur Außenwelt getestet. Oftmals verlaufen auch die organisatorischen Grenzen entlang dieser Module, anders ausdrückt: Die Architektur eines Produkts spiegelt sich im Zuschnitt der Teams wider. Was dabei Ursache, was Wirkung ist, kann spekuliert werden. Aus Sicht der CI hat diese Abbildung von Codebereichen auf Teams aber den Vorteil, dass sich auftretende Problem zumindest auf Teamebene relativ klar zuordnen lassen. Somit lässt sich schnell klären, wer sich am kompetentesten um ein bestimmtes Problem kümmern kann.

4.4.4 Systemtests

Systemtests überprüfen ein komplettes Softwareprodukt. Dabei werden oft dieselben Schnittstellen verwendet, die auch im späteren Wirkbetrieb zum Einsatz kommen, etwa grafische Benutzeroberflächen.

Systemtests: Die Langläufer

Systemtests dauern in der Regel deutlich länger als Unit- oder Komponententests, da sie eine komplexere Testinfrastruktur benötigen. Für die Einbindung in einen CI-Build bedeutet dies, dass alle Möglichkeiten ausgeschöpft werden müssen, um Build-Zeiten kurz zu halten, etwa durch Parallelisierung auf mehrere Build-Server.

Systemtests werden außerdem meist von Anfang an als eigenständiger Schritt der Qualitätssicherung betrachtet und daher typischerweise nicht von Entwicklern, sondern von Testern erstellt und durchgeführt – ebenfalls im Gegensatz zu den Unit- oder Komponententests. Aus CI-Sicht bringt dies zum einen eine komplett neue Welt an Werkzeugen mit sich, die es anzubinden gilt. Zum anderen verleiht die einfache Durchführbarkeit eines Systemtests über die Benutzerschnittstelle dazu, diese Tests überhaupt nicht zu automatisieren, sondern manuell »vom Praktikanten durchklicken« zu lassen. Dabei wird gerne übersehen, dass der eigentlichen Testdurchführung der Aufbau der Testumgebung vorausgehen muss, also die Erstellung des zu testenden Produkts, die Installation auf ein Testsystem, das Einspielen eines Testszenarios in eine Datenbank usw. Für diese Aktivitäten muss dann jedes Mal ein Entwickler herangezogen oder der Tester entsprechend qualifiziert werden. Im Ergebnis aber werden in beiden Fällen identisch ablaufende, sich monoton wiederholende Tätigkeiten von qualifiziertem (= kostbarem) Personal erbracht, das an anderer Stelle wesentlich produktiver eingesetzt werden könnte.

Systemtests werden oft nicht automatisiert.

4.4.5 Inspektionen

Neben Tests zur Überprüfung funktionaler Anforderungen sollten auch Inspektionen eingesetzt werden. Dabei handelt es sich um Werkzeuge wie etwa statische Codeanalysen oder die Erhebung der Codeabdeckung (Anteil des Codes, der während der Tests durchlaufen wurde).

Entscheidend ist, für alle gemessenen Metriken Schwellwerte festzulegen, ab denen ein Build als fehlgeschlagen gilt und eine Reaktion (meistens durch Entwickler) gefordert wird. In der Praxis ist hierbei praktikabler, die Inspektionswerkzeuge etwas laxer einzustellen, dafür aber die Anzahl erlaubter Warnungen auf null festzulegen, als einen etwas beliebigen »20-Warnungen-sind-noch-in-Ordnung«-Ansatz zu verfolgen (warum nicht 15 oder 25?).

Nur »aktive« Kennzahlen messen

Vermeiden Sie außerdem die Erhebung umfangreicher Metriken »auf Vorrat«. Wirklich wichtige Kennzahlen laufen dabei Gefahr, in einer Flut an Daten unterzugehen. Nebenbei bemerkt, kann die Berechnung von Metriken den Build-Prozess erheblich verlangsamen. Messen Sie also nur »aktive« Kennzahlen, die auch Reaktionen einfordern können.

4.5 Builds (und Tests) nach jeder Änderung

Wie oft sollte gebaut werden?

Viele Arbeitsgruppen ohne CI bauen einmal pro Nacht (*nightly build*). Dieser regelmäßig getaktete Build stellt einen großen Fortschritt gegenüber einer späten Integrationsphase am Ende eines Projektes dar. Der Witz an der CI besteht jedoch an der schnellen Rückmeldung an den Entwickler hinsichtlich seiner jüngsten Codeänderungen. Bei einem Build pro Nacht kann dies im ungünstigsten Falle bis zu 24 Stunden zeitlichen Abstand zwischen Ursache (Codeänderung) und Wirkung (Rückmeldung durch CI-System) bedeuten. Für ein effektives CI-System ist das zu lang. Builds sollten daher durch Veränderungen im Versionskontrollsystem ausgelöst werden.

Sollte jede Revision gebaut werden?

Eine Frage, die in diesem Zusammenhang oft gestellt wird: Sollte idealerweise nach jeder Änderung neu gebaut werden? Also Revision für Revision? Im Prinzip ist das korrekt, da es kleinstmögliche »Deltas« zwischen den einzelnen Builds erzeugt und sich neu auftretende Probleme bestmöglich auf bestimmte Änderungen im Code eingrenzen lassen. In der Praxis kann dies jedoch bei hoher Änderungsfrequenz und langer Build-Zeit zu einem enormen Rückstau an Builds führen. Als Lösung hat es sich in vielen Teams als praktikabel erwiesen, einfach so oft wie möglich zu bauen, auch wenn dann die Änderungen mehrerer Entwickler gemeinsam in einem Build geprüft werden. Sind die Build-Zeiten hinreichend gering (unter 15 Minuten für Kompilation und einen schnellen Test), reicht dieses Vorgehen den meisten Arbeitsgruppen aus, um ausreichend Trennschärfe bei neu auftretenden Problemen zu erzielen. Alternativ kann auch ein schneller Build mit geringem Testumfang und hoher Frequenz mit einem langsameren Build mit vollständigem Testumfang und dafür reduzierter Frequenz kombiniert werden.

4.6 Schnelle Build-Zyklen

Lange Builds: Achillesferse des CI-Konzepts

Im vorausgegangenen Abschnitt wurde bereits auf die Notwendigkeit schneller Rückmeldungen hingewiesen. Lange Build-Zeiten sind in der Tat die Achillesferse des gesamten CI-Konzepts. Gerade Teams, die euphorisch ihre ersten Schritte in einem CI-gestützten Entwicklungsprozess unternehmen, neigen dazu, Builds mit allerlei Inspektionen und Testwerkzeugen zu spicken. In der Folge wuchert ein zunächst flinker Build mit 10 Minuten Dauer auf ein Vielfaches dieser Zeit.

Optimierung der Build-Zeit

Die Optimierung der Build-Zeit wird somit zur ständigen Aufgabe des Build-Managers. Neben offensichtlichen Maßnahmen wie der technischen Aufrüstung der Build-Server und dem Streichen überflüssigen Ballasts aus den Build-Skripten haben sich drei Ansätze als besonders wirksam erwiesen:

■ Staffeln des Builds (build staging, pipelining):

Der Build wird in mehrere Stufen zerlegt, etwa Kompilieren, Testen und Paketieren. Dadurch kann der Entwickler bereits nach Abschluss der ersten Stufe – hier: Kompilieren – eine erste Rückmeldung erhalten. Schlägt eine Stufe fehl, können zudem nachgelagerte Stufen meist entfallen und so Ressourcen auf dem Build-Server für andere Builds freigegeben werden.

■ Modularisierung:

Das Softwareprodukt wird in mehrere Module zerlegt, die weitestgehend unabhängig voneinander gebaut werden können. Finden Änderungen nur in einem Modul statt, können die Teilprodukte der anderen Module aus vorhergehenden Builds wiederverwendet werden und müssen nicht komplett neu gebaut werden. Diese Idee kennen Sie vermutlich bereits aus der inkrementellen Kompilierung von Quelltexten.

■ Parallelisierung:

Ist ein Build bereits modularisiert, können Module gegebenenfalls parallel gebaut werden. Findet dies auf mehreren Rechnern gleichzeitig statt, spricht man von verteilten Builds (*distributed builds*). Insbesondere beim Testen eines Produktes in unterschiedlichen Umgebungen können hier enorme Geschwindigkeitszuwächse erzielt werden.

4.7 Tests in gespiegelter Produktionsumgebung

In einer idealen Welt würde ein neuer Softwarestand in genau der Umgebung des späteren Produktionsbetriebs getestet. In der Praxis ist dies jedoch in den wenigsten Fällen möglich. Hier nur ein paar Beispiele:

- Die Duplikierung der Produktionsumgebung wäre zu teuer, weil riesige Datenmengen bewegt werden müssten oder teure Hardware mehrfach vorgehalten werden müsste.
- Die Duplikierung der Produktionsumgebung ist aus juristischen Gründen nicht möglich, weil sie sensible personenbezogene Daten enthält.
- Die Produktionsumgebung ist so heterogen, dass nicht alle möglichen Umgebungen getestet werden können. So kann eine Desktop-Anwendung nicht im Voraus in allen denkbaren Kombinationen aus Betriebssystem, Service Packs, Sprachanpassung, Virenscanner, Netzwerkanbindung, installierter Software, angeschlossener Hardware usw. getestet werden.

- Die **Produktionsumgebung existiert noch nicht** – etwa bei der Entwicklung einer eingebetteten Software für eine neuartige Hardware, die bisher nur als Simulator verfügbar ist.

Lassen Sie sich vom notwendigen Aufwand nicht abschrecken und versuchen Sie trotzdem, die **spätere Produktionsumgebung so detailliert wie möglich nachzubilden** – hinsichtlich der Hardware als auch des Datenvolumens und des Lastprofils: Wenn Sie beispielsweise nur auf einem Rechner mit einer CPU testen, werden Sie nie Fehler beobachten können, die sich aus der Nebenläufigkeit eines Multi-CPU-Systems ergeben. Wenn nötig, berechnen Sie die Reparaturkosten eines Fehlers, der es in den Wirkbetrieb schafft, und vergleichen Sie diese Summe mit der Investition in adäquate Testsysteme.

Virtualisierung und Cloud Computing

Doch woher sollen Sie die benötigten Testsysteme in ihrer Vielzahl an unterschiedlichen Konfigurationen nehmen? **Die Virtualisierung von Rechnern und die flexible Anmietung von Rechenzeit bei Cloud-Computing-Anbietern können hierbei entscheidend helfen**, auch mit überschaubaren Ressourcen zahlreiche Produktionsumgebungen zu simulieren. Durch deren einfache, hardwarelose Handhabung kommen virtualisierte Systeme zudem einer Automatisierung des kompletten Testaufbaus besonders entgegen. Sie lassen sich beispielsweise technisch sehr einfach und schnell auf einen definierten Ausgangszustand zurücksetzen.

4.8 Einfacher Zugriff auf Build-Ergebnisse

Produkte im Kontext ihrer Builds

»Wo finde ich denn den letzten Stand zum Testen?« – diese Frage kann mit der Einführung eines CI-Systems überflüssig werden. Das CI-System fungiert nicht nur als Nonstop-Qualitätssicherung. Es dient gleichzeitig auch als Archiv der Artefakte, also der durch den Build erzeugten Ergebnisse. Im Gegensatz zu einem FTP-Server oder einem Netzlaufwerk werden die **Artefakte in einem CI-System im Kontext ihres Builds aufbewahrt**. Ein Tester kann daher **nicht nur das neueste Installationsprogramm herunterladen** – er sieht auch gleichzeitig, wann dieses **erstellt wurde und welche letzten Änderungen in diesen Stand eingeflossen sind**. Die erspart unnötige Rückfragen und Zeitverluste (»Ob der Patch #180473 schon enthalten ist? Das weiß nur Peter – und der ist schon im Feierabend«).

Einfacherer Zugriff – schnellere Rückmeldung

Allgemein: Ein einfacher Zugriff auf Build-Ergebnisse erhöht die Chance, dass neue Produktstände schneller als bisher in die Hände von Personen gelangen, die Rückmeldung zur Qualität der Software geben können. Gleichzeitig werden Entwickler von den Arbeitsunterbre-

chungen befreit, in denen sie früher Versionen für Test, Vertrieb oder das Produktmanagement ad hoc zusammenstellen mussten.

4.9 Automatisierte Berichte

Eine der schönen Eigenschaften eines CI-Systems ist seine Nähe zum Build-Prozess: Jedes einzelne Bit vom Auschecken der Quelltexte bis zur Paketierung des Endprodukts obliegt der Kontrolle des CI-Systems. Im Zusammenspiel mit einer automatischen Visualisierung dieser Daten lassen sich so praktisch alle Aspekte des Erstellungsprozesses überwachen und auswerten. Für die Nutzung dieser Informationen hat sich ein dreifacher Ansatz bewährt:

■ *Aktive Benachrichtigung:*

Treten neue Probleme auf, sollte das CI-System möglichst schnell und trennscharf diejenigen Personen benachrichtigen, die zur Problembehebung in der Lage sind. Die Nachricht sollte dazu präzise Informationen beinhalten, mit denen sich potenzielle »Problemzonen« bereits eingrenzen lassen. Ansonsten sollte ein CI-System von sich aus so spärlich wie möglich Nachrichten verschicken (»Keine Nachrichten sind gute Nachrichten.«). Es besteht sonst die Gefahr, dass »Continuous Integration« eher als »Continuous Noise« wahrgenommen wird.

■ *Öffentliche Statusanzeige:*

An einem Ort mit hohem Verkehr (etwa auf dem Flur oder in der Kaffeeküche) zeigt ein Informationsradiator den Zustand des CI-Systems in vereinfachter Form an – in kleinen Arbeitsgruppen hat sich hier eine ampelartige Darstellung mit Lampen bewährt. Dadurch wird der Status des CI-Systems auf unaufdringliche Weise mehrfach am Tag von vielen Personen wahrgenommen.

■ *Detaillierte Informationen auf Abruf:*

Liegen Probleme vor oder sollen Fragen zu vergangenen Builds geklärt werden, kann ein CI-System mit einer gebrauchsfertigen Aufbereitung archivierter Informationen glänzen. In der Regel geschieht dies über die Anzeige in einem Webbrowser und kann daher auch von verteilten Teams eingesehen werden. Dabei können sich die Auswertungen nicht nur über einzelne Builds erstrecken, sondern auch die Entwicklung wichtiger Kennzahlen im Laufe eines Projekts aufzeigen oder mehrere Projekte untereinander vergleichen.

4.10 Automatisierte Ausbringung (Deployment)

Ein vollautomatischer Build muss nicht mit dem Paketieren eines Produktes enden. Ein CI-System kann auch den letzten Schritt zum Wirkbetrieb eines Produktes steuern, bei einer Webanwendung beispielsweise das Ausbringen in einen Server. Neben den inzwischen hinlänglich dargelegten Vorteilen der Automatisierung spielt beim Ausbringen auch eine akkurate Protokollierung aller Vorgänge eine Rolle (»Wer hat welche Version wann ausgebracht?«).

Rolle rückwärts mit CI

In besonders ausgereiften Umgebungen lassen sich neue Programmversionen nicht nur per CI-System ausbringen, sondern – als letzte Rettung – auch wieder zurücknehmen (*roll back, back out*), etwa durch die automatische Ausbringung einer Vorgängerversion.

Deployment der Anwendung

Beim Deployment ist zu unterscheiden zwischen der Anwendung und den zugehörigen Daten. Die Anwendung kann in der Regel einfach durch den Austausch des alten mit einem vollständigen, neuen Stand aktualisiert werden.

Migration der Anwendungsdaten

Die Anwendungsdaten hingegen müssen in der Regel kontrolliert auf ein neues Format oder Datenbankschema migriert werden. Da Anwendungsdaten im Gegensatz zu Quelltexten nicht im Voraus bekannt sein können, unterliegen sie auch keiner Versionierung. Wie kann also ein vorhandener Datenbestand, typischerweise in einer relationalen Datenbank vorliegend, in ein neues Datenbankschema überführt werden, das genau zur installierten Anwendungsversion passt?

Migrationsskripte

Eine einfach zu realisierende Lösung verwendet sogenannte Migrationsskripte: Zunächst wird das Datenbankschema um eine Tabelle erweitert, in der die aktuelle Schemaversion gespeichert wird. In unserem Beispiel soll diese Tabelle METAINFO heißen. Alle Veränderungen am Schema werden anschließend ausschließlich über Migrationsskripte ausgeführt. Diese Skripte enthalten alle notwendigen SQL-Anweisungen zur Aktualisierung des Schemas und erhöhen abschließend die Schemaversion in METAINFO um eins. Die Skripte werden entsprechend der zugehörigen Schemaversion benannt, z.B. 001.sql, 002.sql, 003.sql usw. Ist die höchste Schemaversion beispielsweise 9, so kann eine neue Datenbank mit aktuellem Schema durch die sequentielle Ausführung der Migrationsskripte 001.sql bis 009.sql aufgebaut werden. Auf eine bestehende Datenbank in Schemaversion 7 müssen zur Aktualisierung hingegen nur die Dateien 008.sql und 009.sql angewendet werden. Da die Migrationsskripte zusammen mit den anderen Quelltexten versioniert werden, kann das Initialisieren und Aktualisieren von Datenbanken durch das CI-System automatisiert erfolgen. Einen Wermutstropfen enthält diese Methode jedoch: Sobald man in

den Migrationsskripten fortgeschrittene SQL-Funktionsmerkmale einsetzt, die nur von einem bestimmten Hersteller angeboten werden, ist man auf diesen festgelegt. Manche Arbeitsgruppen verwenden beispielsweise in Komponententests leichtgewichtige Datenbanken wie etwa HSQLDB, in Systemtests hingegen das Fabrikat des Produktivsystems, etwa Oracle. Hier scheitert der beschriebene Ansatz, denn sobald in Migrationsskripten PL/SQL-Kommandos eingesetzt werden, können diese nur noch unter Oracle ausgeführt werden.

Einen sehr interessanten Ansatz verfolgt daher das Open-Source-Projekt LiquiBase (<http://www.liquibase.org>): In einer XML-Datei werden Modifikationen am Schema einer Datenbank in einzelnen Änderungssätzen (*change sets*) definiert, die dann auf eine Datenbank angewendet werden können. Dies kann aus der Kommandozeile, aber auch aus Build-Werkzeugen wie Ant oder Maven heraus geschehen, was LiquiBase zum idealen Mitspieler in automatisierten Builds macht. Durch die herstellerneutrale XML-Notation der Änderungssätze arbeitet LiquiBase mit zahlreichen gängigen Datenbanksystemen zusammen, unter anderem DB2, Oracle, MySQL oder Microsoft SQL-Server. Die XML-Datei mit den Änderungssätzen wird zusammen mit allen anderen Quelltexten im Versionskontrollsystem verwaltet und kann daher im Rahmen von Builds ausgeführt werden.

Weitere sehr lesenswerte Überlegungen zur systematischen Evolution von Datenbankschemas finden Sie in [Scott06].

LiquiBase

Datenbank-Refactoring

4.11 Zusammenfassung

In diesem Kapitel haben Sie zehn empfohlene Praktiken kennengelernt, auf denen effektive CI-Systeme basieren:

- gemeinsame Codebasis
- automatisierter Build
- häufige Integration
- selbsttestender Build
- Builds (und Tests) nach jeder Änderung
- schnelle Build-Zyklen
- Tests in gespiegelter Produktionsumgebung
- einfacher Zugriff auf Build-Ergebnisse
- automatisierte Berichte
- automatisierte Ausbringung (Deployment)

Manche dieser Praktiken setzen Sie sicherlich bereits ein, etwa die gemeinsame Codebasis innerhalb eines Versionskontrollsystems. Andere hingegen erfordern noch zusätzliche technische Infrastruktur

(z.B. gespiegelte Produktionsumgebungen) oder kulturelles Umdenken (z.B. häufige Integration). Betrachten Sie diese Veränderungen jedoch nicht ausschließlich als zusätzlichen Aufwand, sondern eher als gute Investition: Jede der beschriebenen Praktiken wird sich unterm Strich lohnen – die eine bereits kurzfristig, die andere eher langfristig.

Dieses Kapitel beschließt den ersten Teil des Buches, in dem wir uns produktunabhängig mit den Grundlagen, Vorteilen und Praktiken der CI vertraut gemacht haben. Im nächsten Kapitel beginnen wir, diese Konzepte mit dem CI-Server »Hudson« in die Tat umzusetzen.

5 Hudson im Überblick

- Wer steckt hinter Hudson?
- Wie funktioniert Hudson?
- Was sind die wichtigsten »Hudson-Highlights«?
- Welche anderen CI-Systeme gibt es?

Dieses Kapitel gibt Ihnen einen schnellen und kompakten Überblick über das Projekt Hudson. Es ist somit der ideale Startpunkt, wenn Sie wenig Zeit haben oder als Projektleiter nur das »große Bild« ohne tiefgehende technische Details benötigen.

Sie erfahren dabei zunächst, wie Hudson entstanden ist und welches Team dahintersteckt. Gerade bei Open-Source-Projekten ist dieses Wissen nicht unerheblich, denn jeder Werkzeugwechsel ist eine Investition, die sich auch dauerhaft lohnen soll. Anschließend lernen Sie Hudsons Kernkonzepte, seine Architektur sowie zehn ausgewählte Highlights kennen. Zum Schluss vergleichen wir Hudson mit Alternativen, kostenlosen wie kommerziellen.

5.1 Die Hudson-Story

Hudsons Gründungslegende beginnt 2006 mit Kohsuke Kawaguchi, einem Softwareentwickler, der damals bei Sun Microsystems in Kalifornien arbeitete. Kawaguchi charakterisiert sich selbst gerne als »faul und Informatiker«. Er begann deshalb in seiner Freizeit mit der Programmierung eines Werkzeugs, das ihm langweilige und sich wiederholende Aufgaben in der Softwareerstellung abnehmen sollte – gewissermaßen ein virtueller Butler für Entwickler. Die Namenswahl für den Assistenten fiel auf »Hudson«.

Kollegen meldeten rasch Interesse an so einem persönlichen Assistenten an, und Kawaguchi war selbstverständlich alles andere als faul: So veröffentlichte er Hudson im April 2007 in der Version 1.100 auf Suns Entwicklerplattform dev.java.net. Seitdem erscheint im Schnitt

April 2007: Version 1.100

einmal pro Woche eine neue Version, ganz im Sinne des Prinzips »*release early, release often*«. Die Versionsnummern werden dabei einfach hochgezählt. Dieses Buch bezieht sich auf Version 1.360 (August 2010).

April 2008:
Duke's Choice Award

Einen besonderen Schub erfuhr das Projekt im April 2008 durch den »Duke's Choice Award«, den Sun Microsystems auf seiner Leitkonferenz JavaOne in der Kategorie »Entwicklerwerkzeuge« vergab. Die Auszeichnung erhöhte nicht nur die Sichtbarkeit des Projektes in der Entwicklergemeinde. Sie führte auch dazu, dass Kawaguchi sich im Rahmen seiner Arbeitszeit nun voll um Hudson kümmern konnte. Im Umfeld der Übernahme von Sun Microsystems durch Oracle Anfang 2010 beschloss Kawaguchi, seine Aktivitäten rund um Hudson in einem eigenen Unternehmen weiterzuführen, und bietet nun kommerzielle Unterstützung und Anpassungen im Rahmen seines Unternehmens InfraDNA (<http://www.infradna.com>) an. Hudson selbst bleibt weiterhin ein freies Open-Source-Projekt.

April 2010: 200 Committer

Inzwischen arbeiten über 200 Committer an Hudson, davon ungefähr 15 am Anwendungskern. Der Rest entwickelt Plugins für die unterschiedlichsten Anwendungsfälle. Im Schnitt entstehen so 1–2 neue Plugins pro Woche. Dies entspricht Hudsons Design-Philosophie, einen möglichst kleinen und kompakten Kern mit CI-Grundfunktionen bereitzustellen und die Integration von Versionsmanagementsystemen, Build-Mechanismen und Drittsystemen spezialisierten Plugins zu überlassen. Somit kann jede Hudson-Instanz durch Installation passender Plugins auf die umgebende Infrastruktur abgestimmt werden, ohne überflüssigen, ungenutzten Funktionsballast mitzuschleppen.

Inzwischen hat Hudson weite Verbreitung gefunden, von kleinen Entwicklungsteams bis hin zu den großen Namen wie etwa eBay, Hewlett-Packard, SAP, Goldman Sachs, Yahoo, Xerox, Allianz und – wenig überraschend – Sun bzw. inzwischen Oracle selbst.

MIT-Lizenz

Hudson ist kostenlos, quelloffen und steht unter der äußerst liberalen MIT-Lizenz, die eine kommerzielle Nutzung ausdrücklich gestattet. So bietet beispielsweise Sun bzw. inzwischen Oracle seit Mitte 2009 für Unternehmensanwender Supportverträge für Hudson an. Dies unterstreicht dreierlei: den fortgeschrittenen Reifegrad des Produkts, die strategische Bedeutung Hudsons bei Sun sowie die Bedeutung, die Hudson bei Unternehmensanwendern inzwischen zukommt.

5.2 Architektur und Konzepte

In diesem Abschnitt erfahren Sie, mit welchen Systemen Hudson typischerweise »nach außen« kommuniziert (Systemlandschaft) und wie Hudson mit Blick »nach innen« funktioniert (Datenmodell). Abschlie-

ßend betrachten wir, über welche Benutzerschnittstellen Sie mit Hudson arbeiten können.

5.2.1 Systemlandschaft

Zunächst aber: »Was ist Hudson? Ganz konkret, aus technischer Sicht?«

Hudson ist ein CI-Server, der als Java-Webapplikation realisiert ist. Hudson muss also immer innerhalb eines Webcontainers wie Jetty, Tomcat, JBoss, WebSphere o.Ä. betrieben werden. Wer keinen großen Webcontainer installieren möchte: Hudson bringt einen »Mini«-Webcontainer mit (Winstone), der selbst für Produktionssysteme völlig ausreichend ist.

Ein CI-System umfasst aber neben dem eigentlichen CI-Server (hier: Hudson) zahlreiche weitere Systeme. Abbildung 5–1 zeigt exemplarisch eine solche Systemlandschaft, in deren Mittelpunkt der CI-Server steht.

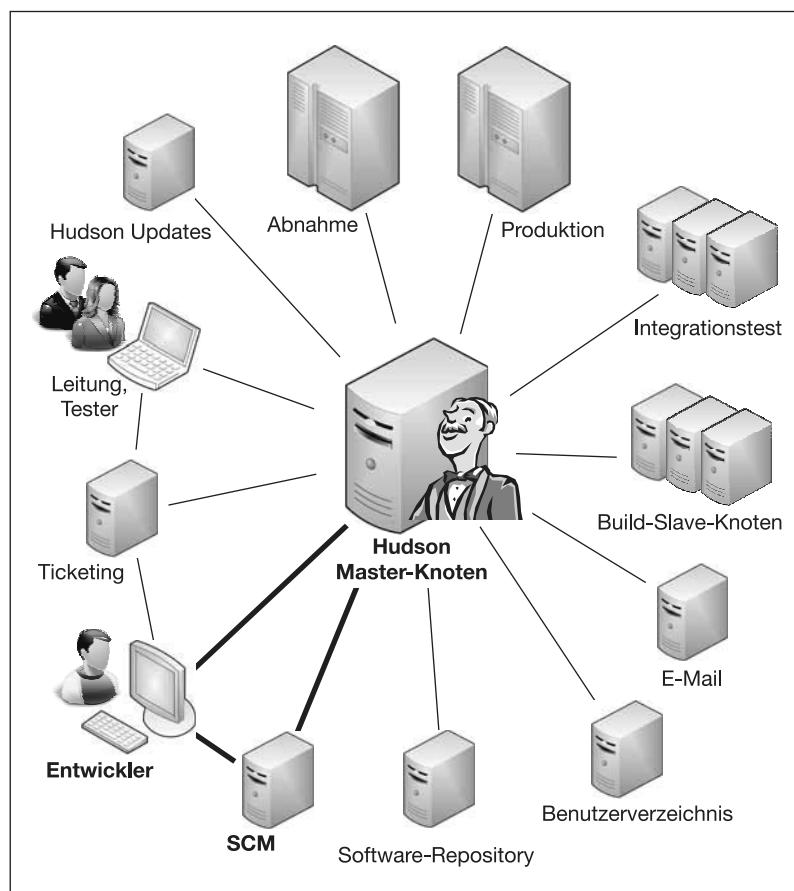


Abb. 5–1
Umgebung eines
CI-Systems

Ein minimales CI-System umfasst nur drei Komponenten (in Abb. 5–1 fett hervorgehoben):

- Einen *Entwicklerrechner*, auf dem neue Codeänderungen entwickelt und anschließend in ein Software-Configuration-Management-System (SCM-System) übertragen werden.
- Das *SCM-System* enthält alle Quelltexte, die notwendig sind, um ein Projekt zu bauen, und ordnet ihnen eindeutige Revisionen zu. Hudson verwendet den Begriff »SCM« synonym zu »Versionskontrolle«. Ein ausgewachsenes SCM-System kann aber deutlich mehr leisten als nur das Versionieren von Quelltexten und beispielsweise auch andere Artefakte eines Softwareprodukts kontrolliert verwalten. Dazu können etwa Anforderungen, Problembeschreibungen oder archivierte Endprodukte gehören. Hudson versteht unter dem Begriff »SCM« aber – wie eingangs erwähnt – nur den Aspekt der Versionierung. Um den Bezug zu Hudsons Benutzeroberfläche zu erleichtern, folgt dieses Buch dem abweichenden Sprachgebrauch. Wenn Sie also SCM lesen, denken Sie einfach an Subversion, CVS, Git, Perforce usw.
- Der *CI-Server*, also hier Hudson, holt sich vom SCM-System die kompletten Quelltexte eines Projekts ab und baut es ohne manuelles Zutun eines Benutzers. Die Ergebnisse sind über eine Weboberfläche abrufbar.

Ein CI-System kommt selten allein.

In der Praxis besteht ein CI-System jedoch aus weit mehr Komponenten. Abbildung 5–1 zeigt ein solches »Build-Ökosystem« mit seinen typischen Bestandteilen. In der Abbildung sind alle Dienste als eigenständige Server dargestellt. In vielen kleineren Teams werden diese Dienste jedoch auch auf wenigen Rechnern zusammengefasst.

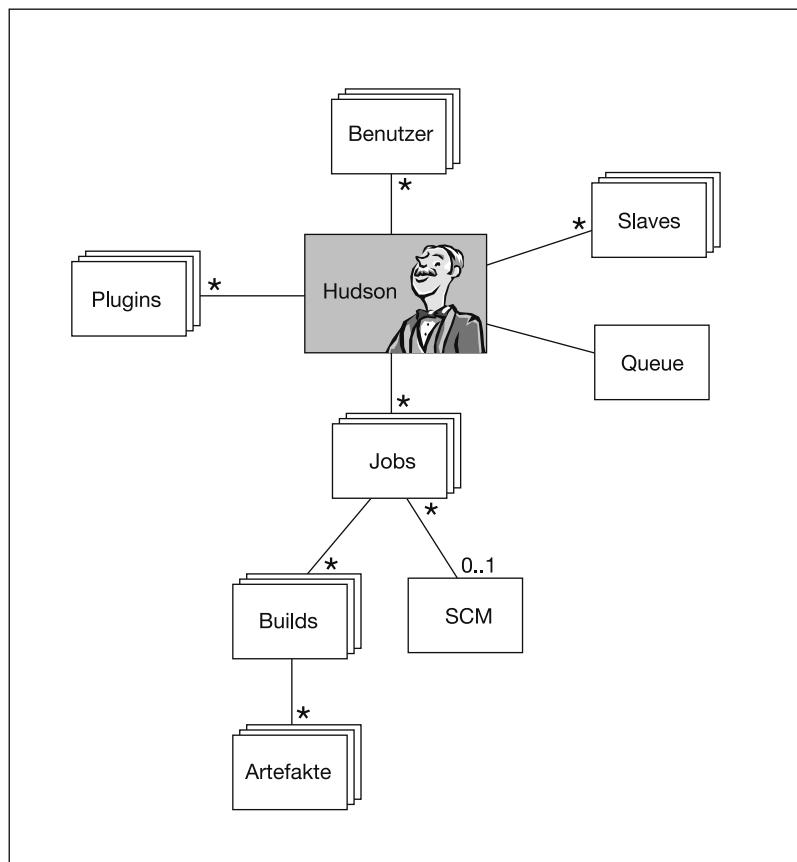
- Der *E-Mail-Server* versendet Benachrichtigungen an die Projektmitglieder, um über den Ausgang neuer Builds zu informieren. Trotz reger Entwicklung neuer Informationskanäle (Instant Messenger, Twitter, mobile Endgeräte) ist E-Mail immer noch in den meisten Teams der Standard für Benachrichtigungsaufgaben.
- Das *Benutzerverzeichnis* dient der Authentifizierung der Benutzer und unterstützt die Verwaltung von Zugriffsrechten.
- Das *Ticketing-System (issue tracker, bug tracker)* enthält Änderungswünsche (*feature requests*) und Problembeschreibungen (*bug reports*). Durch Verzahnung von SCM- und Ticketing-System mit dem CI-Server wird transparent, welche Arbeiten (Ticket) wie umgesetzt wurden (SCM-System) – und ob diese Änderungen erfolgreich waren (CI-Build).

- An den Arbeitsplätzen der *Projektleitung und der Testabteilung* kann der aktuelle Status eines Softwareprojekts eingesehen werden.
- Die *Testinfrastruktur* wird für Integrationstests benötigt, bei denen die spätere Einsatzumgebung so realistisch wie möglich simuliert wird. Dazu gehören etwa Datenbanken, Applikationsserver und spezialisierte Drittsysteme.
- Auf *Demo- und Abnahmesystemen (staging server, stager)* werden die gebauten Produkte ausgebracht, um beispielsweise neue Funktionen dem Management vorzustellen oder eine Kundenfreigabe einzuholen.
- Auf dem *Produktivsystem (production system, live server)* läuft ein Softwarereprodukt im Wirkbetrieb. Denkt man bei CI zunächst an die Phase der Produktentwicklung, so kann sie auch bei der Ausbringung (*deployment*) wertvolle Dienste leisten: Zum einen vermeidet die Vollautomatisierung Flüchtigkeitsfehler einer manuellen Installation. Zum anderen sorgt die Protokollierung aller Aktivitäten für eine lückenlose Nachvollziehbarkeit (»Wann haben wir das letzte Update auf dem Produktionssystem eingespielt? In welcher Version?«).
- *Build-Slave-Knoten (slave nodes)* erlauben es, den Build-Prozess auf mehrere Rechner zu verteilen. Dies ist besonders attraktiv, wenn viele Projekte gebaut werden müssen, die einzelnen Builds lange laufen oder das Produkt in heterogenen Umgebungen (z.B. auf unterschiedlichen Betriebssystemen) getestet werden muss.
- In *Software-Repositories* befinden sich Bibliotheken, die während des Build-Vorgangs in das Endprodukt integriert werden. Dabei kann es sich um Produkte von Drittherstellern handeln (z.B. Open-Source-Frameworks), aber auch um Eigenentwicklungen, die der besseren Handhabung halber als eigenständige Module verwaltet werden. Bekannte Vertreter dieser Gattung sind Maven-Repositories wie Nexus oder Artifactory. Ein CI-Server kann Software-Repositories nicht nur auslesen, sondern auch mit neuen Versionen füllen, die während eines Build-Laufs entstanden sind.
- Der öffentliche *Update-Server* des Hudson-Projekts hält Aktualisierungen für die Hudson-Webapplikation und zugehörige Plugins bereit und unterstützt bei der automatischen Installation von JDKs, Ant und Maven. Dies funktioniert allerdings nur, wenn Hudson einen Zugang zum Internet besitzt. Große Unternehmen setzen durchaus auch interne Update-Server auf, um die Auswahl an installierbaren Plugins besser zu kontrollieren und firmeneigene Plugins einfacher verteilen zu können.

5.2.2 Datenmodell

Hudsons Datenmodell stellt eine »Innensicht« auf den gesamten CI-Vorgang dar. Abbildung 5–2 zeigt die wichtigsten Objekte des Modells, die wir im Folgenden kurz erörtern.

Abb. 5–2
Hudsons Datenmodell



Hudson

Im Zentrum des Datenmodells steht das Hudson-Objekt, von dem es pro Hudson-Installation nur eine Instanz gibt. Das Hudson-Objekt erfüllt zweierlei Funktionen: Zum einen enthält es alle systemweiten Konfigurationen und Zustandsinformationen, beispielsweise Einstellungen zum E-Mail-Versand oder zur Absicherung des Systems. Zum anderen nimmt das Hudson-Objekt als Master-Knoten (*master node*) an der Ausführung von Builds teil.

Jobs

Wichtigste Aufgabe einer Hudson-Installation ist die Ausführung von Jobs. Hudson unterscheidet zwei Kategorien von Jobs: Projekte und externe Jobs.

Projekte bilden den typischen Anwendungsfall eines CI-Systems ab, also Auschecken von Quelltexten, Bauen und Testen der Artefakte, Archivieren der Ergebnisse und Versenden von Benachrichtigungen. Die weit überwiegende Mehrheit aller Jobs dürfte vom Typ »Projekt« sein. Um ausgewählte Szenarien noch besser zu unterstützen, bietet Hudson momentan Projekte in drei Ausprägungen an:

Projekte

- Ein *Free-Style-Projekt* bietet die meisten Freiheitsgrade. Dem Auschecken der Quelltexte schließen sich ein oder mehrere Build-Schritte an, z.B. mit Ant, Maven, Batchdateien oder Shell-Skripten. Abschließend werden die Build-Ergebnisse ausgewertet, gegebenenfalls archiviert und Benachrichtigungen verschickt.
- Ein *Maven-2-Projekt* ist da schon sehr viel mehr auf Mavens besondere Art und Weise spezialisiert, Software zu bauen. Durch Auswertung der Informationen in POM-Dateien erspart man sich einige manuelle Angaben in der Projektkonfiguration und erschließt sich zusätzliche Funktionalität wie automatisches Abhängigkeitsmanagement zwischen weiteren in Hudson angelegten Maven-2-Projekten.
- Ein *Multikonfigurationsprojekt* (auch als Matrix-Build bezeichnet) erlaubt die mehrfache Durchführung eines Builds in unterschiedlichen Konfigurationen. Dies könnten beispielsweise unterschiedliche JDK-Versionen, Datenbanken oder Applikationsserver sein. Hudson baut dabei alle spezifizierten Konfigurationen und stellt die Ergebnisse in zusammengefasster, tabellarischer Form dar.

Externe Jobs dienen der Überwachung von Abläufen, die nicht innerhalb Hudsons stattfinden, sondern – wie es der Name ja andeutet – extern. Bei externen Jobs fungiert Hudson also nicht als aktiver Ausführer eines Jobs, sondern eher als passiver Buchhalter, der den Ausgang eines anderen Prozesses zur Archivierung und Visualisierung mitgeteilt bekommt.

Externe Jobs

Builds

Der Begriff »Build« ist – je nach Kontext – unterschiedlich belegt, so dass wir die Verwendung in Zusammenhang mit Hudsons Datenmodell festlegen müssen:

- Zum einen kann darunter der Build-*Prozess* verstanden werden, der in einer langen Kette Schritt für Schritt des Build-Vorgangs ausführt: Auschecken, Kompilieren, Testen, Dokumentation erstellen, Archivieren, Verteilen, Benachrichtigen usw. Diese Bedeutung ist im folgenden Text *nicht* gemeint.
- Zum anderen kann ein »Build« auch das Ergebnis einer Ausführung eines Jobs sein. In dieser Bedeutung wird der Begriff in Hudsons Datenmodell und auch im folgenden Text verwendet.

Bei jedem Lauf eines Jobs entsteht also ein neuer, weiterer Build. Hudson nummeriert diese Builds pro Job streng monoton steigend durch, so dass beispielsweise Build »foobar #5« den fünften Lauf des Jobs »foobar« darstellt und es innerhalb einer Hudson-Installation nur genau einen Build mit dieser Bezeichnung geben kann.

Builds können auf unterschiedliche Weise ausgelöst werden: manuell, nach Zeitplan (z.B. einmal pro Nacht), bei Veränderungen im SCM oder durch Abhängigkeiten zu anderen Projekten.

Der Ablauf eines Build-Vorgangs erfolgt nach einem festen Schema. Zunächst werden die Quelltexte aus dem SCM abgerufen. Dann erfolgt eine Reihe von Build-Schritten, typischerweise Ant-, Maven- oder Skript-Aufrufe, welche die Quelltexte in Artefakte umwandeln. In einer letzten Phase werden abschließende Aktionen (*post-build actions*) ausgeführt, etwa das Verschicken von E-Mail-Benachrichtigungen, Auswerten von Testberichten, Archivieren von Dateien und ggf. Auslösen weiterer Builds.

Da Hudson die Ergebnisse jedes Builds archivieren kann, sind später nicht nur die Daten eines einzelnen Builds abrufbar (»Wie lange dauerte eigentlich Build foobar #5?«), sondern auch Trends über den Verlauf aller Builds (»Werden unsere Builds wirklich langsamer – oder wir nur ungeduldiger?«).

Artefakte

Das Ergebnis eines Builds wird als Artefakt bezeichnet. Dabei handelt es sich immer um Dateien, beispielsweise das Installationsprogramm einer Software, eine Webapplikation oder eine JAR-Datei, die als Bibliothek in andere Projekte eingebunden werden soll. Im Gegensatz zu allen anderen Dateien, die im Verlauf eines Builds entstehen und beim nächsten Build wieder gelöscht oder überschrieben werden können, werden Artefakte dauerhaft archiviert.

Ein Artefakt besteht technisch gesehen immer aus genau einer Datei. Ein Build kann jedoch mehrere Artefakte erzeugen. Für Projekte vom Typ »Free Style« gibt man Hudson mithilfe eines regulären Aus-

drucks in der Notation eines Ant-FileSets die Namen der Dateien an, die archiviert werden sollen. Bei Projekten vom Typ »Maven« ist durch die POM-Datei ja bereits beschrieben, welches Endergebnis der Maven-Build liefern soll. Hier müssen Sie nicht explizit Ihr Artefakt konfigurieren – Hudson findet dies durch Auswertung der POM-Datei selbst heraus.

SCM

Kontinuierliche Integration setzt voraus, dass alle Quelltexte eines Builds aus einem Versionsmanagementsystem bezogen werden. Hudson verwendet hierzu synonym den Begriff SCM (*Software Configuration Management*). Sie können pro Projekt ein oder mehrere SCM-Quellen angeben. Wird in einer dieser Quellen eine Änderung festgestellt, wird aus allen SCMs der aktuelle Stand bezogen und das Projekt gebaut.

Benutzer

Hudson verwaltet eine Liste aller ihm bekannten Personen. Wie baut Hudson diese Liste auf? Personen finden auf zwei unterschiedlichen Wegen Zugang:

- Personen werden in der Benutzerverwaltung explizit angelegt. Dieser Personenkreis benötigt in der Regel interaktiven Zugriff auf die Oberfläche von Hudson.
- Personen tragen zu einer SCM-Revision als Committer bei und werden von Hudson während des Builds implizit angelegt. Im Laufe eines Projektes lernt Hudson also alle Committer automatisch durch Auswertung der SCM-Revisionen kennen. Dieser Personenkreis muss nicht zwingenderweise Zugriff auf die Hudson-Oberfläche benötigen.

Neben Namen und Beschreibungstext können für Personen weitere Angaben hinterlegt werden, z.B. deren E-Mail-Adresse.

Plugins

Eine der größten Stärken Hudsons ist seine Plugin-Architektur. Hudson wird mit einer Handvoll Plugins ausgeliefert, die vor allem die Subversion-Anbindung und die Maven-Unterstützung beinhalten. In den weitaus meisten Fällen wird man über den eingebauten Plugin-Manager aus dem öffentlichen Plugin-Verzeichnis des Hudson-Projekts individuell ausgewählte Plugins hinzustallieren.

Hudson verwendet übrigens kein bekanntes Plugin-Framework mit Isolationsmöglichkeiten der Plugins untereinander wie etwa OSGi. Die theoretisch vorstellbaren Konflikte bereiten in der Praxis glücklicherweise (noch) kein allzu großes Kopfzerbrechen. Alle Plugins tragen eine Versionsnummer und sind mit Metadaten ausgestattet, die angeben, für welche Hudson-Version das Plugin entwickelt wurde.

Queue

Alle Build-Aufträge werden zunächst in einer Warteschlange (*build queue*) aufgenommen und deren Ausführung geplant. Hudson verteilt diese Build-Aufträge dann an freie Build-Prozessoren (*builders*) aller angeschlossener Knoten. Ein Job kann nicht mehrfach in der Queue geplant werden. Maximal kann ein Job momentan gebaut werden und dessen nächste Ausführung bereits in der Queue geplant sein. Dies vermeidet lange »Rückstaus«, die entstehen würden, wenn für ein Projekt mit vielen Änderungen in kurzer Zeit immer weitere neue Jobs geplant würden.

Slaves

In verteilten Builds werden Build-Aufträge nicht nur auf dem Master-Knoten ausgeführt, sondern auch an sogenannte Slave-Knoten (*slave nodes*) delegiert. Welche Vorteile bringt diese Verteilung? Die wichtigsten drei Motive aus der Praxis:

■ *Parallelisierung der Gesamtlast des CI-Systems.*

Durch das Verteilen der Jobs auf mehrere Rechner soll die Last parallelisiert und somit schneller abgearbeitet werden. Dies funktioniert am besten mit vielen, aber tendenziell kurzen Builds, da die Parallelisierung auf Job-Ebene stattfindet. Ein einzelner, stundenlanger »Bandwurm«-Build kann also nicht von Hudson auf magische Weise zerlegt und auf mehrere Knoten verteilt werden.

■ *Unterstützung unterschiedlicher Plattformen.*

Es soll auf mehr als einer Plattform gebaut und getestet werden. Slave-Knoten müssen nicht identisch in Hard- und Software sein. Sie können für jeden Job spezifizieren, welche Umgebung zur Ausführung benötigt wird. Hudson delegiert den Job dann an einen passenden Knoten.

■ *Entlastung des Master-Knotens.*

In manchen Hudson-Installationen werden Builds sogar ausschließlich auf Slave-Knoten gebaut, um auf dem Master-Knoten mehr Ressourcen für andere Aufgaben zur Verfügung zu haben.

Auf diese Weise bleibt beispielsweise die webbasierte Benutzeroberfläche auch während äußerst rechenintensiver Builds immer flüssig bedienbar und wird nicht von parallelen Prozessen »an die Wand gedrückt«.

5.2.3 Benutzerschnittstellen

Nachdem Sie das Umfeld eines Hudson-Servers und dessen Innenleben als Datenmodell kennengelernt haben, betrachten wir abschließend, über welche Schnittstellen Sie mit Hudson in Kontakt treten können. Dazu stehen Ihnen gleich drei Möglichkeiten zur Auswahl (Abb. 5–3):

- die webbasierte grafische Oberfläche (GUI)
- die REST-ähnliche Anwendungsschnittstelle (API)
- die Kommandozeilenanwendung (CLI)

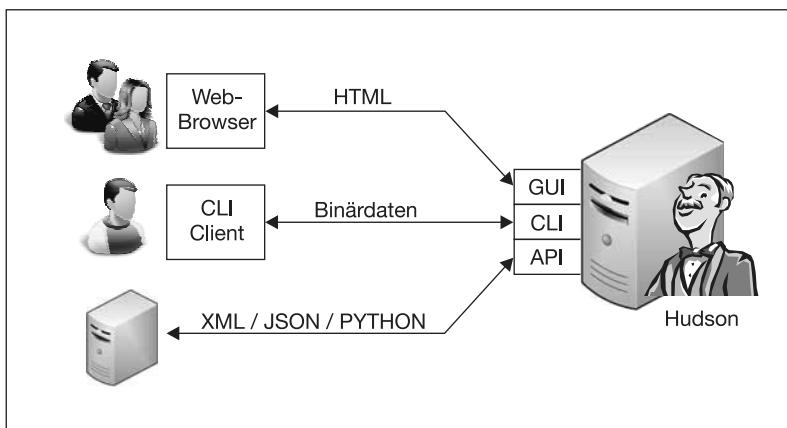


Abb. 5–3
Benutzerschnittstellen

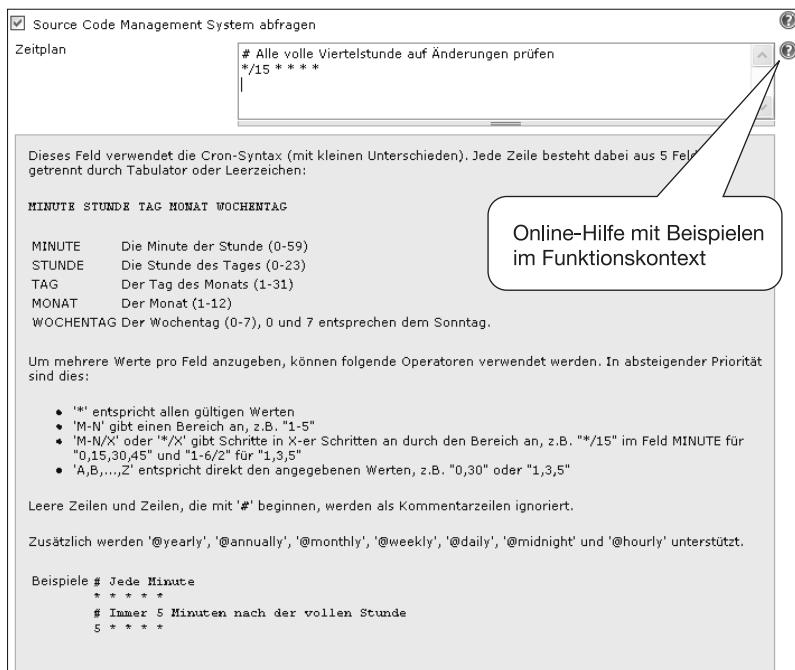
Die webbasierte grafische Oberfläche (GUI)

Die meisten Anwender kommunizieren mit Hudson ausschließlich über die webbasierte Oberfläche. Mit Kenntnis der Konzepte aus dem vorausgegangenen Abschnitt kann man sich schon sehr viel bei einem Rundgang durch die Oberfläche erschließen. Folgende »Bonbons« sollten Sie jedoch keinesfalls verpassen:

Zu den meisten Konfigurationsmöglichkeiten ist eine umfangreiche Online-Hilfe hinterlegt. Nutzen Sie sie! Der Umfang dieser Hilfen übersteigt die im Hudson-Wiki verfügbare Dokumentation bei Weitem. In der Hilfe finden Sie zudem in vielen Fällen gebrauchsfertige Beispiele, die Sie herauskopieren und in Ihre Einstellungen einfügen können.

Online-Hilfe

Abb. 5-4
Beispiel einer
ausführlichen Online-Hilfe



Sprechende URLs

Der Aufbau der URLs der Weboberfläche erfolgt einer klaren Systematik, die Hudsons Datenmodell widerspiegelt. So lassen sich auf einfache Weise »Einsprung-URLs« bilden, die Sie etwa aus einem Wiki oder einem Ticketing-System direkt an die gewünschte Stelle innerhalb der Hudson-Oberfläche führen. Zusätzlich existieren nützliche Permalinks, also konstante URLs, die zu besonders interessanten Stellen führen, etwa `/job/foobar/lastSuccessfulBuild` zum letzten erfolgreichen Build des Projektes foobar. Tabelle 5-1 zeigt weitere Beispiele dieser Systematik.

Tab. 5-1

Beispiele für Hudsons
sprechende URLs

URL	Bedeutung
/	Übersicht (Dashboard) anzeigen
/jobs/foobar	Übersicht zu Projekt »foobar« anzeigen
/job/foobar/5	Build Nr. 5 des Projekts »foobar« anzeigen
/job/foobar/5/testReport	Testergebnisse aus Build Nr. 5 des Projekts »foobar« anzeigen
/job/foobar/lastBuild	Letzten Build des Projekts »foobar« anzeigen
/job/foobar/build?delay=0sec	Neuen Build des Projekts »foobar« ohne Verzögerung sofort starten
/user/swiest	Benutzer »swiest« anzeigen

URL	Bedeutung
/pluginManager/installated	Liste aller installierten Plugins anzeigen
/view/MeineProjekte	Listenansicht »MeineProjekte« anzeigen

Meist übersehen: In der oberen rechten Ecke befindet sich eine Volltextsuche, mit der Sie sich einige Klicks ersparen können. Beispiel: Sie haben ein Projekt »foobar« angelegt. Eine Suche nach »foo 2 co« führt Sie dann – durch automatische Vervollständigung – direkt zur Konsoleausgabe des Builds Nr. 5 des Projekts »foobar«.

Volltextsuche



Abb. 5–5
Schnelle Navigation über
Volltextsuche

Zu vielen Objekten (z.B. Knoten, Jobs, Builds, Personen) können Sie Beschreibungen hinterlegen, die dann in der Oberfläche angezeigt werden. Dieser Beschreibungstext kann HTML-Auszeichnungen enthalten. Damit lassen sich nicht nur Formatierungen einfügen, sondern auch Tabellen aufbauen und Verknüpfungen zu externen Systemen herstellen. Abbildung 5–2 enthält Anregungen, was Sie in den Beschreibungen hinterlegen könnten.

Ressource	Beispiele für Beschreibungen
Knoten	Maßgebliche Hardwarekomponenten des Knotens, installierte Software, Kontaktinformationen des Administrators, regelmäßige Wartungszeiten
Jobs	URLs auf Staging- und Produktivsysteme oder korrespondierende Übersichten im Ticketing-System
Builds	SCM-Revisionsnummer, Modulversion (Maven)
Personen	URL auf Blog

Tab. 5–2
Einsatzbeispiele für
Beschreibungen

Die REST-ähnliche Anwendungsschnittstelle (API)

Mit der Anwendungsschnittstelle (API) können Sie Hudson fernsteuern – zum Beispiel aus einem Skript heraus. Im Prinzip rufen Sie per HTTP genau dieselben URLs auf, die Sie interaktiv in der GUI anklicken würden. Die URLs sind REST¹-ähnlich nach folgendem Schema aufgebaut:

1. Representational State Transfer (REST) ist ein Softwarearchitekturstil, bei dem jede Ressource einer Anwendung durch eine eigene Adresse, z.B. eine URL, angesprochen werden kann [Fielding 2000]. Auf jede dieser Ressourcen können dann bestimmte Operationen angewendet werden, etwa Anlegen (PUT), Verändern (POST), Abrufen (GET) oder Löschen (DELETE).

<http://hudson/pfad/zur/ressource/aktion?param1=wert1¶m2=wert2...>

Beispiel: Sie möchten einen Build des Jobs »foobar« starten. Dazu rufen Sie mit dem Linux-Kommando wget folgende URL auf:

```
wget http://hudson/jobs/foobar/build
```

Benötigt Ihr Aufruf Parameter oder erwarten Sie strukturierte Daten in der Rückantwort, so bietet Hudson eine praktische, eingebaute Dokumentation aller möglichen API-Funktionen an: Dazu navigieren Sie in der GUI auf die Seite der Ressource, die Sie fernsteuern möchten, zum Beispiel die Übersichtsseite eines Projekts. Dann hängen Sie an die URL ein /api an und bekommen die unterstützten API-Aufrufe angezeigt:

```
wget http://hudson/jobs/foobar/api
```

Rückgabeformate

Je nach gewünschtem Rückgabeformat hängen Sie zusätzlich /xml, /json oder /python an ihren API-Aufruf. Hudson liefert Ihnen dann die Ergebnisse in diesen »maschinenlesbaren« Formaten. Abbildung 5–6 zeigt Ihnen links die Übersichtsseite eines Hudson-Systems (<http://hudson/>) und rechts daneben die entsprechende XML-Version der REST-Schnittstelle (<http://hudson/api/xml>).

Abb. 5–6

Darstellung der Startseite als XML

The figure consists of two side-by-side screenshots. On the left, the 'HTML-Darstellung' (HTML representation) shows the Hudson dashboard with various job status cards and a search bar. On the right, the 'XML-Darstellung der API' (XML representation of the API) shows the raw XML code generated by the Hudson server. A large curved arrow points from the top of the Hudson interface towards the XML code, indicating the mapping between the user interface and the underlying API.

```

<hudson>
  <assignedLabel/>
  <mode>NORMAL</mode>
  <nodeDescription>Hudson Master-Knoten</nodeDescription>
  <nodeName />
  <numExecutors>2</numExecutors>
  <description><h1>Willkommen auf NG-LUNA</h1></description>
  - <job>
    <name>foobar</name>
    <url>http://localhost:8080/job/foobar/</url>
    <color>blue</color>
  </job>
  - <job>
    <name>greetr</name>
    <url>http://localhost:8080/job/greetr/</url>
    <color>blue</color>
  </job>
  <overallLoad />
  - <primaryView>
    <name>Alle</name>
    <url>http://localhost:8080/</url>
    <slaveAgentPort>0</slaveAgentPort>
    <useCrumbbs>false</useCrumbbs>
    <useSecurity>true</useSecurity>
  - <view>
    <name>Alle</name>
    <url>http://localhost:8080/</url>
  </view>
</hudson>

```

Für die wichtigsten Ansichten stehen API-Funktionen zur Verfügung. Bei Plugins ist dies bisher leider die Ausnahme. Benötigen Sie Daten aus einem Hudson-System, die nicht über die REST-Schnittstelle abgefragt werden, könnte Ihnen die Kommandozeilenanwendung weiterhelfen.

Die Kommandozeilenanwendung (CLI)

Als dritte Benutzerschnittstelle bietet Hudson eine Kommandozeilenanwendung an, das *command line interface*, kurz: CLI. Das CLI schließt eine Lücke zwischen Weboberfläche und REST-API, da es besser automatisierbar ist als die Weboberfläche, sich gleichzeitig aber besser interaktiv nutzen lässt als die REST-API.

Woher erhalten Sie die passende CLI-Anwendung für Ihre Hudson-Instanz? Ganz einfach: Hudson bringt sie bereits mit. Unter der URL `http://hudson/cli` können Sie das CLI in Form eines Java-Archivs herunterladen.

Der Aufruf, der einen Build des Projektes »foobar« startet, muss lediglich den anzusprechenden Hudson-Server, das Kommando `build` und den Projektnamen enthalten:

```
java -jar hudson-cli.jar -s http://hudson build foobar
```

Eine Übersicht aller verfügbaren CLI-Kommandos erhalten Sie mit:

```
java -jar hudson-cli.jar -s http://hudson help
```

Beachten Sie, dass Sie auch für das Kommando `help` die Server-URL Ihres Hudson-Servers angeben müssen. Die verfügbaren Kommandos sind nämlich nicht im CLI implementiert, sondern hängen vom Server ab, den Sie ansprechen. Aktuelle Hudson-Versionen bieten über 30 Kommandos an, etwa zum Ein- oder Ausschalten eines Wartungsfesters, zur Build-Queue-Steuerung und zur Fernkonfiguration.

Ein wahres »Schweizer Taschenmesser« für Administratoren stellt die Möglichkeit dar, Groovy-Kommandos per CLI auf dem Hudson-Server ausführen zu lassen. Beispiel: Sie möchten per Skript abfragen, welche Plugins in welcher Version auf Ihrer Hudson-Instanz installiert wurden. Dies können Sie folgendermaßen realisieren:

Groovy über das CLI

```
$ java -jar hudson-cli.jar -s http://localhost:8080 groovysh =>
  'hudson.model.Hudson.instance.pluginManager.plugins.each { =>
    println("${it.longName} - ${it.version}") }';
Static Analysis Utilities - 1.3
Hudson batch task plugin - 1.13
Checkstyle Plug-in - 3.2
Hudson CVS Plug-in - 1.0
Maven Integration plugin - 1.345
Hudson Support Subscription Notification Plugin - 1.2
Hudson SSH Slaves plugin - 0.9
Hudson Subversion Plug-in - 1.8
```

5.3 Die Top-10-Highlights

In diesem Abschnitt lernen Sie zehn ausgewählte »Hudson-Highlights« kennen: von der Installation über die täglichen Handgriffe bis hin zu Erweiterungsmöglichkeiten durch Plugins. Eine Auswahl an Highlights hat zugegebenermaßen immer etwas Subjektives, erfolgte hier aber aus der Perspektive eines Softwareentwicklers bzw. IT-Projektleiters.

5.3.1 Schnelle Installation

Wenn Sie bereits andere Serveranwendungen installiert und getestet haben, wissen Sie, dass dies mitunter eine abendfüllende Veranstaltung werden kann: Zunächst muss das passende Installationspaket für den eigenen Rechner gefunden werden. Im Anschluss sind dann Abhängigkeiten und Vorbedingungen zu erfüllen (leider auch die undokumentierten). Und speziell unter Windows startet man nicht selten ein Installationsprogramm, das ungefragt Dateien und Registry-Einträge auf der Festplatte verteilt...

*»Hudson komplett«
in einer WAR-Datei*

Hudson ist in dieser Hinsicht erfrischend einfach gestrickt: Es gibt nur eine Datei zum Herunterladen (unter der konstanten URL <http://hudson-ci.org/latest/hudson.war>), in der die komplette Software enthalten ist. Diese kann in einen Servlet-Container (z.B. Jetty, Tomcat, JBoss) ausgebracht werden oder wird ganz einfach direkt in der Kommandozeile gestartet. Ein spezielles Installationsprogramm ist also nicht notwendig. Darüber hinaus legt Hudson alle seine Daten kompakt unterhalb eines einzigen Datenverzeichnisses ab, dem sog. <HUDSON_HOME>. Möchte man sich von Hudson wieder trennen, löscht man dieses Verzeichnis und hat ein rückstandsfrei gesäubertes System. Die Datenhaltung in Dateien macht außerdem die Installation oder Anbindung eines Datenbanksystems überflüssig.

5.3.2 Effiziente Konfiguration

Der Installation schließt sich die Konfiguration an. Je nach Philosophie des Softwareherstellers geschieht dies typischerweise entweder durch Anpassung von Textdateien oder aber über eine grafische Benutzeroberfläche. Beide Ansätze haben ihre Stärken und Schwächen: Textbasierte Konfigurationen lassen sich einfach sichern, versionieren und automatisch erstellen, erfordern aber Lernaufwand oder – Himmel bewahre! – einen Blick ins Handbuch. Gute grafische Benutzeroberflächen sind hingegen intuitiver zu verstehen. Es ist aber unklar, wie und wo die Konfigurationsdaten intern gespeichert werden.

Hudsons Philosophie verbindet beide Ansätze elegant, indem alle Einstellungen über die Browseroberfläche veränderbar sind, hingegen serverseitig als XML-Dateien abgespeichert werden. Somit kann der Administrator je nach vorliegender Aufgabe und persönlichen Vorlieben entscheiden, ob er seine Arbeit schneller im Browser oder im Texteditor erledigen kann.

Grafische Konfiguration
im Frontend, XML im
Backend

In der Praxis verwenden die meisten Administratoren das Webinterface – selbst hartgesottene Kommandozeilen-Liebhaber. Sie schätzen gleichzeitig aber die einfache Sicherungs- und Wiederherstellungsmöglichkeit der XML-Konfigurationsdateien.

5.3.3 Unterstützung zahlreicher Build-Werkzeuge

Bei aller Standardisierung: Jedes Build-System ist ein kleiner Mikrokosmos, der individuell an die Infrastruktur einer Arbeitsgruppe angepasst ist. Hudson ist es weitestgehend egal, wie Sie Ihre Software bauen. Selbstverständlich werden die wichtigsten Werkzeuge aus der Java-Welt, Ant und Maven, direkt unterstützt. Sie können aber auch Shell-Skripte bzw. Batch-Dateien starten und somit ausgefallene oder proprietäre Build-Werkzeuge einbinden. Über Plugins erhalten Sie weitgehende Unterstützung für jüngere oder stärker spezialisierte Build-Werkzeuge wie etwa gant oder rake.

Direkte Unterstützung
von Ant, Maven, Shell und
Batch

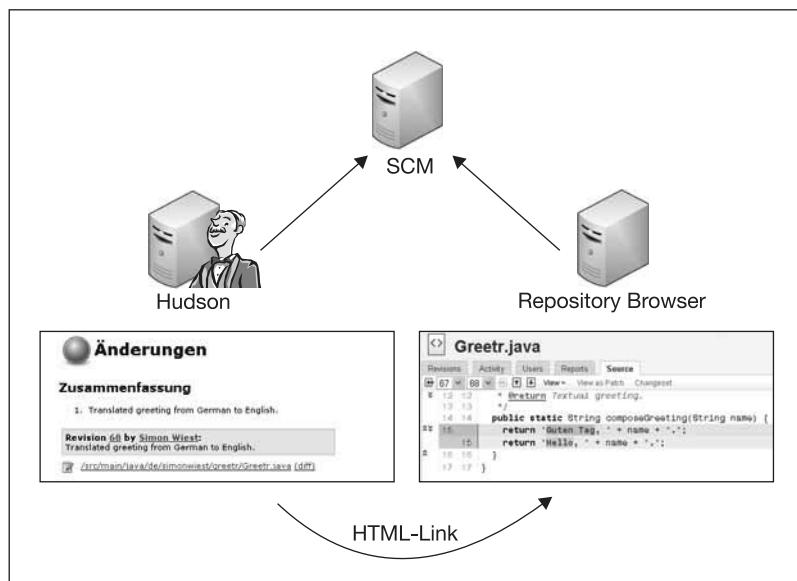
5.3.4 Anbindung von Versionsmanagementsystemen

Kontinuierliche Integration setzt voraus, dass alle für einen Build benötigten Daten in einem Versionsmanagementsystem verwaltet werden. Hudson unterstützt die »üblichen Verdächtigen«, CVS und Subversion, direkt. Kommerzielle Vertreter wie Perforce oder Open-Source-Projekte wie Git oder Mercurial können per Plugin angebunden werden. Hudson kann Versionsmanagementsysteme nicht nur nach Änderungen befragen, sondern auch detaillierte Informationen zu einzelnen *change sets* anzeigen, also wer wann was wo geändert hat.

Direkte Unterstützung von
Subversion und CVS

Wenn Sie einen spezialisierten Repository-Browser verwenden, um in Ihren versionierten Daten zu navigieren (z.B. FishEye, Sventon oder ViewSVN), so können Sie direkt aus Hudson über Links in entsprechende Übersichten im Repository-Browser springen. Abbildung 5-7 zeigt dies aus der Anwendersicht: Im linken Teil der Abbildung sehen Sie einen Ausschnitt aus Hudsons Weboberfläche. Dateinamen sind mit Links hinterlegt, die direkt zum Repository-Browser führen, der auf der rechten Seite dargestellt ist (hier: Atlassian FishEye).

Abb. 5-7
Verknüpfung zwischen
Hudson und einem
Repository-Browser



5.3.5 Testberichte

Direkte Unterstützung
von JUnit

Obwohl »kontinuierliche Integration« im strengen Wortsinn nur das regelmäßige Zusammenführen aller Produktbestandteile bezeichnet, wird in der Praxis auch das Testen mit eingeschlossen. Typische Test-Frameworks in der Java-Welt sind JUnit, TestNG oder Selenium. Für viele dieser Testwerkzeuge kann Hudson deren Ergebnisprotokolle verstehen und grafisch aufbereiten. Somit haben Sie in Hudson nicht nur alle Codeänderungen im Blick, sondern Seite an Seite auch die zugehörigen Testergebnisse. Abbildung 5–8 zeigt exemplarisch eine navigierbare Auswertung eines JUnit-Laufs auf Ebene eines Java-Packages.

Testergebnis : de.acme.controller							
Fehlgeschlagen(+5)							
Tests(40) Total: 4 Minuten 36 Sekunden Anzeigebereich hinzufügen							
Testname	Dauer Alter						
>>> de.acme.controller.ValueItemListPropertiesTabControllerTest testGetListValuesEditorProperties	5.097 1						
>>> de.acme.controller.ValueItemListPropertiesTabControllerTest testGetPropertiesSheet	6.179 1						
>>> de.acme.controller.ValueItemListPropertiesTabControllerTest testGetPropertiesSheetForNewValueItemList	7.109 1						
>>> de.acme.controller.ValueItemListPropertiesTabControllerTest testSavePropertiesForListValues	5.022 1						
>>> de.acme.controller.ValueItemListPropertiesTabControllerTest testUpdateListValues	5.01 1						
Alle Tests							
Klasse	Dauer	Fehlgeschlagen	(Veränderung)	Skip	(Veränderung)	Summe	(Veränderung)
AbstractPropertiesTabControllerTest	1 Minute 29 Sekunden	0	0	0		18	
EapPropertiesTabControllerTest	5 Sekunden	0	0	0		1	
FeaturePropertiesTabControllerTest	16 Sekunden	0	0	0		5	
FilePropertiesTabControllerTest	26 Sekunden	0	0	0		5	
LocalePropertiesTabControllerTest	16 Sekunden	0	0	0		3	
ProductPropertiesTabControllerTest	1 Minute 29 Sekunden	0	0	0		15	
RuleBasedPropertiesTabControllerTest	5,1 Sekunden	0	0	0		1	
ValueItemListPropertiesTabControllerTest	20 Sekunden	5	+5	0		5	

Abb. 5–8
JUnit-Testergebnisse
eines Java-Packages

5.3.6 Benachrichtigungen

Die wenigsten von uns werden den ganzen Tag vor einem Hudson-Browserfenster sitzen wollen, um Build-Probleme frühzeitig zu erkennen. Hudson kann Sie daher über eine Vielzahl von Kommunikationskanälen auf dem Laufenden halten, u.a. per E-Mail, Instant-Messenger, Twitter, RSS-Feed. Auch sogenannte *eXtreme Feedback Devices (XFD)* lassen sich mit minimalem Aufwand anschließen. So können Sie Erfolg und Misserfolg Ihrer Builds auf intuitive und originelle Weise signalisieren lassen. Wie wäre es beispielsweise mit einer Ampel aus Leuchtbären (Abb. 5–9)?

Direkte Unterstützung von
E-Mails und RSS-Feeds

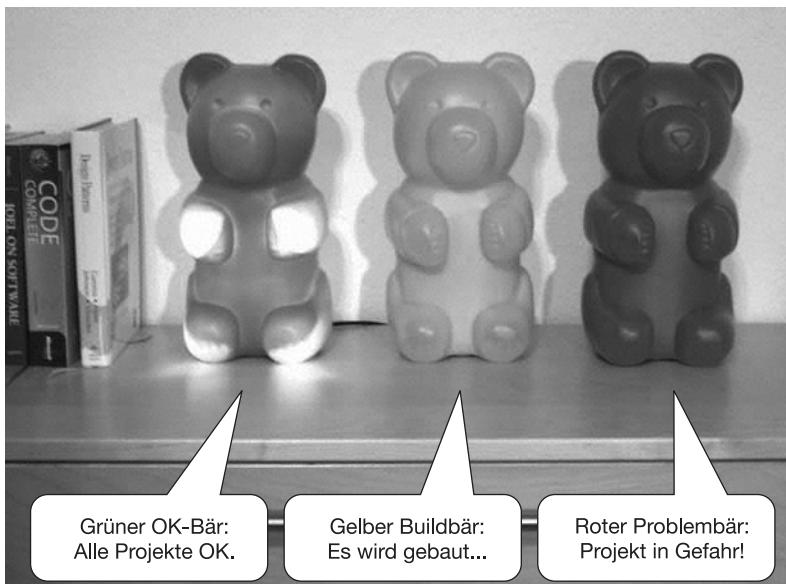


Abb. 5–9
»Bärenampel« als eXtreme
Feedback Device

5.3.7 Remoting-Schnittstelle

Effizienz durch Remoting

In den meisten Fällen wird man Hudson interaktiv über die Weboberfläche benutzen. Für bestimmte Aufgaben ist dies jedoch mühsam: Stellen Sie sich vor, Ihr Chef hat sich an Ihrer Begeisterung für Hudson angesteckt und weist Sie an, alle 236 Projekte Ihrer Abteilung in Hudson anzulegen. Manuelles »Durchklicken« wäre hier viel zu langsam und fehleranfällig. Stattdessen lässt sich Hudson mit einfachen REST-Kommandos fernsteuern, z.B. aus einer Batch-Datei oder einem Perl-Skript heraus. Die REST-Schnittstelle nimmt nicht nur Kommandos entgegen, sondern erlaubt auch den Abruf von Informationen über die Hudson-Instanz, ein bestimmtes Projekt, einen bestimmten Build usw. Um die maschinelle Weiterverarbeitung zu erleichtern, stellt Hudson diese Informationen in XML-, JSON- sowie Python-Notation bereit.

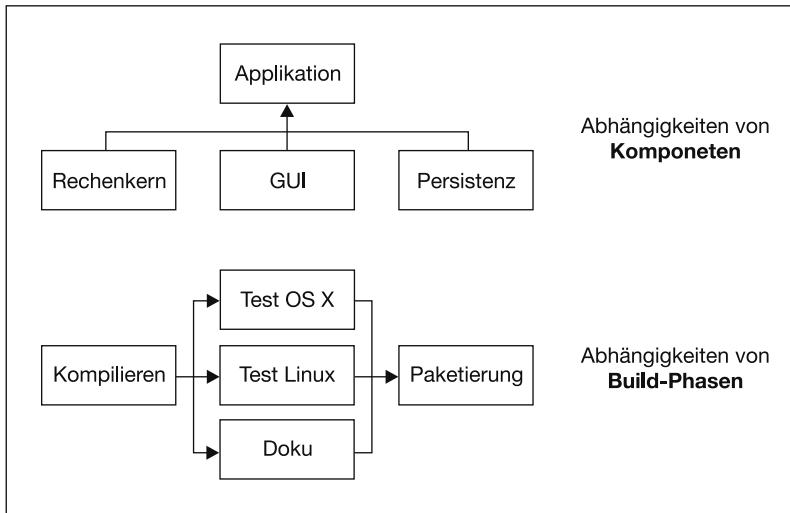
Zusätzlich können Sie mit einem Kommandozeilen-Client (CLI) die wichtigsten Funktionen einer Hudson-Instanz aus der Entfernung ausführen.

5.3.8 Abhängigkeiten zwischen Jobs

Automatisches Management von Abhängigkeiten

Die wenigsten Softwareprojekte werden völlig isoliert voneinander entwickelt, sondern stehen untereinander in Beziehung. Ein Beispiel: Eine Applikation setzt sich aus den drei Modulen Rechenkern, GUI (Benutzeroberfläche) und Persistenz (Datenhaltung) zusammen (Abb. 5–10, oben). Bei einer Änderung des Rechenkerns sollte nicht nur dieses Modul, sondern auch die resultierende Applikation neu gebaut und getestet werden, um Probleme im Zusammenspiel mit der GUI oder der Persistenzschicht zu entdecken. Hudson bietet dazu die Möglichkeit, Beziehungen zwischen vor- und nachgelagerten Projekten zu definieren. Der erfolgreiche Build des Rechenkerns stößt dann automatisch die Integration und den Test der Applikation an. Wünschenswerterweise werden dabei nur diejenigen Teile des Produkts neu gebaut und getestet, die von den Änderungen betroffen sein könnten. Das aufwendige Bauen *aller* Module nach *jeder* Änderung kann entfallen. Dies spart Ressourcen und verkürzt die Build-Zeit.

Abhängigkeiten können auch genutzt werden, um einen langen Build-Ablauf in Phasen zu unterteilen. Für jede Phase wird in Hudson ein eigenes Projekt mit entsprechenden Abhängigkeiten angelegt. Im Idealfall kann Hudson dann die Ausführung bestimmter Phasen parallelisieren und die Build-Zeit verringern (Abb. 5–10, unten).

**Abb. 5-10**

Abhängigkeiten zwischen
Projekten

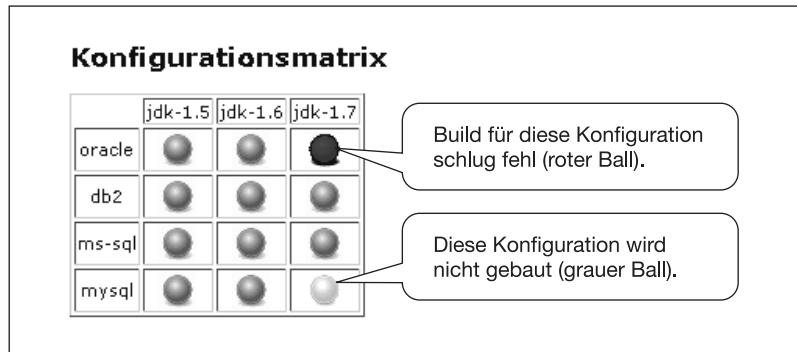
5.3.9 Multikonfigurationsbuilds (»Matrix-Builds«)

Stellen Sie sich vor, Sie entwickeln eine Software, die für drei Java-Versionen und vier Datenbankanbieter gebaut und getestet werden soll. Kommt Ihnen das bekannt vor? Sie könnten natürlich für jede dieser zwölf Kombinationen einen Job in Hudson anlegen. Übersichtlich und einfach zu verwalten wäre das allerdings nicht mehr.

*Verwandte
Konfigurationen in einem
Projekt verwalten*

Abhilfe schaffen hier Hudsons Multikonfigurationsbuilds, oft auch als Matrix-Builds bezeichnet: Dazu verändern Sie zunächst Ihr Build-Skript so, dass es mit zwei Parametern für Java-Version und Datenbankanbieter aufgerufen werden kann, etwa durch Setzen von Umgebungsvariablen vor dem Start des Build-Skripts. Dann legen Sie in Hudson einen Job vom Typ »Multikonfigurationsbuild« an. Sie definieren zwei Achsen für die Java-Version und den Datenbankanbieter mit den jeweiligen Ausprägungen. Wird dieser Job nun gestartet, baut und testet Hudson für Sie vollautomatisch alle 12 Kombinationen durch und stellt die Ergebnisse übersichtlich in einer Tabelle dar (Abb. 5-11). Sind bestimmte Kombinationen aus technischen oder fachlichen Gründen nicht möglich, können Sie diese gezielt vom Build ausschließen. Sie können sogar Einfluss auf die Reihenfolge der Abarbeitung der Kombinationen nehmen, um kurz laufende Builds zuerst auszuführen zu lassen. Somit erfahren Sie von neu aufgetretenen Problemen noch schneller und können früher mit deren Behebung beginnen.

Abb. 5-11
Effiziente Verwaltung
von Konfigurationen
durch Hudson



5.3.10 Verteilte Builds

Schnellere Builds
durch Verteilung

Buildzeiten haben die unliebsame Angewohnheit, stetig zu wachsen. Zum einen entsteht im Laufe eines Projekts mehr und mehr Quelltext, der kompiliert und getestet sein möchte. Zum anderen verlockt die Vollautomatisierung dazu, ein immer größeres Arsenal an Inspektionen und Analyseverfahren in den Build aufzunehmen. Fast jedes Entwicklerteam stellt sich daher irgendwann die Frage: »Wie schrumpfen wir unsere Build-Zeiten?«

Eine ebenso naheliegende wie mächtige Idee ist die Verteilung des Builds auf mehrere Rechner über das Netzwerk. Doch Vorsicht, der Teufel steckt hier im Detail:

- Was passiert, wenn ein Rechner zeitweilig vom Netz geht?
- Wie werden die verteilten Build-Ergebnisse wieder zentral zusammengeführt und dargestellt?
- Wie kann ein Rechner während des Builds auf Teilergebnisse eines anderen Rechners zugreifen?
- Wer verteilt die benötigten Build-Werkzeuge auf alle beteiligten Rechner?
- Wie können Rechner mit heterogenen Betriebssystemen gleichförmig angesprochen werden?

Hudson bietet hier zahlreiche Funktionen und Konzepte an, welche die Realisierung eines verteilten Builds immens erleichtern, z.B. Überwachung der Verfügbarkeit der beteiligten Rechner oder automatische Verteilung benötigter Build-Werkzeuge.

5.3.11 Plugins

Ihr Arbeitgeber verwendet aus historischen Gründen ein ausgefallenes Versionsmanagementsystem, der Senior-Architekt möchte seine Lieblingsmetriken in Hudson sehen, Sie möchten ein brandneues Werkzeug in Ihren Build-Prozess integrieren: In der Praxis werden Sie immer wieder vor unvorhersehbare Herausforderungen gestellt. Hudson hält dazu über 200 Plugins aus den unterschiedlichsten Anwendungsgebieten bereit. Sollte das passende Plugin fehlen, so ist zumindest die Chance groß, dass sich ein bestehendes Plugin für den eigenen Fall anpassen lässt. Die liberale MIT-Open-Source-Lizenz lädt ein, bestehenden Quelltext als Sprungbrett für eigene Entwicklungen zu verwenden. Falls Sie ein eigenes Plugin realisieren möchten, finden Sie in Kapitel 9 eine Einführung in die Plugin-Entwicklung für Hudson.

*Infrastruktur-Chamäleon
dank Plugins*

5.4 Hudson im Vergleich zu Mitbewerbern

Wie in der Einleitung des Buches bereits angesprochen, mag Hudson nicht für alle Arbeitsgruppen der ideale CI-Server sein. Bei aller Euphorie, die Sie von einem Hudson-Committer erwarten dürfen, sollen an dieser Stelle auch die wichtigsten Alternativen angesprochen werden.

Die Auswahl an CI-Servern ist zahlenmäßig groß. Eine Vergleichsmatrix mit den dreißig bekanntesten Vertretern finden Sie unter [ThoughtWorks10]. Trotzdem lässt sich eine gewisse Konzentration auf wenige Produkte feststellen. Im Folgenden sollen diese Alternativen zunächst vor- und dann Hudson gegenübergestellt werden.

Zuvor noch ein kurzes Wort der Warnung: Die vorliegende Auswahl der Alternativen in diesem Abschnitt stellt keine objektive Erhebung oder Empfehlung dar, sondern basiert auf der Häufigkeit der Nennung in persönlichen Gesprächen des Autors mit Anwendern. Vor- und Nachteile einzelner Produkte können sich zwischenzeitlich geändert haben, so dass Sie für Ihre Vorhaben stets die aktuellen Versionsstände miteinander vergleichen sollten. Alle hier genannten Produkte sind als kostenlose Testversionen (zeitlich beschränkt) oder Einsteigerversionen (leistungsbeschränkt) verfügbar, so dass Sie mit überschaubarem Aufwand Ihre Projekte in unterschiedlichen CI-Servern testen können.

5.4.1 Proprietäre Eigenentwicklungen

Jeder Vergleich wäre unvollständig, wenn er nicht zunächst auf die unzähligen, selbst entwickelten »Hauslösungen« zur Automatisierung des Build-Prozesses einginge.

Architektur und besondere Merkmale

In der Regel handelt es sich bei diesen Hauslösungen um einen bunten Mix von sich gegenseitig aufrufenden Programmen, die als Shell-Skripte, in Perl und einem halben Dutzend weiterer Skriptingsprachen im Lauf der Jahre gewachsen sind. Speziell in stark kontrollierten Branchen (etwa im Finanzsektor oder der Medizintechnik) werden gesetzliche Auflagen zur Dokumentationspflicht und Nachvollziehbarkeit oftmals durch proprietäre Erweiterungen realisiert, die eine spätere Migration auf verbreitetere Standardprodukte erschweren. Darüber hinaus kann der Wartungsaufwand nicht unerhebliche Dimensionen erreichen: Ein Build-System benötigt zahlreiche Schnittstellen zu weiteren Systemen (u.a. Versionsverwaltung, Repositories, Testsysteme, Staging-/Produktionsserver, Benutzerverwaltung). Da diese Systeme sich laufend weiterentwickeln, müssen auch die Schnittstellen eines hauseigenen Build-Systems angepasst werden. Durch den proprietären Charakter können dafür aber nur eigene Mitarbeiter eingesetzt werden – von weit verfügbaren und breit getesteten Lösungen kann in diesem Falle nicht profitiert werden.

Typischerweise wird in diesen Umgebungen nicht kontinuierlich nach Änderungen gebaut, sondern in festen Zeitintervallen, etwa einmal pro Nacht oder Woche. Öfter könne ja auch nicht gebaut werden, so die häufig zu hörende Begründung, da der Build über mehrere Stunden laufe.

Muss bereits großer Aufwand in die Erstellung und laufende Pflege des Build-Systems gesteckt werden, ist es nicht verwunderlich, dass Auswertungen über mehrere Builds hinweg eher stiefmütterlich implementiert werden. Nicht selten beschränkt sich das »Reporting« auf E-Mails, die am Ende des Build-Prozesses abgesetzt werden, oder auf unstrukturierte Protokolldateien im zweistelligen Megabyte-Bereich.

Vergleich zu Hudson

Glücklicherweise weisen die genannten Probleme bereits auf ihre Lösung: Umstellung auf verbreitete Build-Werkzeuge, Zerlegen des Build-Prozesses in kürzere Teilmodule sowie Visualisierung der Ergebnisse durch eine darauf spezialisierte Anwendung, also durch einen CI-Server wie Hudson.

In der Praxis gestaltet sich dieser Wandel jedoch schwieriger als erwartet: Es bremsen mangelnde Ressourcen (»Der Kunde zahlt schließlich nur das Produkt, nicht den Prozess.«), Angst vor Risiken (»Da kennt sich keiner aus – besser nichts ändern ...«), aber auch persönliche Befindlichkeiten (»Werden ich oder meine Skripte verzichtbar?«). Realistisch betrachtet wird man in diesen Fällen eine Verbesserung nur durch viele kleine, inkrementelle Verbesserungen herbeiführen können, beginnend mit der Kapselung der Schnittstellen zu Drittsystemen und Modularisierung von Build-Phasen. Diese Module werden dann sukzessive auf gängige Build-Werkzeuge migriert und dienen als natürliche Trennstellen zum Zerlegen monolithischer »Bandwurm-Builds«.

Parallel dazu können bestehende Build-Prozesse bereits als *black box* durch CI-Server angestoßen werden. Oft lässt sich schon in dieser Phase das Ausgabeformat eines Builds so ändern, dass die Visualisierung durch den CI-Server übernommen werden kann. Beispiel: Werden Testergebnisse nicht unstrukturiert in eine Protokolldatei geschrieben, sondern als JUnit-kompatible XML-Datei ausgegeben, können praktisch alle gängigen CI-Server diese Ergebnisse aus dem Stand visualisieren.

Fazit

Hauseigene CI-Systeme starten zwar klein und schnell. Sie tendieren aber dazu, zu hochkomplexen Monstern zu wachsen, die nur noch von Mitarbeitern mit Kopfmonopol überschaut werden können. Möchte man sich das Nachentwickeln von CI-Grundfunktionen sparen, gleichzeitig aber eine weitgehende Integration mit hauseigenen Systemen erreichen, drängt sich als Alternative ein quelloffenes Produkt mit guten Erweiterungsmöglichkeiten auf – etwa Hudson.

5.4.2 CruiseControl

CruiseControl (<http://cruisecontrol.sourceforge.net>) darf zu Recht als der Altvater der CI-Systeme bezeichnet werden. Seit fast einer Dekade leistet CruiseControl unauffällig, aber zuverlässig seine Dienste und stellte für viele Entwickler die Eintrittskarte in die Welt der CI dar. Ursprünglich von der Firma ThoughtWorks initiiert, wurde das Projekt bereits früh als Open Source veröffentlicht und ist kostenlos auf der Sourceforge-Plattform verfügbar.

Altvater der CI-Systeme

Da zahlreiche Artikel und Anleitungen im Netz auf dieses Projekt verweisen, steht der Name CruiseControl in vielen Arbeitsgruppen sogar synonym für Continuous Integration.

CruiseControl.NET

Für weitere Bekanntheit sorgte das Schwesterprojekt CruiseControl.NET (<http://ccnet.thoughtworks.com>), das vornehmlich die Werkzeugwelt der Microsoft-Landschaft bedient, etwa NAnt, MSBuild, NUnit, NDepend.

Abb. 5–12

*Build-Übersicht in
CruiseControls
»Dashboard«-Oberfläche*

**Architektur und besondere Merkmale**

*Zwei Komponenten:
»Die Schleife« und die
Weboberfläche*

Ein CruiseControl-Server besteht aus einem Prozess, der zeitgesteuert nach Änderungen in Versionsmanagementsystemen horcht und gegebenenfalls neue Builds anstößt. Nach Beendigung des Builds werden Benachrichtigungen abgesetzt. Da dieser Prozess endlos läuft, wird er als »die Schleife« (*the loop*) bezeichnet. Abgesehen von einer Konsoleausgabe hat dieser Prozess keine Oberfläche, über die ein Benutzer den Fortschritt beobachten könnte. Stattdessen liefert CruiseControl gleich zwei webbasierte GUIs mit: die »klassische« Ansicht und das *dashboard* (Abb. 5–12, Abb. 5–13). Beide visualisieren den Status der Schleife und zeigen Informationen aus vergangenen Builds an. Technisch betrachtet handelt es sich bei beiden GUIs um Java-Webapplikationen, die in einem schlanken Jetty-Container betrieben werden.

CruiseControl fokussiert sich auf die wichtigsten CI-Schritte »Änderungen erkennen, Projekt bauen, Team benachrichtigen«. Umfangreiche Visualisierungen oder ein ausgefeiltes Rechte- und Rollenmodell sucht man hier vergebens.

*Wenige Möglichkeiten
zur Interaktion*

Ohne sein großes Verdienst um die Popularisierung des CI-Gedankens schmäler zu wollen, merkt man CruiseControl sein Alter an: Die Oberflächen dienen ausschließlich der passiven Betrachtung des Fortschritts. Außer dem manuellen Auslösen eines Builds stehen hier keine

Aktionen zur Verfügung. Auch lässt sich CruiseControl nicht über die Oberfläche konfigurieren. Dies erfolgt ausschließlich über Änderungen an einer XML-Datei. Gerade an diesem Umstand entzündete sich bei vielen Anwendern der Unmut. Wie schwer dieser Nachteil in der Praxis wiegt, kommt auf den Umfang der Änderungsaktivitäten an. Gerade bei sporadisch durchgeführten Wartungsarbeiten wünscht man sich eine geführte Konfigurationsmöglichkeit herbei, die Benutzereingaben automatisch validiert und kontextsensitive Hilfen bereithält.

CruiseControl unterstützt in der aktuellen Version auch verteilte Builds über mehrere Rechner in einer Master-Slave-Architektur durch ein optionales Erweiterungspaket. Master und Slaves müssen dabei individuell eingerichtet, konfiguriert und gestartet werden. Die Zuteilung von Build-Aufträgen und das Einsammeln der dezentral entstandenen Build-Ergebnisse auf den Master erfolgt dann automatisch.

Verteilte Builds

Build	Time	Description
build.4	1 minute ago	greetr passed (1 minute ago)
build.3	3 minutes ago	de.simonwies.greetr.GreeterAppTest
build.2	5 minutes ago	de.simonwies.greetr.GreeterTest
build.1	7 minutes ago	de.simonwies.greetr.GreeterAppTest
build.0	12 minutes ago	de.simonwies.greetr.GreeterTest
build.-1	24 minutes ago	de.simonwies.greetr.GreeterAppTest

Abb. 5-13

*Ansicht eines Builds in
CruiseControls
»Dashboard«-Oberfläche*

Vergleich mit Hudson

Im Vergleich zu Hudson fällt zunächst auf, dass CruiseControl – der Name ist Programm – nur ein Minimum an Benutzerinteraktion benötigt und zulässt. Hudson hingegen kann zur aktiven Schaltzentrale eines Entwicklungsteams ausgebaut werden, weil es neben deutlich überlegenen Analyse- und Visualisierungsmöglichkeiten zahlreiche Schnittstellen zu Drittsystemen bietet.

*Engerer Aufgabenfokus
bei CruiseControl*

Des Weiteren ist CruiseControl weitestgehend agnostisch gegenüber den im Build eingesetzten Werkzeugen. Grundsätzlich ist dies kein Nachteil, da es maximale Flexibilität in der Auswahl der Werkzeuge

*Bessere
Werkzeugunterstützung
bei Hudson*

ermöglicht. Versteht aber ein CI-System auch den internen Aufbau eines Builds, etwa durch Auswertung einer Maven-POM-Datei, kann es eine wesentlich bessere Werkzeugunterstützung anbieten. Beispielsweise kann Hudsons Maven-Projekttyp durch Analyse von POM-Dateien automatisch Build-Abhängigkeiten zwischen Projekten aufbauen und die richtigen Artefakte im Build-Arbeitsverzeichnis auffinden.

*Verteilte Builds möglich,
aber in Hudson
komfortabler*

*Keine Rechte und Rollen
bei CruiseControl*

Zukunft unklar

Sowohl CruiseControl als auch Hudson unterstützen grundsätzlich verteilte Builds. Der Unterschied zeigt sich jedoch im täglichen Betrieb. Hier fehlen CruiseControl die Funktionen zur zentralen Überwachung und dem Ausbringen der benötigten Build-Infrastruktur auf die Slave-Knoten.

Im Unternehmensumfeld dürften vor allem CruiseControls fehlendes Rechte- und Rollensystem Probleme bereiten. Zwar lässt sich der Zugriff auf die Weboberfläche mittels der Sicherheitsmechanismen des ausführenden Webcontainers einschränken. Ein feingranulares Rechtemodell ist hingegen nicht vorgesehen, etwa um die Sichtbarkeit von Projekten für einzelne Benutzer einzuschränken. Auf der anderen Seite existieren ja auch nur wenige »aktive« Funktionen, die es abzusichern gälte ...

Der gravierendste Unterschied dürfte jedoch die deutlich abklingende Weiterentwicklung CruiseControls sein, die einem dynamischen Wachstum der Entwicklergemeinde Hudsons gegenübersteht. Aktualisierungen erscheinen für CruiseControl nur noch in monatelangen Abständen und überwiegend zur Fehlerkorrektur. Große Versionssprünge dürften bei CruiseControl in naher Zukunft nicht zu erwarten sein.

Fazit

Obwohl CruiseControl sicherlich auch weiterhin in vielen Abteilungen zuverlässig und robust seine Dienste leisten wird, sollten bei Neueinführungen dringend Alternativen geprüft werden. Soll eine Lösung aus dem Open-Source-Lager zum Einsatz kommen, wäre Hudson hier sicher ein vielversprechender Kandidat, der alles kann, was CruiseControl anbietet – und noch mehr.

5.4.3 ThoughtWorks Cruise

Viele Anwender erhofften sich 2008 mit der ersten öffentlichen Version von Cruise einen zeitgemäßen (wenngleich auch kommerziellen) Nachfolger von CruiseControl. Zwar suggeriert die sicher gewollte Namensähnlichkeit eine Verwandtschaft zwischen beiden Produkten. Der Hersteller ThoughtWorks betont jedoch ausdrücklich, dass sich

beide Produkte nur in geringem Umfang Code teilen würden. Verständlich, denn schließlich müssen bisherige CruiseControl-Anwender erst einmal überzeugt werden, von einem kostenlosen Produkt auf dessen kommerzielles Pendant zu wechseln. Dementsprechend gering sind die Marktanteile für Cruise bisher.

ThoughtWorks positioniert Cruise als ein System für Continuous Integration, das darüber hinaus Software-Deployment und Release-Management abdeckt. In der Praxis kann man sich Cruise als ein CruiseControl-System vorstellen, dem ein Workflow-System zur schrittweisen Freigabe von Builds in modernem Web-2.0-Gewand übergestülpt wurde. Die folgende Betrachtung bezieht sich auf die Version 1.3.2 (September 2009).

The screenshot shows the 'cruise' web interface. At the top, there are tabs for 'CURRENT ACTIVITY', 'AGENTS', 'MY CRUISE', and 'ADMINISTRATION'. Below the tabs, the 'Current Activity' section displays a hierarchical tree of build jobs. The tree starts with 'Greetr' at the root level, which has three children: 'compile_unittests', 'functional-tests', and 'documentation'. 'compile_unittests' has one child, 'compile', which is marked as 'passed'. 'functional-tests' has one child, 'selenium', also marked as 'passed'. 'documentation' has three children: 'Javadocs', 'schemaspy', and 'umlgraph', all marked as 'passed'. Under 'documentation', there is a 'deploy' node with three children: 'deploy-server-ochsenhausen', 'deploy-server-schoeneburg', and 'deploy-server-wennedach', all marked as 'passed'. Each job entry includes a timestamp, a label indicating the last successful build, and a link to the build details. At the bottom of the page, there is a copyright notice for ThoughtWorks (2009) and links for 'Need Help?' and 'Support'.

Abb. 5-14

Aktivitätsübersicht
in Cruise

Architektur und besondere Merkmale

Cruise ist erhältlich für Windows, Mac OS X und Linux. Als Versionsmanagementsysteme werden Subversion, Perforce, Git und Mercurial direkt unterstützt, wenige weitere können über separat zu installierende Erweiterungen angebunden werden.

Cruise arbeitet mit einer Master/Slave-Architektur, wobei die Slave-Knoten bei Cruise als Agenten (*agents*) bezeichnet werden. Für jeden Agenten lassen sich seine charakteristischen Eigenschaften angeben, z.B. Betriebssystem, JDK-Version oder installierte Datenbanken. Werden den Build-Jobs dazu korrespondierende Anforderungen hinterlegt, führt Cruise diese auf passenden Agenten aus.

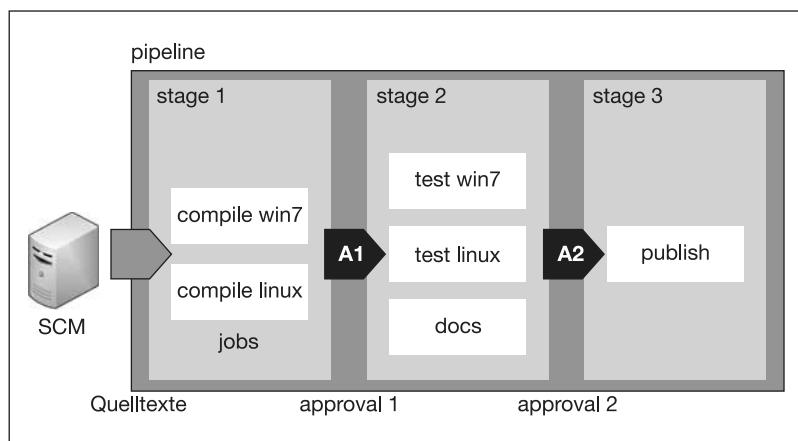
Verteilte Builds

Pipeline-Modell

Das Alleinstellungsmerkmal von Cruise dürfte das sogenannte *Pipelining*-Modell sein. Hierbei wird der Build-, Release- und Deployment-Prozess in mehrere Stufen (*stages*) unterteilt (siehe Abb. 5–15). Innerhalb dieser Stufen werden Arbeitspakete in Jobs aufgeteilt, die parallel und auf unterschiedlichen Agenten ausgeführt werden können. Die Pipeline kann durch Änderungen im Versionsmanagementsystem, durch den Abschluss anderer Pipelines oder manuell gestartet werden. Dabei werden die aktuellen Quelltexte aus dem Versionsmanagementsystem den Jobs der ersten Stufe zugeführt. Wurden alle Jobs einer Stufe abgearbeitet, muss eine Freigabe (*approval*) für die nächste Stufe erteilt werden. Dies geschieht automatisch, in Abhängigkeit von Testergebnissen oder durch manuelle Freigabe, z.B. durch den QA-Manager. Als letzte Stufe kann die Veröffentlichung des Produkts in den Produktionsbetrieb stehen. Dieser mehrstufige Freigabeprozess, der von einer Änderung im Versionsmanagementsystem bis hin zum Produktionsbetrieb durchläuft, stellt den größten Mehrwert von Cruise gegenüber anderen CI-Servern dar.

Abb. 5–15

Pipelining in Cruise

**XML-Konfiguration**

Cruise bietet eine aufgeräumte Weboberfläche mit zeitgemäßer Optik (Abb. 5–14). Leider beschränkt sie sich auf die Darstellung der Pipelines und Stufen. Die Konfiguration des Systems erfolgt ausschließlich über XML-Fragmente, die in großen Textfeldern editiert werden. Echtzeitvalidierung der Eingaben, Syntaxhervorhebung oder automatische Codevervollständigung fehlen.

Über eine Anbindung an LDAP bzw. Active Directory lässt sich der Zugriff auf Cruise gezielt einschränken.

Cruise ist in zwei Editionen erhältlich: Cruise Free und Cruise Professional. Für Erstere kann eine kostenlose Jahreslizenz bei ThoughtWorks angefordert werden. Cruise Free ist auf 10 Benutzer (Personen,

Rechte und Rollen**Lizenzzmodell**

die im Versionsmanagement Änderungen einchecken) und 10 lokale Agenten eingeschränkt. An verschiedenen Stellen der Produkt-Website wird diese Edition auch als Cruise Trial bezeichnet, was die Intention des Herstellers wohl besser darlegen dürfte. Für höhere Anforderungen ist Cruise Professional erforderlich. Lizenzen kosten je nach Anzahl der Benutzer und Agenten ab 3.200 USD pro Jahr.

Vergleich mit Hudson

Dem Anwender stellt sich Cruise als CruiseControl plus Freigabe-Workflow mit Weboberfläche dar. Unverständlich ist dabei, dass in der Teildisziplin »Continuous Integration« Cruise leider sogar dem »CI-Opa« CruiseControl unterliegt. So fehlen beispielsweise grundlegende Dinge wie eine Möglichkeit zur Zeitplanung von Builds: Momentan werden alle überwachten Versionsmanagementsysteme minütlich abgefragt. Für den Unternehmenseinsatz ist dies nicht ausreichend flexibel.

CI-Bereich noch schwächer als CruiseControl

Ist man im ersten Moment von der aufgeräumten Weboberfläche angetan, erstaunt die Beibehaltung des gusseisernen XML-Konfigurationskonzeptes aus CruiseControl-Tagen. Es verwundert, dass ThoughtWorks auf der einen Seite die Oberfläche als »usability-driven« bezeichnet, auf der anderen Seite aber den Hauptkritikpunkt an CruiseControl, die ausschließliche Konfiguration per XML ohne wirkliche Hilfsmittel, ins Herz des kommerziellen Nachfolgers pflanzt.

XML-Konfiguration

Die Dokumentation zu Cruise behandelt zwar alle Masken und Eingabefelder, bietet aber wenig technische Hintergrundinformation. Diese finden sich im – schwach frequentierten – Onlineforum. Ein Entwicklerprogramm oder eine Plugin-API existieren momentan nicht, was den Ausbau und die Weiterentwicklung des gesamten Produktes exklusiv auf den Schultern von ThoughtWorks lasten lässt. Es ist fraglich, ob sich mit diesem Modell die Werkzeugvielfalt in Build-Ketten abdecken lässt. Sogar für bereits etablierte Build-Werkzeuge, wie etwa Maven, fehlen intelligente Integrationen. So kann etwa ein Maven-Build momentan lediglich als externe Kommandozeilenanwendung aufgerufen werden. Weitergehende Funktionen, wie etwa die automatische Konfiguration eines Jobs durch Auswertung einer POM-Datei, sind nicht verfügbar.

Keine Plugin-API

Fazit

Ist Cruise also der legitime Nachfolger von CruiseControl? Nein. So interessant der Pipeline-Workflow in Cruise sein mag: Die CI-Grundfunktionen sind momentan sogar noch schwächer als die des wesentlich älteren CruiseControls. Wer bei CruiseControl Wert auf die kostenlose Verfügbarkeit gelegt hat, wird sich daher eher Hudson

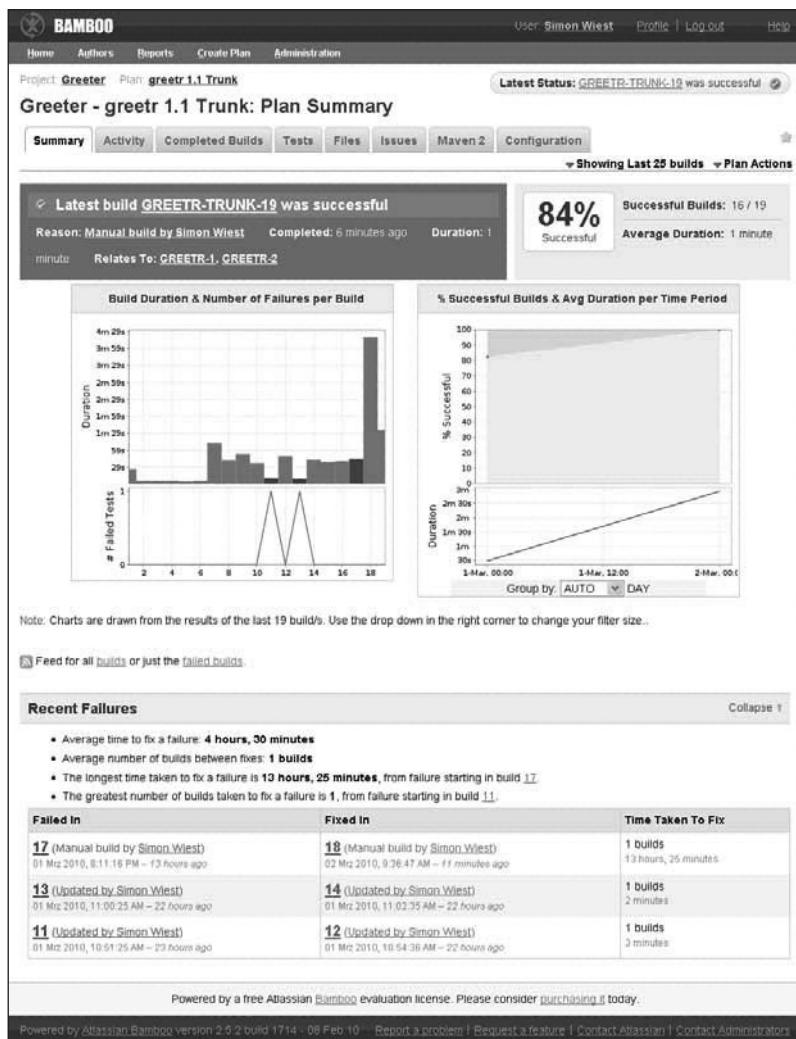
zuwenden. Wenn kommerzielle Vertreter ebenfalls in Betracht gezogen werden dürfen, gibt es aber auch gute Gründe, die Produkte Atlassian Bamboo und JetBrains TeamCity näher zu betrachten. Dies werden wir in den folgenden zwei Abschnitten tun.

5.4.4 Atlassian Bamboo

Die Firma Atlassian ist vielen Anwendern der IT-Branche als Hersteller des Enterprise-Wikis Confluence und des Issue-Trackers JIRA ein Begriff. In den letzten Jahren konnte Atlassian sein Portfolio rund um Softwareentwicklung und webbasierte Zusammenarbeit stetig erweitern, so auch 2007 mit dem CI-Server Bamboo. Die folgende Betrachtung bezieht sich auf Version 2.5.2 (Februar 2010).

Abb. 5–16

Ansicht eines Build-Plans
in Bamboo



Architektur und besondere Merkmale

Bamboo ist eine Java-Webapplikation, die entweder im mitgelieferten Jetty-Server betrieben (*standalone distribution*) oder in einen existierenden Tomcat-Server ausgebracht (*EAR-WAR distribution*) wird. Als Serverbetriebssysteme werden Microsoft Windows, Linux/Solaris und Apple Mac OS X offiziell unterstützt. Bamboo benötigt ein Java Development Kit (kein JRE) ab Version 1.5 und setzt eine Datenbank zur Speicherung seiner Daten voraus. Offiziell werden MySQL 5.x, PostgreSQL ab Version 8.2, Microsoft SQL Server 2005/2008, Oracle 10g/11g und HSQLDB (zu Testzwecken) unterstützt.

Java-Webapplikation

Bamboo arbeitet mit einer verteilten Architektur, bei der ein zentraler Server angelegte Projekte mit Build-Plänen (*build plans*) verwaltet und Build-Aufträge zur Ausführung an Build-Agenten (*build agents*) delegiert. Abschließend werden die Build-Ergebnisse der Agenten wieder eingesammelt, auf dem Server archiviert und über die Weboberfläche visualisiert. Für alle Agenten werden deren Fähigkeiten (*capabilities*) konfiguriert, etwa installierte JDKs, Betriebssystemversionen oder Build-Werkzeuge (z.B. Ant, Maven, Bash-Shell). Pro Build-Plan werden Anforderungen (*requirements*) an die Agenten deklariert, etwa »Dieser Plan benötigt JDK 1.6 auf einem Windows-Rechner mit Oracle 11g mindestens 2 CPUs«. Durch einen Abgleich der Fähigkeiten der Agenten und den Anforderungen eines Build-Plans entscheidet der Server, auf welchem Agenten ein Build zur Ausführung kommt.

Master-Slave-Architektur

Bei den Build-Agenten wird zwischen lokalen Agenten (*local agents*) unterschieden, die zusammen mit dem Server in der gleichen JVM ausgeführt werden, und entfernten Agenten (*remote agents*), die autonom laufen, in der Regel auf einer anderen Hardware als der Server. Entfernte Agenten sind Java-Anwendungen, die als JAR-Dateien entweder manuell gestartet werden oder durch einen betriebssystem-abhängigen Java Service Wrapper gestartet und überwacht werden (*remote agent supervisor*).

Bamboo kann über Plugins erweitert werden. Zum Zeitpunkt der Drucklegung dieses Buches waren rund 35 Plugins verfügbar, von denen der überwiegende Teil allerdings (noch) nicht mit der aktuellen Bamboo-Version kompatibel war.

Plugin-Konzept

Die Benutzeroberfläche folgt weitestgehend den Atlassian-typischen Konventionen, die aus den anderen Produkten des Herstellers bekannt sind Abb. 5–16, Abb. 5–17). Zusätzlich zur interaktiven Oberfläche steht auch eine REST-Schnittstelle zur Verfügung, mit der die wichtigsten Informationen abgefragt werden können, z.B. eine Liste aller angelegten Projekte, der Ausgang eines Builds oder die erzeugten Artefakte eines Builds.

Benutzeroberfläche

Abb. 5-17
Ansicht eines Builds in
Bamboo

The screenshot shows the Bamboo web interface for a build named "GREETR-TRUNK-19" from the "greeter" project. The build was triggered by "Simon Wiest" and completed successfully on "02 Mar 2010, 9:46:14 AM". The build took 1 minute and ran on the "Default Agent". The revision is 73. The "Code Changes" section lists a single commit by Simon Wiest adding tests for greeting unknown strangers. The "JIRA ISSUES" section lists two issues: "GREETER-1" (Related) and "GREETER-2" (Related), both of which have been changed to fixed. The "Comments" section contains a single entry from Simon Wiest stating it was the first build on new hardware. The "Tests" section indicates 4 tests in total. At the bottom, there are links for purchasing a license, reporting problems, requesting features, and contacting Atlassian administrators.

Benutzerverwaltung

Bamboo kommt mit einer eingebauten, autarken Benutzerverwaltung, kann jedoch auch an einen externen LDAP-Server angebunden werden. Bei komplizierteren Anforderungen lassen sich über das Single-Sign-On-Produkt Atlassian Crowd weitere Benutzerverzeichnisse und Technologien verwenden, z.B. Microsofts Active Directory. Rechte und Rollen lassen sich feingranular und übersichtlich zuordnen.

Atlassian sieht Bamboos Stärken in folgenden vier Bereichen liegen:

■ *Schnelles Einrichten neuer Build-Pläne:*

Gängige Infrastrukturen (etwa Subversion in Kombination mit Maven, JUnit und E-Mail-Benachrichtigungen) werden direkt unterstützt und können assistentengeführt schnell und einfach angelegt werden. Build-Pläne für unterschiedliche Entwicklungs-zweige (z.B. trunk, release candidate, maintenance) werden in Projekten zusammengefasst, was die Konfiguration beschleunigt und die Übersichtlichkeit erhöht.

■ *Enge Integration mit anderen Atlassian-Produkten:*

Bamboo fühlt sich nicht nur so an wie die anderen Produkte aus der Atlassian-Familie, es verwendet auch intern die gleichen Technologien, wie zum Beispiel Schnittstellen zu Atlassian Crowd. Darüber hinaus lassen sich die Produkte auch untereinander verknüpfen, so dass beispielsweise ein JIRA-Ticket automatisch einen

Verweis auf zugehörige Bamboo-Builds erhält und von dort wiederum auf eine Darstellung der zugehörigen Codeänderungen in Fisheye gesprungen werden kann.

■ *Hohe Skalierbarkeit:*

Bamboos verteilte Architektur bewältigt auch umfangreiche Build-Aufkommen einer Abteilung oder eines ganzen Unternehmens. Zur Abdeckung von großen Spitzenlasten können entfernte Agenten auch in elastischen Instanzen (*elastic instance*) in der Amazon EC2 Cloud nach Bedarf gestartet werden. Bei diesen Instanzen handelt es sich um von Atlassian vorkonfigurierte Systeme, die jeweils einen entfernten Agenten enthalten. Dieser Agent wird nach Hochfahren der Instanz aktiv und steht dann dem Bamboo-Server für Builds zur Verfügung.

■ *Berichte und Visualisierungen:*

Bamboo bietet zahlreiche Kennzahlen und Auswertungen auf unterschiedlichsten Ebenen, etwa Laufzeiten per Build, JUnit-Testergebnisse per Java-Package, Builds per Committer usw. Die Berichte werden als Text oder als Diagramm dargestellt.

Bamboos Lizenzmodell orientiert sich an der Anzahl der Build-Agenten. Werden keine entfernten Agenten und nur bis zu 10 Build-Pläne benötigt, ist Bamboo mit Lizenzgebühren von 10 USD praktisch kostenlos. Kommen hingegen entfernte Agenten zum Einsatz, steigen die Gebühren (ab 2.000 USD). Die Anzahl der Build-Pläne ist in diesen Fällen unbegrenzt. Open-Source-Projekte können kostenlose Lizenzen beantragen.

Lizenzmodell

Vergleich mit Hudson

Arbeitet man in einer Mainstream-Infrastruktur (z.B. mit Subversion, Ant/Maven, JUnit und E-Mail-Benachrichtigungen) und benötigt vor allem die grundlegenden CI-Funktionen wie automatisches Bauen von Projekten nach Codeänderungen, so ist Bamboo ein unaufgeregter und zuverlässiger Begleiter, der sich gut in eine bestehende Atlassian-Landschaft integriert.

Sollen jedoch auch neuere, noch weniger verbreitete Werkzeuge angebunden oder Berichte von spezielleren Testframeworks visualisiert werden, hat Hudson dank seiner Plugin-Vielfalt klar die Nase vorn. Bamboo verfügt zwar ebenfalls über eine Plugin-Schnittstelle, die es erlaubt, eigene Plugins zu entwickeln – Hudsons Plugin-Szene erscheint aber um ein Vielfaches vitaler. Der überwiegende Teil der Bamboo-Plugins war zudem zum Zeitpunkt der Drucklegung dieses Buches mit der aktuellen Version inkompatibel und wirkte verwaist. Ein Grund dafür

Plugins stiefmütterlich behandelt

dürfte das unübersichtliche Geflecht aus Wiki-Seiten der Online-Dokumentation zur Bamboo und dessen Plugins sein, bei denen ein Einsteiger leicht den Überblick verliert. Hier fehlt Bamboo (noch) eine eingebaute Plugin-Verwaltung nach dem Vorbild des »großen Bruders« Confluence, in der verfügbare und installierte Plugins inklusive Versionsnummern anzeigt und konfiguriert werden können.

*Maven-Integration gut,
in Hudson besser*

Die Maven-Integration ist mit Bamboos Version 2.5 wesentlich verbessert worden. So können neue Build-Pläne durch Auswerten einer vorhandenen POM-Datei deutlich schneller angelegt werden. Hudsons Maven-Unterstützung ist in dieser Hinsicht jedoch noch einen Schritt weiter entwickelt, z.B. durch die Multi-Modul-Darstellung oder dem automatischen Erkennen abhängiger Maven-Projekte.

Verteilte Builds

Bamboos Konzept des Abgleichs von Fähigkeiten der Agenten einerseits und Anforderungen der Build-Pläne andererseits ist wesentlich eingängiger als Hudsons Label-Konzept. Ähnlich wie Hudson unterstützt auch Bamboo Builds in der Amazon EC2 Cloud. Bamboos vorkonfigurierte Instanzen funktionieren tatsächlich ohne viel Konfigurationsaufwand. Allerdings stellen sich im Alltag prompt die kleinen, aber lästigen praktischen Probleme ein: So muss beispielsweise der Agent in der elastischen Instanz über das Internet Zugriff auf das Versionsverwaltungssystem haben, von dem er Code auschecken soll. Hier zögern Administratoren, diese firmeninterne Schatzkiste zum Internet hin zu öffnen. Sicherlich sind alle diese Probleme technisch lösbar. Die Euphorie um Build-Skalierung in der Wolke »mit ein paar Mausklicks« wird dadurch allerdings deutlich gedämpft.

*Verteilung der
Build-Werkzeuge*

Eine weitere Herausforderung verteilter Builds stellt die Verteilung der benötigten Build-Werkzeuge dar. Schließlich werden zum Bauen eines Projekts nicht nur die Quelltexte, sondern auch die passenden Werkzeuge benötigt, etwa Compiler und spezialisierte Development Kits. Bamboo setzt voraus, dass diese durch einen Administrator bereits auf den Systemen der entfernten Agenten eingerichtet und in Bamboo eingetragen wurden. Hudson kann hingegen aktiv die zentrale Verteilung von Build-Werkzeugen an Slave-Knoten übernehmen und sogar JDKs sowie Ant- und Maven-Installationen in unterschiedlichen Versionen direkt aus dem Internet auf Slave-Knoten installieren. Wie stark man davon profitieren kann, hängt natürlich davon ab, wie viele Knoten man verwalten muss und wie häufig sich Änderungen in der Konfiguration ergeben.

Fazit

Arbeitet man in einer Infrastruktur, die ausschließlich gängige Werkzeuge einsetzt, und hat man vielleicht sogar schon das eine oder andere Atlassian-Produkt im Haus, dann kann Bamboo der ideale fehlende Mosaikstein für das Aufgabengebiet »Continuous Integration« sein. Möchte man hingegen auch neueste Technologien anbinden oder gar eigene Plugins für hausinterne Werkzeuge schreiben, so punktet Hudson hier mit ungleich mehr Beispielen und Material im Internet und vor allem einer deutlich vitaleren Entwicklergemeinde.

5.4.5 JetBrains TeamCity

Das vierte Produkt, auf das hier eingegangen werden soll, ist TeamCity von JetBrains. Es hat einen ähnlichen Hintergrund wie Atlassians Bamboo: Auch TeamCity wird von einem Werkzeugspezialisten hergestellt, der sich durch seine Produkte einen sehr guten Namen unter Entwicklern erarbeitet hat – in diesem Falle durch die Entwicklungs-umgebung IntelliJ IDEA und das Visual Studio Add-In ReSharper. Analog zu den Mehrwerten, die Atlassian durch die Integration von Bamboo mit seinen Flaggschiff-Produkten zu schaffen versucht, sind bei JetBrains interessante Ansätze im Zusammenspiel von TeamCity mit interaktiven Entwicklungsumgebungen (IDEs) zu erwarten. Die folgende Betrachtung bezieht sich auf die Version 5.1 (April 2010).

The screenshot shows the TeamCity web interface. At the top, there are navigation links: Projects, My Changes, Agents (2), Build Queue (0), Administration, My Settings & Tools, Welcome, TeamCity Administrator, and Logout. Below the navigation bar, the main content area displays a list of projects:

- GREETR (Next big thing in online greetings)**
 - inspection: #187.3 (Starting up IntelliJ IDEA (Mala)) [U-94.380...done.] - No artifacts, swtest (1) - 08 Mar 10 15:26 (13m.09s) [Run...]
 - #186.2 (Integrations total: 19, errors: 0) - No artifacts, swtest (1) - 08 Mar 10 15:26 (13m.09s) [Run...]
 - trunk: #1.1-SNAPSHOT r#7.19 (Tests failed: 1, passed: 5) - Artifacts, No changes - 08 Mar 10 16:43 (5s) [Run...]
- PROGNOSR (Cutting edge number prediction)**
 - documentation: #0.0.1-SNAPSHOT.9 (Success) - Artifacts, No changes - 08 Mar 10 16:43 (0%) [Run...]
 - tests (linux): #0.0.1-SNAPSHOT.3 (Tests passed: 11) - Artifacts, No changes - 08 Mar 10 16:38 (2%) [Run...]
 - tests (macos x): #0.0.1-SNAPSHOT.3 (Tests passed: 11) - Artifacts, No changes - 08 Mar 10 16:34 (2%) [Run...]
 - tests (windows): #0.0.1-SNAPSHOT.3 (Tests passed: 11) - Artifacts, No changes - 08 Mar 10 16:31 (2%) [Run...]
 - trunk: #0.0.1-SNAPSHOT.4 (Tests passed: 2) - No artifacts, No changes - 08 Mar 10 16:28 (2%) [Run...]
 - #0.0.1-SNAPSHOT.3 (Tests passed: 11) - No artifacts, No changes - 08 Mar 10 16:28 (2%) [Run...]

At the bottom of the page, there are links for Help, Send TeamCity Feedback, TeamCity Enterprise 5.1 EAP (build 13130), License agreement, and a small logo.

Abb. 5-18
Projektübersicht
in TeamCity

Architektur und besondere Merkmale

Java-Webapplikation

Wie alle Produkte, die wir bisher betrachtet haben, ist auch TeamCity als Java-Webapplikation realisiert. TeamCity kann in einen bestehenden J2EE-Container ausgebracht oder aber über den mitgelieferten Tomcat-Server betrieben werden. Die Betriebssysteme Windows, Mac OS X und Linux werden explizit unterstützt. TeamCity verwendet zur Datenspeicherung neben einem Datenverzeichnis auch eine relationale Datenbank. Für Testzwecke ist die mitgelieferte HSQLDB-Datenbank ausreichend, für Produktionseinsatz sollte aber besser eine externe Datenbank eingesetzt werden. Unterstützt werden hier zurzeit MySQL, PostgreSQL, Oracle, Microsoft SQL Server und Sybase.

Projekte und Build-Konfigurationen

TeamCity erlaubt die Gruppierung von sogenannten Build-Konfigurationen (dies entspricht den Jobs in Hudson bzw. den Build-Plänen in Bamboo) zu Projekten. Build-Konfigurationen eines Projektes teilen sich Pfade ins Versionsmanagementsystem und die Zuweisung von Benutzerrechten.

Verteiltes Bauen

Auch TeamCity arbeitet mit einer Master-Slave-Architektur, bei der ein zentraler Server Build-Aufträge an Agenten verteilt, die auf unterschiedlicher Hardware betrieben werden können. Zur Überwachung eines solchen *build grids* bietet TeamCity spezielle Ansichten, auf denen sich die Auslastung der einzelnen Agenten im Zeitverlauf anzeigen lassen. Zusätzlich ermittelt TeamCity für jeden Agenten einen Geschwindigkeitsindex. Eilige Builds lassen sich beim Start interaktiv besonders schnellen Knoten zuordnen. Für umfangreiche Builds bietet TeamCity eine Integration mit Amazons EC2 Cloud. Anders als bei Bamboo wird allerdings kein vorkonfigurierter Rechner in Form eines Amazon-EC2-Rechnerabilds (*Amazon Machine Image, AMI*) bereitgestellt. Dieses muss durch den Build-Manager vorbereitet und im Vorfeld auf den Amazon-Servern abgelegt werden. In der Praxis wird man dieses Merkmal aber sowieso nur mit individuell angepassten Rechnerabildern nutzen wollen, so dass dieser Umstand in langfristig betriebenen Installationen keinen wirklichen Nachteil darstellt.

Private Builds

TeamCitys wichtigste Alleinstellungsmerkmale dürften die privaten Builds (*private builds* bzw. *remote run*) und der vorgetestete Commit (*pre-tested commit*) sein. Private Builds erlauben es dem einzelnen Entwickler, seine Codeänderungen auf dem CI-Server probeweise zu bauen, und zwar ohne diese zuvor ins Versionsmanagement übernehmen zu müssen. Dies hat dreierlei Vorteile: Erstens wird dadurch der Arbeitsplatzrechner des Entwicklers entlastet, da der Build auf einer anderen Hardware ausgeführt wird. Der Entwickler kann also parallel seinen Rechner weiter nutzen. Zum zweiten werden die Änderungen in der Infrastruktur des CI-Servers getestet, die umfangreicher sein kann

als die eines Arbeitsplatzrechners (z.B. weitere Datenbanksysteme oder Applikationsserver). Drittens hat ein privater Build aus psychologischer Sicht einen unverbindlicheren Charakter, was zu häufigeren Builds und damit früherer Fehlererkennung animieren kann.

Technisch ist ein privater Build dadurch realisiert, dass vom Arbeitsplatzrechner nur die Codedifferenzen zum letzten versionierten Stand auf den CI-Server übertragen und dort vor dem Build auf eine ausgecheckte Kopie gepatcht werden. Dies erfordert ein gutes Zusammenspiel von IDE, Versionsmanagementsystem und CI-Server und erklärt, warum dieses Merkmal nur für ausgewählte Versionsmanagementsysteme (CVS, Subversion, Perforce) und ausgewählte IDEs (IntelliJ IDEA, Eclipse, Visual Studio) verfügbar ist. Mit TeamCity 5.1 besteht darüber hinaus erstmals die Möglichkeit, private Builds per Kommandozeile, also außerhalb der genannten IDEs, anzustoßen.

Vorgetestete Commits erweitern private Builds, indem Codeänderungen erst nach einem erfolgreichen Build automatisch ins Versionsmanagementsystem übernommen werden. Dadurch soll sichergestellt werden, dass nur fehlerfreier Code (im Sinne der automatischen Testmöglichkeiten) eingeholtet werden kann und es zu keinen Blockaden ganzer Arbeitsgruppen kommt, nur weil ein Kollege versehentlich unfertigen Code eingeholtet hatte (kurz vor Abreise in einen 14-tägigen Urlaub auf einer abgelegenen Berghütte – Sie kennen das).

Wie bereits erwähnt, ist von JetBrains als Hersteller einer bekannten Entwicklungsumgebung eine besonders enge Integration zwischen CI-Server und IDE zu erwarten. JetBrains berücksichtigt hier nicht nur das Produkt aus dem eigenen Hause, IntelliJ IDEA, sondern auch Eclipse und Visual Studio. Die Integration umfasst beispielsweise das Starten der bereits angesprochenen privaten Builds und vorgetesteten Commits, das Abfragen des Zustands überwachter Projekte oder das Verfolgen von Builds in Echtzeit. Darüber hinaus kann nicht nur von der IDE auf die Weboberfläche des CI-Servers navigiert werden. Auch umgekehrt kann in der Weboberfläche eine Quelltextdatei oder ein fehlgeschlagener JUnit-Test angewählt und dadurch die korrespondierende Stelle in der IDE geöffnet werden. Technisch wird dies übrigens durch einen schlanken Server ermöglicht, der in den IDE-Plugins eingebettet läuft und Kommandos vom CI-Server entgegennimmt.

Vorgetester Commit

IDE-Integration

Abb. 5-19

Ansicht eines Builds in
TeamCity

The screenshot shows the TeamCity build results for the project 'GREETER' in the 'trunk' branch. The build number is '#1.1-SNAPSHOT r87.17' and it was triggered on '06 Mar 10 16:34'. The build status is 'Failed' with 1 test failing. The code coverage summary shows 75% classes, 76.9% methods, and 76.2% lines. A detailed error message for the failed test 'GreeterTest.testComposeGreeting' is provided, indicating a comparison failure between expected and actual greeting outputs. The interface also shows 5 tests passed and 1 new test.

JUnit-Unterstützung

JUnit-Tests werden von TeamCity auf besondere Weise unterstützt: Zum einen werden JUnit-Tests bereits während des Builds in Echtzeit überwacht und ausgewertet. Der Entwickler kann somit bereits auf erste fehlgeschlagene Tests reagieren, während der Build noch läuft. Des Weiteren kann TeamCity die Ausführungsreihenfolge der JUnit-Tests beeinflussen und somit beispielsweise fehlgeschlagene Tests des vorausgegangenen Builds zuerst starten.

Zwei interessante Technologien, die JetBrains vom Produkt IDEA übernommen hat, sind zum einen die sogenannten Inspektionen, die mögliche Schwachstellen im Code aufgrund statischer Analyse finden sollen (vergleichbar zu Checkstyle, PMD und FindBugs). Zum anderen bringt TeamCity eine Funktion zur Ermittlung und Visualisierung der Codeabdeckung (*code coverage*) mit (vergleichbar zu Cobertura, EMMA oder Clover).

Das Rechte-Management ist sehr umfangreich implementiert, inklusive einer LDAP-Anbindung. Globale und projektbezogene Rechte lassen sich einzelnen Personen und Gruppen zuweisen. Manche Arbeitsgruppen würden sich jedoch die Möglichkeit wünschen, noch feiner, also auf Build-Konfigurationsebene, Rechte vergeben zu können.

Plugins

TeamCity lässt sich über Plugins erweitern. Zurzeit sind rund 25 Plugins verfügbar, davon etwa die Hälfte vom Hersteller JetBrains entwickelt und als Open Source gestiftet. Ähnlich wie bei Atlassians Bamboo scheint die Plugin-Szene keine maßgebliche Säule im Produktkonzept zu sein. Vielleicht sind aber auch die meisten Anwender mit dem Leistungsumfang »aus der Box« einfach zufrieden.

Rollen und Rechte

Plugins

JetBrains bietet zwei unterschiedliche Editionen an: Die kostenlose Professional-Edition ist auf 3 Build-Agenten, 20 Benutzer und 20 Build-Konfigurationen beschränkt. Außerdem steht nur die eingebaute Benutzerverwaltung zur Verfügung, die LDAP-Anbindung fehlt. Umfangreichere Installationen erfordern eine Enterprise-Lizenz (1720 EUR). Weitere Build-Agenten können für 258 EUR pro Agent hinzugekauft werden. Open-Source-Projekte können kostenlose Lizenzen beantragen.

Editionen

Vergleich mit Hudson

Mit privaten Builds und den vorgetesteten Commits verfügt TeamCity über ein klares Alleinstellungsmerkmal gegenüber anderen Mitbewerbern, auch gegenüber Hudson. In der Hudson-Entwicklergemeinde wird schon seit Längerem über die Implementierung eines vergleichbaren Features nachgedacht. Erste Ansätze unterstützen aber bisher nur verteilte Versionsmanagementsysteme wie Git. Hier hat JetBrains ganz klar die Nase vorn, indem es über Kompetenz für die komplette Kette von der IDE über das Versionsmanagement bis hin zum CI-Server verfügt.

Die IDE-Integration ist erwartungsgemäß sehr viel weitgehender als die der momentan verfügbaren Hudson-Plugins für Eclipse oder IntelliJ IDEA. Sieht man allerdings von den privaten Builds ab, haben viele Entwickler in der Praxis auch kein Problem damit, im ohnehin geöffneten Webbrowser die Oberfläche des CI-Servers parallel mitlaufen zu lassen.

TeamCitys Weboberfläche wirkt auf den ersten Blick nüchtern monochrom und »tabellenlastig«, erweist sich im Detail aber als gut durchdacht (Abb. 5–18, Abb. 5–19). Viel »Intelligenz« ist beispielsweise in dynamisch per AJAX aufgebauten Menüs eingebaut. Eine einfache Visualisierung der wichtigsten Werkzeuge wie Checkstyle, PMD oder FindBugs ist vorhanden. Momentan reicht diese im Umfang jedoch nicht an die entsprechenden Visualisierungsplugins heran, die für Hudson verfügbar sind.

Die Verwaltung der Agenten für verteilte Builds wird bei TeamCity durch sinnvolle und übersichtliche Visualisierungen unterstützt. Werkzeuge zum zentralen Ausbringen einer Build-Umgebung (JDK, Ant, Maven) wie bei Hudson fehlen aber. Es wird bei TeamCity vorausgesetzt, dass alle beteiligten Rechner zuvor durch einen Administrator eingerichtet wurden und der Build-Agent gestartet wurde.

Obwohl TeamCity durchaus ein Plugin-Konzept und sogar brauchbare Entwicklerdokumentation mitbringt, scheint die Plugin-Szene leider (noch) nicht recht in Schwung gekommen zu sein.

Fazit

TeamCity hat in der aktuellen Version einen Reifegrad erreicht, der für die meisten Mainstream-Umgebungen »aus der Box« sehr gut funktionieren sollte. Die kostenlose Professional-Edition ist im Leistungsumfang zwar eingeschränkt, ist aber deutlich mehr als nur eine »Demo-version«. Sie dürfte für kleinere Arbeitsgruppen durchaus produktiv einsetzbar sein.

Soll das CI-System hingegen mit den neuesten Werkzeugen oder proprietärer Infrastruktur kombiniert werden, bietet Hudson durch sein gelebtes Plugin-Konzept dafür zahlreiche einsatzbereite Erweiterungen. Es liefert so auch eine Fülle an Vorlagen für eigene Entwicklungen. Dies spricht überraschenderweise nicht nur notorisch bastelfreudige Informatikstudierende an. Vielmehr weckt dies auch das Interesse gerade großer Unternehmen: Hier wird Hudsons Offenheit geschätzt, wenn der Build-Prozess aus technischen, rechtlichen oder organisatorischen Gründen um individuelle Funktionalität erweitert werden muss. Hudson wird in diesem Umfeld weniger als Produkt mit festgelegtem Funktionsumfang verstanden, sondern als eine Plattform für Build-Automatisierung gesehen.

5.5 Zusammenfassung

In diesem Kapitel haben Sie die prinzipielle Funktionsweise von Hudson sowie ausgewählte Highlights kennengelernt. Darüber hinaus haben wir die prominenteren Alternativen beleuchtet, um den Überblick abzurunden.

Wenn Sie inzwischen Appetit auf den konkreten Einsatz von Hudson bekommen haben: Ausgezeichnet! Im nächsten Kapitel beginnen wir mit der Installation.

6 Installieren und Einrichten

- Wie wird Hudson installiert und eingerichtet?
- Wie und wo legt Hudson seine Daten ab?
- Wie werden Datensicherungen erstellt?
- Wie wird Hudson aktualisiert?

Nachdem Sie im vorausgegangenen Kapitel Hudson im Überblick kennengelernt haben, beginnen wir nun mit der konkreten Installation und Einrichtung Ihrer Hudson-Instanz.

Dies könnte eigentlich das kürzeste Kapitel dieses Buches sein. Im Vergleich zu anderen Serveranwendungen kann Hudson erfreulich schnell aufgesetzt werden – im einfachsten Fall sogar in unter 60 Sekunden, Herunterladen inklusive!

Installation in Sekunden

Da dies aber gleichzeitig der Beginn einer langen Freundschaft werden könnte, lohnt es sich, sich bereits bei der Installation Gedanken über die spätere Administration und Ausbaufähigkeit Ihrer Hudson-Instanz zu machen. Sie erhalten deshalb nach den sinnvollen »ersten Handgriffen« weiterführende Hinweise, wie Sie Ihre Hudson-Instanz für einen längerfristigen Betrieb optimal vorbereiten.

*Vorausschauende
Planung lohnt aber*

6.1 Schnellstart in 60 Sekunden

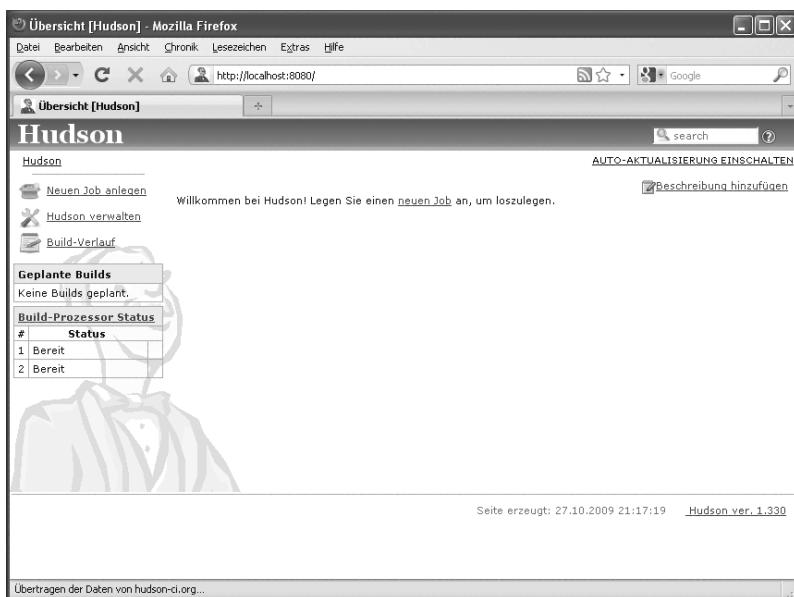
In nur drei einfachen Schritten bringen Sie Hudson auf Ihrem Rechner zum Laufen:

1. Laden Sie die neueste Version der Hudson-Webapplikation unter der URL <http://www.hudson-ci.org/latest/hudson.war> herunter.
2. Starten Sie die soeben heruntergeladene Webapplikation mit `java -jar hudson.war`. Beim ersten Start dauert es ein paar Sekunden, bis Hudson die WAR-Datei entpackt hat und der eingebettete Webcontainer Winstone gestartet ist. In der Konsole können Sie den Start des Webcontainers, das Anlegen des Hudson-Stammverzeichnisses und die Initialisierung mitgelieferter Plugins verfolgen.

3. Öffnen Sie die URL `http://localhost:8080` in einem Webbrowser. Sie sollten jetzt den Startbildschirm wie in Abbildung 6–1 sehen. Gratulation! Hudson, Ihr persönlicher Build-Butler, steht nun zu Ihren Diensten.

Abb. 6–1

*Hudsons Startbildschirm
nach erfolgreicher
Installation*



hudson.war: Webapplikation oder Java-Anwendung?

Vielleicht fragen Sie sich, wie eine WAR-Datei außerhalb eines Webcontainers ausgeführt werden kann? Müssen Webapplikationen nicht zunächst in einen Container wie z.B. Tomcat, JBoss oder WebSphere ausgebracht werden?

Die Datei `hudson.war` hat zwei Gesichter: Zum einen ist sie eine gültige Webapplikation, so wie es die Dateiendung impliziert. Sie benötigt also tatsächlich einen Container zur Ausführung.

Zum anderen ist die Datei jedoch so aufgebaut, dass sie auch ein gültiges Java-Archiv (JAR) mit Main-Klasse darstellt. Beim Aufruf über die Kommandozeile entpackt diese Main-Klasse das Archiv, startet einen darin eingebetteten Servlet-Container (Winstone) und bringt Hudson in diesen Webcontainer aus. Da Winstone nur 320 kB groß ist, kann Hudson problemlos seinen eigenen Servlet-Container mitbringen.

6.2 Systemkonfiguration

Direkt nach der Installation nehmen Sie typischerweise ein paar »erste Handgriffe« zur Konfiguration Ihrer Hudson-Instanz vor. Sie erreichen die Systemkonfiguration über *Hudson* → *Hudson verwalten* → *System konfigurieren*.

Im Abschnitt »JDK« fügen Sie die Versionen des Java Development Kits hinzu, die Sie später in Builds einsetzen möchten. Es handelt sich dabei *nicht* um das JDK, unter dem Hudson selbst ausgeführt wird. Spezifizieren Sie hier Ihre bereits lokal vorhandenen JDKs oder legen Sie eine automatische Installation an, die bei Bedarf über das Internet heruntergeladen und lokal installiert wird. Mehr Informationen zur Funktionsweise der automatischen Installation finden Sie in Kapitel 8 im Zusammenhang mit verteilten Builds.

*Java Development
Kits (JDK)*



Abb. 6–2
Konfiguration eines JDK

Der überwiegende Teil an Java-Projekten wird mit Ant oder Maven gebaut. Spezifizieren Sie im Abschnitt *Ant* bzw. *Maven* Ihre bereits installierten Ant- bzw. Maven-Versionen oder legen Sie eine automatische Installation an, die bei Bedarf über das Internet heruntergeladen und lokal installiert wird.

*Maven- oder Ant-
Installationen*



Abb. 6–3
*Konfiguration einer
Maven-Installation*

Hudson kann E-Mails verschicken, um die Anwender über den Ausgang neuer Builds zu informieren. Dazu geben Sie im Abschnitt »E-Mail-Benachrichtigung« Ihren SMTP-Server für ausgehende E-Mails an.

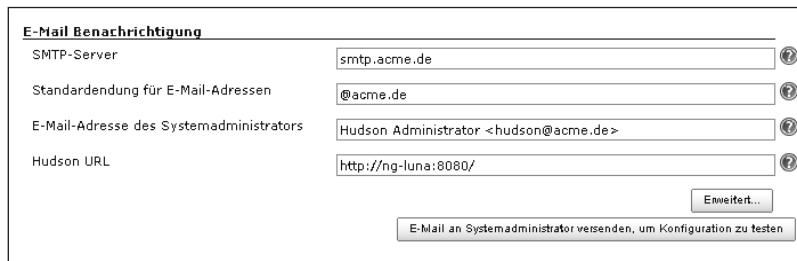
E-Mail-Parameter

Im einfachsten Fall bildet Hudson die E-Mail-Adresse eines Empfängers automatisch aus dessen Hudson-Kontonamen und der Standardendung für E-Mail-Adressen. Dies erspart Ihnen in den aller-

meisten Fällen eine explizite Auflistung von E-Mail-Adressen. Einzelne, davon abweichende Adressen können Sie in den jeweiligen Benutzerprofilen hinterlegen.

Die Hudson-URL wird benötigt, um in den verschickten E-Mails »anklickbare« Rückverweise auf Ihre Hudson-Instanz einzubauen.

Abb. 6-4
Konfiguration der E-Mail-Benachrichtigungen



Übernehmen Sie abschließend alle Ihre Angaben in dieser Bildschirmmaske. Mit der Spezifikation eines JDKs, eines Build-Verfahrens und den E-Mail-Parametern haben Sie die Minimalkonfiguration einer typischen Hudson-Instanz abgeschlossen.

Ungeduldige können nun in Kapitel 7 fortfahren und mit dem Anlegen eines ersten Projektes beginnen. Entscheiden Sie sich für den längerfristigen Einsatz von Hudson, sollten Sie aber auf jeden Fall zurückkehren und auch den folgenden Abschnitt 6.3 lesen.

6.3 Fortgeschrittene Installation

Bei der Installation auch an den dauerhaften Betrieb denken

Die Schnellstartmethode aus Abschnitt 6.1 und die Minimalkonfiguration aus Abschnitt 6.2 reichen aus, um eine produktionstaugliche Hudson-Instanz aufzusetzen. Im täglichen Einsatz ergeben sich allerdings mit der Zeit Fragen, bei denen zusätzliches Hintergrundwissen über Hudson hilfreich ist. Beispiele:

- Welche Systemvoraussetzungen benötigt Hudson?
- Welche Ablaufumgebung ist für Hudson erforderlich?
- Wohin speichert Hudson seine Einstellungen und Daten?
- Wie wird Hudson sauber gestoppt?
- Wie kann Hudson nach dem Einschalten eines Rechners automatisch wieder gestartet werden?
- Wie kann Hudson aktualisiert werden?
- Und im unwahrscheinlichen Fall: Wie wird Hudson deinstalliert?

Diese Fragen werden in verbleibenden Teil dieses Kapitels beantwortet.

6.3.1 Systemvoraussetzungen

Bei der Betrachtung der Systemvoraussetzungen ist zu unterscheiden,

- welche Ressourcen von der *Webanwendung Hudson* als Koordinator der Build-Prozesse benötigt werden und
- welche Ressourcen von den gestarteten *Build-Prozessen* in Anspruch genommen werden.

Die Koordinierung der Build-Prozesse durch Hudson stellt geringe Ansprüche an das darunterliegende Rechnersystem. Daher wird in vielen Arbeitsgruppen zunächst ein etwas betagter Rechner zur »CI-Maschine« auserkoren.

Die Durchführung der Builds hingegen ist in der Regel äußerst ressourcenintensiv. Ein modern ausgestatteter Rechner kann daher Build-Zeiten drastisch verkürzen. Der Einsatz aktueller Hardware stellt somit eine der schnellsten und unkompliziertesten Optimierungen der Build-Zeit dar.

Hudson benötigt zu seiner eigenen Ausführung lediglich das JRE 1.5. Für die Build-Prozesse, die Hudson anstößt, können jedoch abweichende JDKs – ältere wie neuere – eingesetzt werden.

Java Runtime Environment (JRE)

Hudson selbst kommt mit moderatem Speicherbedarf aus. Der genaue Wert variiert je nach Anzahl der Projekte und Builds, aber typischerweise genügen 128 MB. Der maximale Speicherbedarf eines Build-Prozesses kann hingegen deutlich größer sein, durchaus im GB-Bereich.

Hauptspeicher (RAM)

Die Webanwendung Hudson inklusive dem Winstone-Webcontainer benötigt ca. 30 MB Festplattenplatz. Je nach Anzahl der angelegten Projekte, der aufbewahrten Builds und der archivierten Artefakte sind jedoch Datenverzeichnisse mit mehreren GB keine Seltenheit.

Festplattenplatz

Der Rechner, auf dem Hudson betrieben wird, sollte eine sehr gute Netzwerkanbindung an verwendete Versionsmanagementsysteme (für schnelle Checkouts), Datenbankserver (für schnelle Tests) und ggf. Artefakt-Repositories (zur schnellen Archivierung von Build-Produkten) haben.

Netzwerkanbindung

Ein Zugang zum Internet ist zwar nicht zwingend erforderlich, aber Voraussetzung für

- die bequeme Installation von Plugins über den Plugin-Manager,
- Benachrichtigungen bezüglich neuer Versionen von Hudson und installierter Plugins,
- die automatische Installation von JDKs, Ant und Maven.

6.3.2 Ablaufumgebungen

Als Java-Webapplikation muss Hudson in einem Servlet-Container ausgebracht werden. Da Hudson weder Java Server Pages (JSPs) noch fortgeschrittene JEE-Funktionsmerkmale einsetzt, genügt hier ein ganz einfacher Servlet-Container – es muss also kein ausgewachsener Applikationsserver sein. In der Praxis findet man üblicherweise eines der folgenden zwei Szenarien vor:

- a) Hudson läuft als einzige Webapplikation innerhalb des eingebetteten Servlet-Containers Winstone.
- b) Hudson wird als eine von mehreren Webapplikationen in einen eigenständigen Servlet-Container (z.B. Jetty, Tomcat) oder gar Applikationsserver (z.B. Glassfish, JBoss, WebSphere) ausgebracht.

Welchem Szenario man folgen mag, ist weitestgehend von der bestehenden technischen und organisatorischen Infrastruktur abhängig. Die jeweiligen Vorteile werden im Folgenden kurz angesprochen:

Eingebetteter Container (Winstone)

Dieses Szenario haben Sie bereits beim Schnellstart in Abschnitt 6.1 kennengelernt. Vorteile dieses Szenarios:

- *Schneller Installationserfolg*
- Auch *für Produktiveinsatz völlig ausreichend* – selbst bei Hunderten von Projekten und mehreren tausend Builds
- *Geringer Ressourcenverbrauch* des Containers (Winstone ist nur äußerst schlanke 320 kB groß).
- *Schnelles Hoch- und Herunterfahren* des Containers im Sekundenbereich
- *Einfache Konfiguration* des Containers, da die Anzahl der Konfigurationsoptionen¹ überschaubar ist

Eigenständiger Servlet-Container bzw. Applikationsserver

Angesichts der vielen Vorteile des Winstone-Szenarios fragen Sie sich zu Recht, warum man Hudson in einem eigenständigem Servlet-Container oder gar Applikationsserver betreiben sollte. Mögliche Gründe:

- Das *Benutzerverzeichnis des Containers* soll von Hudson mitverwendet werden. Dies vermeidet die redundante Pflege von Benutzerkonten, wenn Hudson zugangsgeschützt betrieben werden soll.

1. Siehe <http://winstone.sourceforge.net>

- *Einfachere Administration* des Containers, da bereits erprobte Skripte zum automatischen Starten, Stoppen und Aktualisieren des Containers existieren
- *Bessere Auslastung* und Wiederverwendung bestehender Container (»Bitte nicht noch ein weiterer Server!«)
- Für die *Überwachung des Containers* existieren bereits technische und organisatorische Vorkehrungen, z.B. Alarmierung eines Administrators per SMS bei Stromausfall, Netzwerkproblemen, vollen Festplatten usw.
- *Organisatorische Aufteilung* zwischen störungsfreiem Betrieb des Containers (IT-Abteilung) und fachlicher Administration der Hudson-Instanz (Fachabteilung Softwareentwicklung)
- Der Container bietet zusätzliche *Sicherheitsfunktionen*, z.B. das Blockieren von Anfragen aus bestimmten IP-Adressbereichen oder eine besonders ausführliche Protokollierung der Serveraktivität.

6.3.3 Datei-Layout

Hudson speichert alle seine Daten als Dateien unterhalb seines Datenverzeichnisses <HUDSON_HOME>. Dies macht die Datenhaltung technisch einfach, robust und übersichtlich. Eine zusätzliche Datenbank wird nicht benötigt.

Als Vorgabewert für <HUDSON_HOME> wird das Verzeichnis .hudson im Stammverzeichnis des ausführenden Benutzers verwendet. Durch Setzen der Umgebungsvariable **HUDSON_HOME** vor dem Start des Servlet-Containers können Sie jedoch auch ein anderes Verzeichnis wählen.

```
# Hudson-Datenverzeichnis ändern  
export HUDSON_HOME=/data/hudson  
  
# Hudson wie gewohnt in Winstone-Container starten  
java -jar hudson.war
```

Konfigurationsdaten speichert Hudson im XML-Format ab. Obwohl Änderungen an der Konfiguration in der Regel über die Weboberfläche vorgenommen werden, können Sie diese auch direkt in den XML-Dateien eintragen. Dadurch lassen sich umfangreiche Änderungen wie etwa das Austauschen von URLs in mehreren hundert Jobkonfigurationen effizient mit der Suchen-und-Ersetzen-Funktion eines Texteditors durchführen. Das »Durchklicken« der Jobkonfigurationen im Browser bleibt Ihnen dadurch erspart.

Hudson übernimmt Änderungen in den XML-Dateien beim nächsten Neustart oder im laufenden Betrieb durch Ausführung der

Ein Verzeichnis enthält alle Hudson-Daten.

Listing 6–1
Beispiel für Start mit geändertem Datenverzeichnis (Linux)

Konfigurationen werden als XML gespeichert.

Funktion *Hudson* → *Hudson verwalten* → *Konfiguration von Festplatte neu laden*.

In den wenigsten Fällen werden Arbeiten direkt in <HUDSON_HOME> notwendig werden. Trotzdem lohnt es sich, den internen Aufbau zu kennen. So können Sie beispielsweise selektive Datensicherungen erstellen oder gezielt Konfigurationseinstellungen ändern. Tabelle 6–1 zeigt den Aufbau von <HUDSON_HOME> einer typischen Hudson-Instanz.

Tab. 6–1

Typischer Aufbau von
<HUDSON_HOME>
(Verzeichnisse sind
fettgedruckt)

Pfad	Inhalt
<HUDSON_HOME>	Stammverzeichnis aller Hudson-Daten
fingerprints	Hash-Codes der von Hudson erstellten Build-Artefakte
jobs	Jobs, die von Hudson verwaltet werden
musterJob	Job »musterJob«
builds	Archiv der durchgeführten Builds
modules	Maven-Untermodule (nur bei Maven-Projekten)
workspace	Arbeitsbereich, in dem der Job gebaut wird
config.xml	Konfiguration des Jobs »musterJob«
nextBuildNumber	Nummer des nächsten Builds von »musterJob«
[...]	Weitere Jobs
plugins	Installierte Erweiterungen
muster-plugin	Entpacktes Plugin »muster-plugin«
muster-plugin.hpi	Plugin in Distributionsformat (ZIP-Archiv)
[...]	Weitere Plugins
tools	Installationspaket des Autoinstallers, z.B. für JDKs, Maven- oder Ant-Versionen (existiert nur bei Bedarf)
updates	Metainformationen zu verfügbaren Plugins, JDKs, Maven- oder Ant-Versionen (für Autoinstaller)
users	Benutzer, die Hudson bekannt sind
mustermann	Benutzer »mustermann«
config.xml	Konfiguration des Benutzers »mustermann«
[...]	Weitere Benutzer
userContents	Statische Dateien, die per URL verfügbar sein sollen
war	Hudson Webapplikation (entpackt)

Pfad	Inhalt
config.xml	Globale Systemkonfiguration
hudson.*	Konfiguration globaler Komponenten und Plugins
nodeMonitors.xml	Konfiguration der Überwachung von Build-Knoten
secret.key	Zahlenschlüssel, der bei verteilten Builds zur gegenseitigen Identifizierung verwendet wird

6.3.4 Installationsmethoden

Neben der manuellen Installationsmethode, die Sie bereits in Abschnitt 6.1 kennengelernt haben, existieren für ausgewählte Betriebssysteme speziell vorbereitete Installationspakete. Diese Pakete werden dann vom Systemadministrator mit dem Paketmanager des Betriebssystems eingespielt. Beide Installationsmethoden werden im Folgenden verglichen.

Manuelle Installation

Diese Methode kennen Sie bereits aus Abschnitt 6.1. Vorteile:

- volle Transparenz hinsichtlich des Installationsvorgangs
- auf allen Betriebssystemen mit installiertem JRE möglich

Installation mit Paketmanager

Für ausgewählte Betriebssysteme, unter anderem Ubuntu, Fedora, FreeBSD oder Solaris, kann Hudson auch über einen Paketmanager installiert werden. Welche Vorteile hat dieser Weg gegenüber der schnellen, schlanken manuellen Installationsmethode?

Ein Paketmanager berücksichtigt die Konventionen des jeweiligen Betriebssystems hinsichtlich der Integration von Serverdiensten. Dies umfasst beispielsweise:

- Anlegen eines eigenständigen Benutzers, unter dem der Dienst ausgeführt wird,
- Anlegen der Protokolldateien in einem üblichen Verzeichnis zur zentralen Überwachung, z.B. als /var/log/hudson/hudson.log unter Ubuntu,
- Anlegen von Skripten zum Starten und Stoppen des Dienstes, z.B. bei Neustart des Rechners, sowie
- automatische Überprüfung auf Aktualisierungen durch den Paketmanager.

Der Einsatz eines Paketmanagers erlaubt zusätzlich auch eine organisatorische Trennung zwischen dem Betreiber des Rechners (durch Administratoren der IT-Abteilung) und dem Nutzer der Dienste (Fachabteilungen). Administratoren können so auf gewohnte Weise Updates einspielen, ohne sich zu intensiv mit Hudson-Spezifika auseinandersetzen zu müssen.

Im Folgenden wird beispielhaft die Installation von Hudson unter Ubuntu dargestellt. Sie benötigen dazu Administratoren-Rechte.

Zunächst laden Sie den Serverschlüssel des Hudson-Projekts herunter. Dadurch werden Installationspaketes der Hudson-Website vom Paketmanager als vertrauenswürdig angesehen:

```
wget -O - http://hudson-ci.org/debian/hudson-ci.org.key |  
apt-key add -
```

Fügen Sie nun die Hudson-Website als Quelle für Installationspaketes hinzu:

```
echo "deb http://hudson-ci.org/debian binary/" >>  
/etc/apt/sources.list
```

Aktualisieren Sie Ihren lokalen Paketindex und installieren Sie abschließend das Paket »Hudson«:

```
sudo apt-get update  
sudo apt-get install hudson
```

Nach Abschluss einer erfolgreichen Installation erreichen Sie Hudson auf Ihrem Rechner unter der URL <http://localhost:8080>.

6.3.5 Starten und Stoppen

6.3.5.1 Starten

Verwenden Sie den eingebetteten Servlet-Container Winstone, so können Sie Hudson manuell starten, wie Sie es bereits in Abschnitt 6.1 kennengelernt haben. In allen anderen Fällen startet Hudson als Webapplikation zusammen mit dem umgebenden Container.

In der Regel möchten Sie jedoch die Verfügbarkeit von Hudson rund um die Uhr sicherstellen und nicht nach jedem Rechnerneustart daran denken müssen, auch Hudson neu zu starten.

Unter Linux integrieren Sie daher die benötigten Start-Skripte Ihres Servlet-Containers in die /etc/init.d-Verzeichnisstruktur. Bei einer Installation über einen Paketmanager geschieht dies automatisch.

Unter Windows können Sie Hudson als Dienst (*service*) betreiben. Hudson bringt dazu eigens eine Funktion zum Einrichten als Dienst

*Automatischer Start
unter Linux*

*Automatischer Start als
Windows-Dienst*

mit: Unter *Hudson* → *Hudson verwalten* → *Als Windows-Dienst installieren* geben Sie lediglich das gewünschte <HUDSON_HOME>-Verzeichnis Ihrer Installation an (oder behalten das existierende bei) und überlassen Hudson die weiteren Details der Einrichtung.

Ändern Sie hier den Pfad zu <HUDSON_HOME>, werden die Daten aus dem bisherigen <HUDSON_HOME>-Verzeichnis *nicht* automatisch in das neue übernommen. Am einfachsten ist es in diesem Fall, das bestehende Verzeichnis *vor* Einrichtung des Dienstes an die gewünschte Stelle zu kopieren und nach Einrichtung das alte Verzeichnis zu entfernen.

War die Einrichtung als Dienst erfolgreich, beendet Hudson die laufende Hudson-Instanz (also sich selbst!), startet den Windows-Dienst und leitet den Browser automatisch auf die neu gestartete Instanz. Hudson wird ab jetzt bei jedem Hochfahren des Rechners automatisch gestartet.

Beachten Sie, dass der Windows-Dienst unter dem lokalen Systemkonto ausgeführt wird. Einige Build-Werkzeuge, wie zum Beispiel Maven, legen standardmäßig ihre Daten in das Stammverzeichnis des ausführenden Benutzers ab. Ein eigenes Benutzerkonto für Hudson erleichtert daher die Verwaltung Ihres Systems. Verwenden Sie ein abweichendes Benutzerkonto, so können Sie dieses in den Windows-Systemeinstellungen unter *Start* → *Systemeinstellungen* → *Verwaltung* → *Dienste* für den Dienst »hudson« angeben.

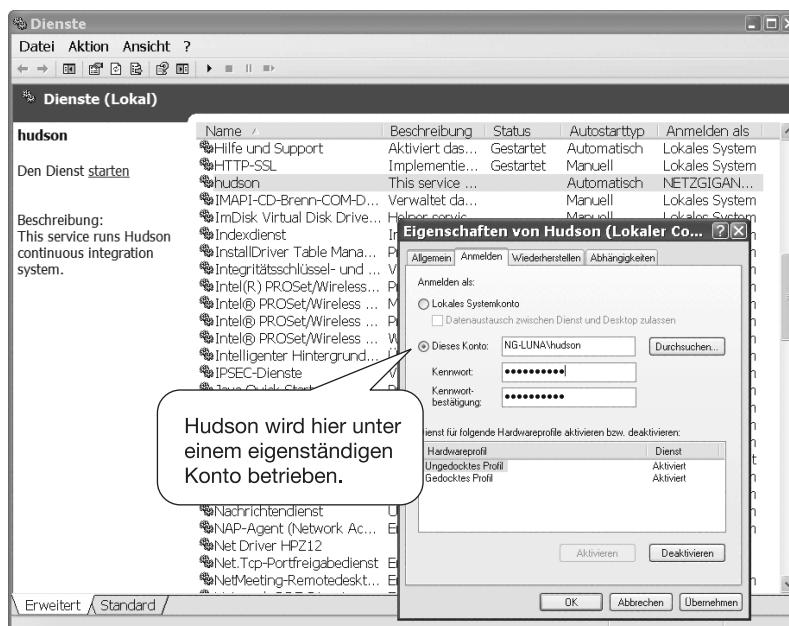


Abb. 6-5
*Hudson als
Windows-Dienst*

Die Installation per Weboberfläche kann wegen fehlender Berechtigungen fehlschlagen. In diesem Falle führen Sie den Hudson-Dienst-Installer mit Windows-Administratorenrechten in der Kommandozeile direkt in <HUDSON_HOME> aus:

```
D:\hudson>hudson.exe install
```

Beachten Sie, dass <HUDSON_HOME>\hudson.exe erst angelegt wird, wenn Sie mindestens einmal eine Installation als Dienst versucht haben.

In der Datei <HUDSON_HOME>\hudson.xml können Sie weitere Anpassungen des Dienstes vornehmen. Wichtig: Achten Sie darauf, dass das Element <arguments> vollständig in einer Zeile stehen muss, sonst schlägt der Start des Dienstes fehl.

Im folgenden Beispiel ändern wir HUDSON_HOME auf H:\HUDSON, vergrößern die Speicherzuordnung auf 320 MB und verschieben den Zugang zur Weboberfläche auf Port 8888:

Listing 6-2

Konfiguration des
Windows-Dienstes in
hudson.xml

```
<service>
  <id>hudson</id>
  <name>Hudson</name>
  <description>
    This service runs Hudson continuous integration system.
  </description>
  <env name="HUDSON_HOME" value="H:\HUDSON"/>
  <executable>java</executable>
  <arguments>
    -Xrs -Xmx320m -jar "%BASE%\hudson.war" --httpPort=8888
    -Dhudson.lifecycle=hudson.lifecycle.WindowsServiceLifecycle
  </arguments>
  <logmode>rotate</logmode>
</service>
```

Um den Dienst wieder zu entfernen, führen Sie erneut den Hudson-Dienst-Installer mit Windows-Administratorenrechten in der Kommandozeile direkt in <HUDSON_HOME> aus, dieses Mal aber mit der Option `uninstall`:

```
D:\hudson>hudson.exe uninstall
```

Die momentan noch laufende Instanz wird dabei übrigens *nicht* beendet. Beim nächsten Neustart des Rechners wird Hudson nun nicht mehr automatisch gestartet. Alternativ könnten Sie auch `sc`, das von Windows mitgelieferte Kommando zur Verwaltung von Diensten, verwenden:

```
D:\hudson>sc delete hudson
```

6.3.5.2 Stoppen

Sie beenden Hudson, indem Sie den umschließenden Container beenden. Verwenden Sie Winstone, so reicht dazu die Eingabe von Strg+C in der Kommandozeile.

Idealerweise sollten zu diesem Zeitpunkt keine Builds mehr laufen, da deren Ergebnisse dann nicht mehr protokolliert und archiviert würden. Unter *Hudson → Hudson verwalten → Herunterfahren vorbereiten* können Sie die Ausführung neuer Builds verhindern und so eine »leer laufende« Hudson-Instanz sicherstellen.

6.3.6 Aktualisieren

Im Gegensatz zu den meisten anderen Softwareprojekten veröffentlicht das Hudson-Team im Schnitt einmal pro Woche(!) eine neue Version. Dies hat den Vorteil, dass Fehlerkorrekturen schneller ausgeliefert werden können und Funktionsinkremeente kleiner und überschaubarer ausfallen. Die Versionsnummer wird dabei jeweils um eins erhöht, ausgehend von 1.100. Der Version 1.362 sind also beispielsweise bereits 362 veröffentlichte Versionen vorausgegangen.

Auf neue Versionen werden Sie bei bestehender Internetverbindung automatisch unter *Hudson → Hudson verwalten* hingewiesen.

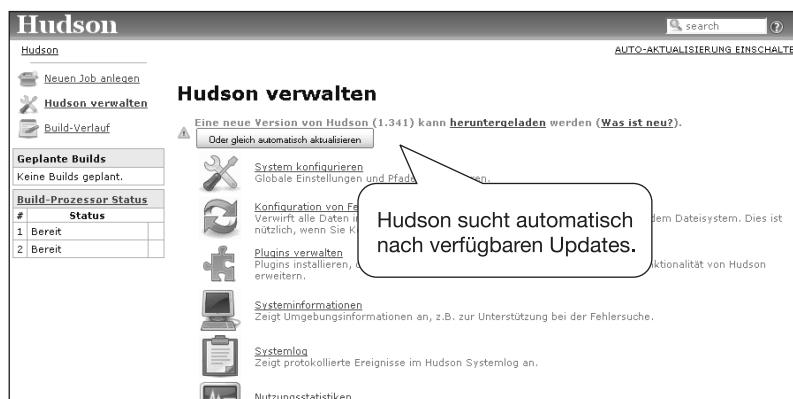


Abb. 6-6
Automatische Suche
nach Updates

So aktualisieren Sie Ihre Hudson-Instanz:

1. Stoppen Sie Hudson (siehe Abschnitt 6.3.5.2).
2. Sichern Sie <HUDSON_HOME> (nur für den Fall der Fälle).
3. Laden Sie die neue Version der Datei hudson.war herunter und überschreiben Sie damit die alte.
4. Starten Sie Hudson neu.

5. Überprüfen Sie, ob die Aktualisierung erfolgreich war, indem Sie am unteren rechten Seitenrand der Weboberfläche die Versionsnummer ablesen.

Neben der Webanwendung sollten Sie auch installierte Plugins aktuell halten. Zur Aktualisierung von Plugins verwenden Sie den Plugin-Manager unter *Hudson* → *Hudson verwalten* → *Plugins verwalten*.

Die Aktualisierung der Webanwendung und der installierten Plugins ist eine der wenigen Situationen, die einen Neustart von Hudson erfordern.

6.3.7 Backups

Sicherungen erstellen

Da Hudson alle seine Daten unterhalb von <HUDSON_HOME> ablegt, können Sie Backups ganz einfach durch Kopieren dieses Verzeichnisses anlegen. Sie müssen Hudson dazu nicht zwingend beenden, solange Sie darauf achten, dass während des Kopievorgangs keine Builds in Bearbeitung sind.

- | | |
|-------------------------------------|--|
| <i>Sicherung im Wartungsfenster</i> | So erstellen Sie eine vollständige Sicherung Ihrer Hudson-Instanz: |
| | <ol style="list-style-type: none">1. Öffnen Sie ein Wartungsfenster mit der Funktion <i>Hudson</i> → <i>Hudson verwalten</i> → <i>Herunterfahren vorbereiten</i>. Hudson führt dann bereits laufende Builds zu Ende, startet aber keine neuen mehr. Zusätzlich wird allen Benutzern in der Weboberfläche der Hinweis »Hudson wird heruntergefahren« angezeigt – was Rückfragen beim Administrator erspart (»Warum läuft mein Build nicht?«).2. Sind alle Builds beendet und ist Hudson im Leerlauf, kopieren Sie <HUDSON_HOME> auf Ihr Sicherungsmedium.3. Nach Abschluss der Sicherung beenden Sie das Wartungsfenster mit <i>Hudson</i> → <i>Hudson verwalten</i> → <i>Herunterfahren abbrechen</i> und führen Hudson wieder in den normalen Betriebsmodus zurück. |

- | | |
|------------------------------|--|
| <i>Voll-/Teilsicherungen</i> | Je nach Umfang Ihrer Projekte kann eine vollständige Sicherung von <HUDSON_HOME> etliche GB umfassen und mehrere Minuten oder gar Stunden benötigen. Dauert Ihnen das zu lange, können Sie eine verfeinerte Backup-Strategie einsetzen, bei der Sie nur ausgewählte Teile aus <HUDSON_HOME> sichern. Typische Strategien sind: |
|------------------------------|--|

■ *Vollsicherung (VOLL):*

Kompletter Schnapschuss des <HUDSON_HOME>-Verzeichnisses. Am einfachsten zu erstellen und garantiert vollständig, dauert aber auch am längsten.

■ *Server-Konfiguration (SK):*

Nur die globalen Konfigurationen und insbesondere die installierten Plugins werden gesichert. Geht am schnellsten, aber enthält keine Dateien aus den angelegten Projekten.

■ *Server- und Projektkonfigurationen (SPK):*

Neben den globalen Konfigurationen werden auch die Konfigurationen aller Projekte gesichert. Sichert in Sekundenschnelle mühsam zu rekonstruierende Einstellungen, enthält aber keine historischen Daten der einzelnen Projekte.

■ *Server- und Projektkonfigurationen, Projektverläufe und Artefakte (SPKVA):*

Sehr ähnlich zur Vollsicherung, beinhaltet aber nicht die Arbeitsbereiche der Projekte. Dies ist selten ein Nachteil, da in der Regel der Inhalt eines Arbeitsbereichs durch Ausführen eines neuen Builds wiederhergestellt werden kann.

■ *Einzelnes Projekt (EP):*

Alle Daten, die zu einem Projekt gehören, mit Ausnahme des Arbeitsbereichs. Diese Sicherung verwenden Sie typischerweise, um ein abgeschlossenes Projekt archivieren.

Tabelle 6–2 zeigt einen Vergleich der genannten Backup-Strategien.

Daten	VOLL	SK	SPK	SPKVA	EP
Globale Konfiguration inkl. Plugins	✓	✓	✓	✓	
Projektkonfigurationen	✓		✓	✓	✓
Projektverläufe	✓			✓	✓
Projektartefakte	✓			✓	✓
Projektarbeitsbereiche	✓				

Tab. 6–2
Überblick über typische
Backup-Strategien

Die vorgestellten Backup-Strategien lassen sich direkt in der Kommandozeile umsetzen. Das folgende Beispiel zeigt eine Realisierung der Strategie SPK unter Linux und kann als Anregung für individuelle Strategien dienen: Ausgewählte Dateien und Verzeichnisse werden mithilfe des Kommandos cpio in ein handliches Archiv zusammengefasst und anschließend per gzip komprimiert. Der Dateiname des Archivs beinhaltet einen Zeitstempel, so dass sich die Sicherung später mühelos einem Datum zuordnen lässt.

Listing 6-3

backup.sh – Skript zur Erstellung einer Teilsicherung (Linux)

```
#!/bin/bash
cd $HUDSON_HOME

(
    # include global server configuration files
    ls *.xml secret.key
    # include global server directories
    find bin plugins users fingerprints userContent
    # include individual jobs
    find jobs -name workspace -prune \
        -o -name builds -prune \
        -o -print
) | cpio -ocv | gzip > /backup/hudson/`date +%Y%m%d_%H%M`.gz
```

Plugin »Backup«

Ist die Kommandozeile nicht Ihre Welt, so können Sie auch sehr komfortabel Backups mit dem Plugin »Backup« erstellen, der für Sie

1. das Einschalten eines Wartungsfensters übernimmt,
2. auswählbare Teile des Datenbestandes sichert,
3. in einem komprimierten Archiv ablegt und
4. danach das Wartungsfenster wieder aufhebt.

Durch regelmäßiges Aufrufen der URL <http://<ihr.hudson>/backup/launchBackup> können Sie Sicherungen ganz einfach automatisieren (z.B. per cron).

Abb. 6-7*Plugin »Backup«***Sicherungen wiederherstellen**

Die Wiederherstellung einer Sicherung gestaltet sich ähnlich einfach wie das Anlegen: Sie stoppen Hudson, kopieren die gewünschten Dateien an den ursprünglichen Platz zurück und starten Hudson danach wieder.

Haben Sie Teilsicherungen auf Basis des oben gezeigten Kommandos für Linux erstellt, können Sie Ihre Sicherung beispielsweise so wiederherstellen:

```
cd $HUDSON_HOME
zcat /backups/20100418_1704.gz | cpio -idmu
```

Listing 6-4

*Wiederherstellen einer
Teilsicherung unter Linux*

Abschließend führen Sie *Hudson* → *Hudson verwalten* → *Konfiguration von Festplatte neu laden* aus oder starten Hudson einfach neu.

Das Plugin »Shelve Project« (zu deutsch etwa: Projekt ins Regal stellen) erlaubt es ebenfalls, auf einfache Weise ein komplettes Projekt »einzufrieren«, temporär aus der Liste der aktiven Projekte zu entfernen und später bei Bedarf wieder verlustfrei »aufzutauen«.

6.3.8 Deinstallation

Die Deinstallation ist denkbar einfach, da Hudson alle seine Daten unterhalb von <HUDSON_HOME> ablegt. Es genügt, Hudson zu beenden und anschließend das <HUDSON_HOME>-Verzeichnis zu löschen.

Haben Sie Hudson mit einem Paketmanager installiert, sollten Sie diesen auch zur Deinstallation verwenden. Nur so werden auch zusätzliche Betriebssystemintegrationen restlos entfernt, zum Beispiel Start-/Stopp-Skripte. Die Einrichtung als Windows-Dienst machen Sie wie in Abschnitt 6.3.5.1 beschrieben rückgängig.

6.3.9 Plugins

Plugins installieren Sie am einfachsten über den eingebauten Plugin-Manager, den Sie unter *Hudson* → *Hudson verwalten* → *Plugins verwalten* erreichen. Bei bestehender Internetverbindung können Sie aus allen im Hudson-Update-Center angebotenen Plugins (zurzeit über 250) auswählen und das Herunterladen und Installieren Hudson überlassen. Zur Aktivierung der frisch installierten Plugins ist allerdings ein Neustart des Hudson-Servers notwendig. Besteht kein Zugang ins Internet, können Sie die Plugins einzeln als HPI-Dateien unter <http://hudson-ci.org/download/plugins> herunterladen und im Plugin-Manager unter dem Karteireiter *Erweiterte Einstellungen* manuell in den Server hochladen. Auch hier ist ein Server-Neustart vonnöten, nachdem Sie alle neuen Plugins hochgeladen haben.

Plugins installieren

Möchten Sie einen Plugin nur übergangsweise deaktivieren, aber nicht komplett aus Ihrem Hudson-Server entfernen, wählen Sie im Plugin-Manager unter dem Karteireiter *Installiert* das entsprechende Häkchen ab und – Sie ahnen es bereits – starten den Server neu. Beim Neustart kann es dann sein, dass umfangreiche Warnungen in der

Plugins deaktivieren

Konsole ausgegeben werden. Das liegt daran, dass Konfigurationsdateien eventuell Einstellungen für die soeben deaktivierten Plugins enthalten und Hudson diese daher nicht berücksichtigen kann. Da Hudson diese Einstellungen beim Lesen einfach ignoriert, stellt dies in der Regel kein Problem dar.

Plugins deinstallieren

Möchten Sie ein Plugin dauerhaft deinstallieren, beenden Sie zunächst den Hudson-Server und löschen dann die entsprechende HPI-Datei und das gleichnamige Plugin-Unterverzeichnis im Verzeichnis <HUDSON_HOME>/plugins. Auch in diesem Fall wird Hudson beim Neustart vermutlich zahlreiche Warnungen ausgeben, weil er mit den »verwaisten« Konfigurationseinstellungen des deinstallierten Plugins nichts anfangen kann. Sobald Sie jedoch eine Konfiguration (des Servers, eines Projekts, eines Benutzers, einer Ansicht usw.) in der Oberfläche verändern und erneut abspeichern, werden die »verwaisten« Bestandteile entfernt und so die Altlasten in Ihren Dateien entsorgt.

Wenn Sie nicht so lange warten möchten und stattdessen lieber alle Konfigurationsdateien auf einen Rutsch bereinigen wollen, bietet Hudson Ihnen dazu unter *Hudson → Hudson verwalten* eine passende Funktion *Verwalten* im Kontext der Warnmeldung *Es liegen Daten in einem veralteten Format und/oder nicht-lesbare Daten vor*. Diese Meldung ist nur sichtbar, wenn beim Starten des Servers veraltete bzw. nicht lesbare (also in der Regel »verwaiste«) Daten vorgefunden wurden.

Abb. 6-8

Warnmeldung als Hinweis
auf »verwaiste«
Konfigurationsdaten



6.3.10 Tipps aus der Praxis

Leerzeichen vermeiden

Vermeiden Sie Leerzeichen in Projekt- und Verzeichnisnamen, vor allem im Pfad zu <HUDSON_HOME>. Hudson selbst hat damit zwar keine Probleme, aber Ihr Build-Prozess setzt vielleicht Skripte und Werkzeuge ein, die nicht mit Leerzeichen in Verzeichnisnamen umgehen können.

Alias-Namen verwenden

Legen Sie einen Alias-Namen für den Rechner an, auf dem Sie Hudson betreiben. Ihre Hudson-URLs werden dadurch sprechender, aus <http://pc0293.acme.de> wird also besser <http://hudson.acme.de>.

Sollte Ihre Hudson-Instanz in der Zukunft auf einen anderen Rechner umziehen, müssen Sie nur den Alias-Namen umleiten. Alle existierenden Verweise, etwa in E-Mails und Fehlertickets, bleiben gültig.

Machen Sie Hudson über den HTTP-Standardport 80 erreichbar. Zum einen kürzt dies alle Verweise auf Ihre Hudson-Instanz und macht diese übersichtlicher. Zum anderen vermeidet es Ärger mit restriktiven Firewalls.

Port 80 verwenden

Je nach Startweise legen Sie den Port unterschiedlich fest. Im einfachsten Fall starten Sie Hudson mit einem zusätzlichen Parameter:

```
java -jar hudson.war --httpPort=80
```

Aus Sicherheitsgründen sollten Sie Hudson nie als Administrator betreiben, sondern dazu ein eigens angelegtes Benutzerkonto mit eingeschränkten Zugriffsrechten verwenden. Da unter Linux nur Administratoren Prozesse starten können, die auf Ports unterhalb von 1024 erreichbar sind, wird oft auf folgende zwei bewährte Lösungen für dieses Problem zurückgegriffen:

- Vorschalten eines Apache HTTP-Servers auf Port 80, der intern an einen Hudson-Server auf einem anderen Port (oberhalb von 1024) weitervermittelt²
- Einrichten eines Port-Mappings mittels iptables³. Diese Technik erlaubt es, die auf Port 80 eingehenden, externen Anfragen auf einen beliebigen, internen Port weiterzuleiten. Diese Art der Umleitung bleibt völlig transparent für den anfragenden Webbrowser.

Die Willkommensmeldung wird auf der Startseite angezeigt und ist optional, aber empfehlenswert. Hier lassen sich auf sehr einfache Weise praktische Informationen wie etwa die Kontaktinformationen des Administrators unterbringen.

Willkommensmeldung

Da HTML-Markup erlaubt ist, können Sie Formatierungen vornehmen und Verweise auf andere Dienste integrieren wie etwa Issue-Management, Wikis oder Sharepoint-Server. Betreiben Sie mehrere Hudson-Instanzen, vermeidet eine individuelle Willkommensmeldung zudem Verwechslungen.

Tipp: Um Logos, Dokumentationen oder andere statische Dokumente einzubinden, legen Sie diese unter <HUDSON_HOME>/userContent ab und referenzieren diese im HTML-Fragment. So wurde auch das Bauarbeiter-Piktogramm in Abbildung 6–9 realisiert. Den zugehörigen Quelltext finden Sie in Abbildung 6–10.

2. <http://wiki.hudson-ci.org/display/HUDSON/Running+Hudson+behind+Apache>

3. <http://netfilter.org/projects/iptables>

Abb. 6-9

Beispiel einer
Willkommensmeldung

**Abb. 6-10**

Konfiguration der
abgebildeten
Willkommensmeldung



Die Willkommensmeldung können Sie entweder in der Systemkonfiguration setzen (*Hudson → Hudson verwalten → System konfigurieren*) oder Sie klicken den Verweis »Beschreibung bearbeiten« auf der Startseite rechts oben an.

6.4 Zusammenfassung

In diesem Kapitel haben Sie gelernt, wie Sie Hudson installieren und sicher betreiben. Sie wissen nun, wie Sie Backups Ihrer Konfigurationen und Projekte erstellen, Hudson aktualisieren und – im unwahrscheinlichen Fall – auch wieder deinstallieren können.

Im nächsten Kapitel werden wir ein erstes Projekt in Hudson anlegen und lassen die Integrationen beginnen.

7 Hudson im täglichen Einsatz

- Wie werden Jobs in Hudson angelegt?
- Wie wertet Hudson Testberichte automatisch aus?
- Wie informiert Hudson über seinen aktuellen Status?
- Wie kann Hudson die Codequalität überwachen?
- Wie werden Versionsmanagementsysteme und Issue-Tracker angebunden?

Die alltägliche Nutzung eines CI-Systems lässt sich grob in zwei Fälle unterscheiden:

- Im ersten Fall, dem vollautomatischen Betrieb, wird unbeaufsichtigt Build für Build durchgeführt. Zwar werden alle Ergebnisse fein säuberlich archiviert, aber der Benutzer möchte eigentlich nur bei Problemen benachrichtigt werden. Hier gilt: »Keine Nachrichten sind gute Nachrichten.«
- Im zweiten Fall hingegen wird die Weboberfläche des CI-Systems interaktiv genutzt, etwa für Ad-hoc-Abfragen oder in Retrospektiven (einer speziellen Art der »Manöverkritik« für einen Projektabschnitt). Diese Nutzungsweise steht und fällt mit guten Visualisierungs- und Auswertungsmöglichkeiten.

Für beide Fälle werden Sie in diesem Kapitel Anregungen finden, wie Sie Ihr Hudson-System zur Schaltzentrale Ihres Entwicklungsteams ausbauen können.

Zunächst werden wir im Schnelldurchlauf ein Softwareprojekt mit Hudson automatisiert bauen lassen. Sie lernen dabei die essenziellen Schritte kennen, die notwendig sind, um Ihre eigenen Projekte als Jobs in Hudson anzulegen.

Im restlichen Teil des Kapitels werden wir – ausgehend von der Minimalkonfiguration des Schnelldurchlaufs – Erweiterungen betrachten. Insbesondere werden Sie empfehlenswerte Hudson-Plugins kennenlernen, mit denen Sie Ihr persönliches Build-System aufrüsten können.

7.1 Ihr erster Hudson-Job im Schnelldurchlauf

Nachdem Sie Ihr Hudson-System wie in Kapitel 6 beschrieben eingerichtet haben, legen wir nun einen ersten Job in Hudson an.

Stellen Sie sich dazu vor, Hudson wäre ein neuer Kollege in Ihrem Team. Welche Angaben würde dieser neue Kollege von Ihnen benötigen, damit er auf seinem Arbeitsplatz erfolgreich den letzten Stand Ihres Entwicklungsprojekts bauen könnte? Unabdingbar wären sicherlich:

- Name des Projekts
- Pfad zu den Quelltexten im Versionsmanagementsystem
- Build-Werkzeug (Ant, Maven, Skripte etc.)
- werkzeugabhängige Einstellungen, wie etwa auszuführende Targets (Ant) oder Goals (Maven)

Genau diese Angaben sind auch notwendig, um ein Projekt in Hudson anzulegen.

7.1.1 Den ersten Job einrichten

Tipp: Testweise ein Open-Source-Projekt bauen

Falls Sie gerade kein eigenes Projekt zur Hand haben, aber Zugang zum Internet, so bauen Sie doch einfach testweise ein Open-Source-Projekt. Für ein Ant-basiertes Projekt bietet sich beispielsweise Apache Ivy an:

Tragen Sie dazu in der Projektkonfiguration unter *Source-Code-Management* → *Subversion* → *Module* → *Repository URL* die Adresse <http://svn.apache.org/repos/asf/ant/ivy/core/trunk> ein. Unter Build-Verfahren fügen Sie lediglich einen Build-Schritt *Ant aufrufen* hinzu und geben unter *Ant Build Datei* *trunk/build.xml* ein. Alle weiteren Felder belassen Sie leer. Dadurch greifen Vorgabewerte, welche das Default-Target in der Datei <Arbeitsbereich>/trunk/build.xml ausführen.

Der erste Build wird mehrere Minuten dauern, da nicht nur die Quelltexte aus Subversion ausgecheckt werden müssen, sondern auch Drittbibliotheken heruntergeladen werden. Die weiteren Builds hingegen werden deutlich schneller sein, da heruntergeladene Dateien aus vorhergehenden Builds wiederverwendet werden.

In unserem Beispiel liegen die Quelltexte in einem Subversion-Server und werden mit Ant gebaut. So legen Sie Ihren ersten Job in Hudson an:

1. Wählen Sie auf der Hudson-Startseite *Neuen Job anlegen* aus. Sie gelangen auf eine Maske zum Anlegen neuer Jobs (Abb. 7–1).

2. Geben Sie einen Namen für Ihr Projekt ein. Tipp: Vermeiden Sie hier Leerzeichen und Umlaute – der Projektname wird später Teil von Dateipfaden, und manche Werkzeuge könnten damit Probleme haben.

Job Name

Externen Job überwachen

Dieses Profil erlaubt die Überwachung von Prozessen, die außerhalb von Hudson ausgeführt werden - eventuell sogar auf einem anderen Rechner! Dadurch können Sie Hudson ganz allgemein zur zentralen Protokollierung von automatisiert ausgeführten Prozessen einsetzen. [Mehr...](#)

"Free Style"-Softwareprojekt bauen

Dieses Profil ist das meistgenutzte in Hudson. Hudson baut Ihr Projekt, wobei Sie universell jedes SCM System mit jedem Build-Verfahren kombinieren können. Dieses Profil ist nicht nur auf das Bauen von Software beschränkt, sondern kann darüber hinaus auch für weitere Anwendungsbereiche verwendet werden.

Multikonfigurationsprojekt bauen (alpha)

Dieses Profil eignet sich sehr gut für Projekte mit zahlreichen Konfigurationen, die etwa in unterschiedlichen Umgebungen getestet oder plattformspezifisch gebaut werden müssen.

Maven 2 Projekt bauen

Dieses Profil baut ein Maven 2 Projekt. Hudson wertet dabei Ihre POM Dateien aus und reduziert damit den Konfigurationsaufwand ganz erheblich. Wahr befindet sich dieses Profil zur Zeit noch in der Entstehungsphase, es ist aber bereits einsetzbar, um Rückmeldungen von Anwendern zu sammeln.

Kopiere bestehenden Job

Kopiere von:

Abb. 7-1
Anlegen eines neuen Jobs

3. Wählen Sie *Free Style-Softwareprojekt bauen* aus und schließen Sie mit *OK* ab. Ihr Projekt ist somit angelegt. Sie gelangen anschließend auf die Projekt-Konfigurationsseite.
4. Im Abschnitt *Source-Code-Management* wählen Sie Subversion aus und geben unter *Module → Repository URL* den Pfad zu Ihrem Projekt an, z.B. `svn://subversion.acme.de/meinprojekt/trunk` (Abb. 7-2). Tipp: Bestimmte Eingaben wie Dateipfade und URLs werden von Hudson im Hintergrund automatisch validiert. Die Überprüfung wird beim Verlassen des jeweiligen Eingabefeldes gestartet und kann wenige Sekunden dauern. Ein Buchstabendreher in der Subversion-URL oder fehlende Zugriffsrechte werden also sofort erkannt und angezeigt.

Source-Code-Management

Keines

CVS

Subversion

Module **Repository URL** [?](#)

Lokales Modulverzeichnis (optional) [?](#)

Update-Kommando verwenden

Wenn angewählt, versucht Hudson - wenn immer möglich - 'svn update' auszuführen, um den Build zu beschleunigen. Dieser bedeutet allerdings auch, dass Artefakte des vorangegangenen Builds zu Beginn des neuen Builds nicht entfernt werden.

Zurücksetzen (revert)

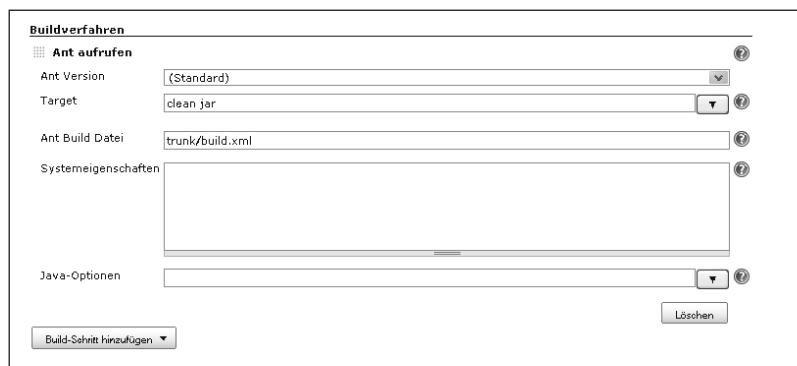
Wenn angewählt, führt Hudson "svn revert" vor dem Ausführen von "svn update" aus. Dies verlangsamt zwar den Buildprozess, verhindert aber, dass Dateien zwischen zwei Builds verändert werden.

Repository Browser [?](#)

Abb. 7-2
Konfiguration des
Versionsmanagements

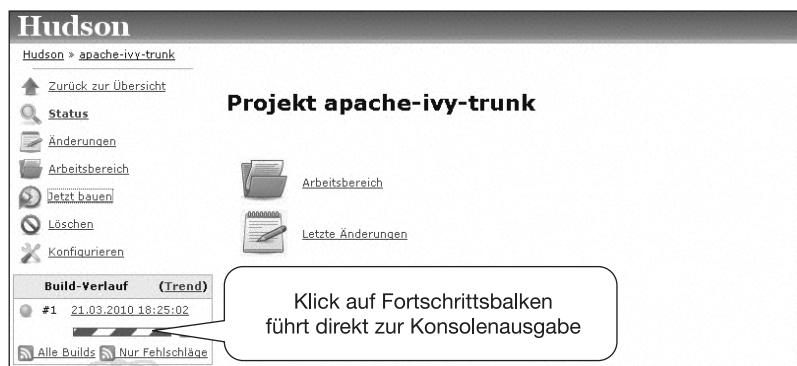
5. Im Abschnitt *Build-Verfahren* fügen Sie einen Build-Schritt vom Typ *Ant aufrufen* hinzu (Abb. 7–3). In das Feld *Target* tragen Sie das auszuführende Ant-Target ein oder lassen das Feld leer, um das Default-Target auszuführen, das in Ihrem Ant-Skript definiert ist. Über die Schaltfläche *Erweitert...* am rechten Rand des Formulars werden zusätzliche Optionen für den Ant-Aufruf eingeblendet. Im Feld *Ant Build Datei* geben Sie den Pfad zu ihrer Ant-Skript ein, und zwar relativ zum Arbeitsbereich – in unserem Beispiel `trunk/build.xml`.

Abb. 7–3
Konfiguration des
Build-Verfahrens



6. Sie schließen die Konfiguration des neuen Jobs ab, indem Sie Ihre Eintragungen mit der Schaltfläche *Übernehmen* am Ende der Seite abspeichern. Sie gelangen zurück zur Übersichtsseite Ihres Jobs (Abb. 7–4).

Abb. 7–4
Starten des ersten Builds



7. Wählen Sie *Jetzt bauen* aus den Symbolen der linken Seitenleiste aus (das Piktogramm zeigt eine Uhr mit einem grünen Dreieck). Im Kasten *Build-Verlauf* können Sie den Fortschritt des Builds verfolgen. Neben Bildnummer und Startzeitpunkt des Builds wird zusätzlich ein Fortschrittsbalken angezeigt.

8. Durch Klicken auf den Fortschrittsbalken gelangen Sie direkt zur Konsolenausgabe Ihres Builds (Abb. 7–5). Dort können Sie die Ausgaben von Ant in Echtzeit mitlesen.



The screenshot shows the Hudson build console for the job "apache-ivy-trunk". The output is as follows:

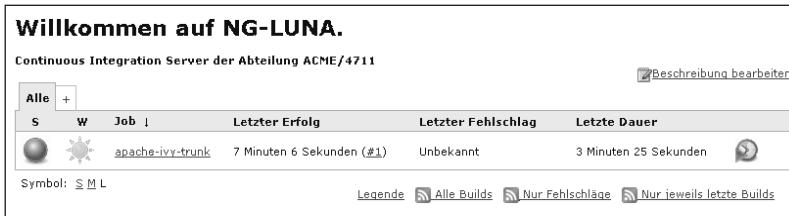
```

Gestartet durch Benutzer anonymous
Checking out a fresh workspace because D:\hudson\jobs\apache-ivy-trunk\workspace\trunk doesn't exist
Checking out http://svn.apache.org/repos/asf/ant/ivy/core/trunk
A test
A test.java
A testjavalog
A testjavalogapache
A testjavalogapacheivy
AU testjavalogapacheivy\TestHelper.java
A testjavalogapacheivy\plugins
A testjavalogapacheivy\plugins\repository
A testjavalogapacheivy\plugins\repository\wfs
AU testjavalogapacheivy\plugins\repository\wfs\WfsResourceTest.java
AU testjavalogapacheivy\plugins\repository\wfs\WfsRepositoryTest.java
AU testjavalogapacheivy\plugins\repository\wfs\WfsTestHelper.java
AU testjavalogapacheivy\plugins\repository\wfs\WfsURI.java
A testjavalogapacheivy\plugins\conflict
AU testjavalogapacheivy\plugins\conflict\ivysettings-latest.xml
AU testjavalogapacheivy\plugins\conflict\ivy-383.xml
A testjavalogapacheivy\plugins\conflict\ivynoconflict-dynamic.xml
AU testjavalogapacheivy\plugins\conflict\ivynoconflict-dynamic.xml

```

Abb. 7–5
Konsolenausgabe
eines Builds

9. Durch Klicken auf den Link *Hudson* am oberen linken Seitenrand gelangen Sie wieder zurück auf die Übersichtsseite Ihres Hudson-Systems (Abb. 7–6). Dort sehen Sie nun Ihren ersten Job in der Listenansicht angezeigt.



The screenshot shows the Hudson project overview page. The "Willkommen auf NG-LUNA." header is visible. Below it, the "Continuous Integration Server der Abteilung ACME/4711" section displays the "apache-ivy-trunk" job. The job details are as follows:

All	W	Job	Letzter Erfolg	Letzter Fehlenschlag	Letzte Dauer
		apache-ivy-trunk	7 Minuten 6 Sekunden (#1)	Unbekannt	3 Minuten 25 Sekunden

Below the table, there is a legend: "Symbol: S M L". At the bottom right, there are links for "Alle Builds", "Nur Fehlenschläge", and "Nur jeweils letzte Builds".

Abb. 7–6
Das neue Projekt erscheint
auf der Übersichtsseite.

Gratulation! Sie haben Ihren ersten Job erfolgreich in Hudson angelegt und gebaut. Sie haben damit die Grundlagen für eine Vollautomatisierung Ihres kompletten Build-Vorgangs gelegt. In den kommenden Abschnitten lernen Sie, wie Sie Ihren Build nun Schritt für Schritt erweitern können, zum Beispiel um:

- automatisiertes Starten von Builds
- automatisiertes Testen
- Versenden von Benachrichtigungen
- Integrieren erzeugter Dokumentation
- Überwachen der Codequalität
- Anbinden von Issue-Trackern

7.1.2 Statusanzeigen auf der Übersichtsseite

Wenn der Build-Prozess ohne Probleme abgeschlossen worden konnte, sollten Sie in der Übersichtsseite sowohl eine blaue Kugel als auch eine gelbe Sonne vor dem Namen des Jobs sehen können.

Ampeldarstellung des letzten Builds

Die blaue Kugel visualisiert den Ausgang des letzten Builds in einer Ampeldarstellung. Hudson kennt hier drei Zustände: erfolgreich (*successful*), instabil (*unstable*) oder fehlgeschlagen (*failed*). Im Gegensatz zu anderen CI-Servern unterscheidet Hudson also nicht nur binär zwischen einem erfolgreichen und einem fehlgeschlagenen Build, sondern kennt noch einen dritten Zustand, den instabilen Build. Worin liegt der Unterschied?

■ *Erfolgreich* (blau):

Der Build-Vorgang konnte aus technischer und fachlicher Sicht erfolgreich abgeschlossen werden.

■ *Instabil* (gelb):

Der Build-Vorgang konnte zwar aus technischer Sicht erfolgreich abgeschlossen werden, es wurden aber nicht alle Qualitätsziele erreicht. Beispiel: JUnit-Tests schlugen fehl, die Codeabdeckung war zu niedrig, die Kommentierung der Quelltexte unvollständig. In diesen Fällen muss typischerweise ein Entwickler tätig werden, um das Problem fachlich zu beheben.

■ *Fehlgeschlagen* (rot):

Der Build-Vorgang konnte aus technischer Sicht nicht erfolgreich abgeschlossen werden, etwa weil während des Kompilierens der freie Arbeitsspeicher zu knapp, die Festplatte zu voll oder das Versionsmanagementsystem nicht erreichbar war. In diesen Fällen muss ein Systemadministrator oder Build-Manager tätig werden, um Probleme eher technischer Natur zu beheben.

Projekte, die noch nie gebaut wurden, zurzeit deaktiviert sind oder zuletzt abgebrochen wurden, werden mit einer grauen Kugel dargestellt.

Pulsierende Bälle weisen, unabhängig ihrer Farbe, auf einen Job hin, der momentan neu gebaut wird.

Eine Ampel in Rot, Gelb und ... Blau?

Fast alle Hudson-Einsteiger wundern sich über die unorthodoxe Farbauswahl in der Ampeldarstellung (rot-gelb-blau statt rot-gelb-grün). Hierzu existieren mehrere Erklärungsansätze:

Zum einen sollte durch die Beschränkung auf die Grundfarben Rot, Gelb und Blau eine zu bunte und unübersichtliche Benutzeroberfläche vermieden werden (»Regenbogeneffekt«).

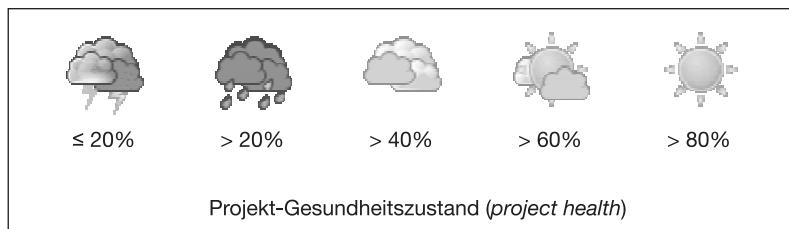
Zum anderen gibt es in einigen asiatischen Sprachen einen besonderen Gebrauch der Bedeutung »blau«^a. Im Japanischen etwa, der MutterSprache des Hudson-Initiators Kohsuke Kawaguchi, wird das grüne Ampelsignal für »Gehen« mit dem Wort für »blau« bezeichnet. Bei neuen Ampeln soll in Japan auch tatsächlich ein blaugrünes Licht zum Einsatz kommen. Dies ist jedoch nicht sprachlich motiviert, sondern hat mit dem verbesserten Farbkontrast für rot-grün-blinde Verkehrsteilnehmer zu tun.

Natürlich ist im westlichen Kulturkreis die grüne Ampelfarbe wesentlich verbreiteter. Es ist daher nicht verwunderlich, dass das Plugin »Green Balls«, welches alle blauen Bälle in grüne umfärbt, zu den meistinstallierten gehört.

a. Siehe auch http://de.wikipedia.org/wiki/Grün_und_Blaue_in_verschiedenen_Sprachen

Die gelbe Sonne stellt einen »Wetterbericht« dar, der den Verlauf der letzten fünf Builds zusammenfasst und in einen »Projekt-Gesundheitszustand« (*project health*) zwischen 0% und 100% umrechnet. Ein sonniges Projekt bedeutet keine nennenswerten Probleme in den letzten fünf Builds. Sind jedoch Probleme aufgetreten, so ziehen dunkle Wolken auf – bis hin zum Gewittersturm. Im Gegensatz zu den farbigen Ampelbällen sind hier fünf Abstufungen möglich (Abb. 7–7).

Trendanzeige als
Wetterbericht



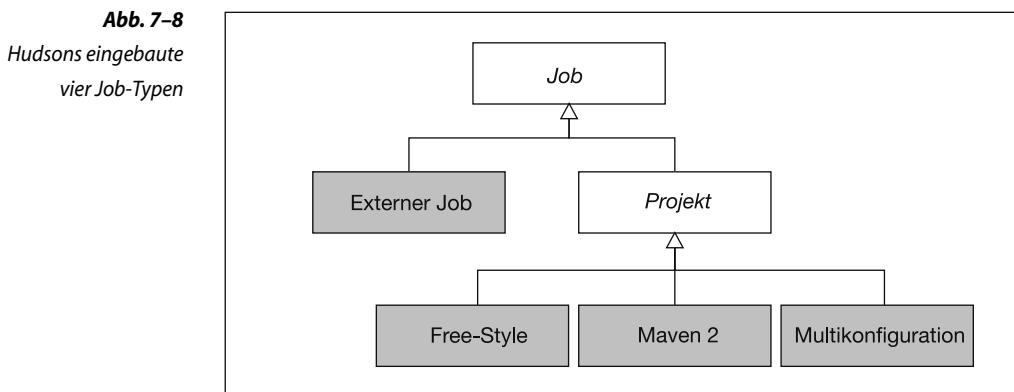
Der Wetterbericht kann auch Daten berücksichtigen, die von Plugins zugeliefert werden, etwa JUnit-Ergebnisse oder Resultate einer statischen Codeanalyse mit Checkstyle. Wenn Sie mit dem Mauszeiger über das Wettersymbol streichen, bekommen Sie eine detaillierte Darstellung über die eingeflossenen Werte angezeigt.

7.2 Jobtypen

Erinnern Sie sich? Beim Anlegen eines Jobs im Schnelldurchlauf in Abschnitt 7.1 wurden Sie zuerst nach dem gewünschten Jobtyp gefragt. In der Tat hat diese Entscheidung weitreichende Folgen. Sie kann – im Gegensatz zu allen anderen Konfigurationsmöglichkeiten eines Jobs – im Nachhinein nicht mehr geändert werden. Was verbirgt sich also hinter diesen Job-Typen?

Alle Aktivitäten innerhalb Hudsons finden in Jobs statt. Auf Hudsons Übersichtsseite sehen Sie alle angelegten Jobs Zeile für Zeile dargestellt. Hudson bringt in seiner Basisdistribution bereits vier Jobtypen mit, die in Abbildung 7-8 grau hinterlegt sind.

Weitere Jobtypen lassen sich per Plugin nachrüsten. So werden Maven-2-Projekte genau genommen erst durch ein spezielles Maven-Plugin ermöglicht. Da dieses aber stets in Hudsons Basisdistribution enthalten ist, fällt dieses Detail dem Benutzer nicht weiter auf.



7.2.1 Gemeinsamkeiten

Jobs Alle Jobs verfügen über einen Namen, eine Beschreibung und eine Liste ausgeführter Läufe (*runs*). Weitere Annahmen über den Zweck oder den genauen internen Aufbau eines Laufs werden bei einem Job nicht gemacht.

Projekte Ein Projekt hingegen ist ein Spezialfall eines Jobs, in dem Software gebaut wird. Dies impliziert bereits gewisse Ablaufschritte wie das Auschecken von Quelldateien, das Bauen der Quelldateien (z.B. Komplilierung und Tests) und abschließende Aktivitäten (z.B. Bewertung des Builds als Erfolg oder Fehlschlag). Läufe eines Projekts werden als

Builds bezeichnet. Der grundlegende Ablauf eines Builds ist in Abbildung 7–9 dargestellt:

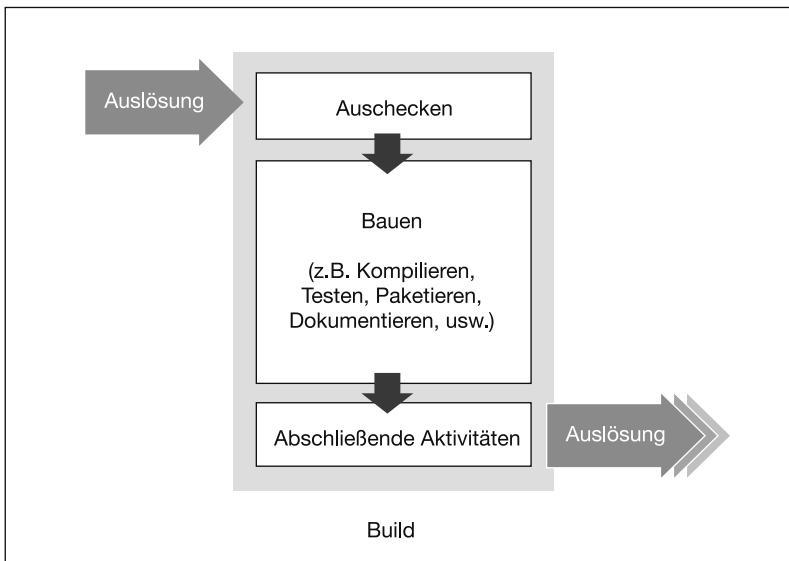


Abb. 7–9
Ablauf eines
Projekt-Builds

Ein Build kann über mehrere Wege ausgelöst werden, typischerweise entweder manuell, zeitgesteuert, durch Änderungen im Versionsmanagementsystem oder durch vorgelagerte Jobs.

Auslösung eines Builds

Im ersten Schritt »Auschecken« werden die benötigten Quelldateien aus einem Versionsmanagementsystem abgerufen und im Arbeitsbereich des Projekts lokal angelegt. Es können dabei auch mehrere Systeme abgefragt werden – entscheidend ist lediglich, dass am Ende dieses Schrittes alle Dateien im Arbeitsbereich vorliegen, die zur erfolgreichen Bearbeitung des Builds notwendig sind. Beachten Sie, dass die Kommunikation mit dem Versionsmanagementsystem von Hudson vorgenommen wird und nicht Teil eines Build-Skripts ist.

Auschecken

Im nächsten Schritt »Bauen« werden die Quelldateien verarbeitet und in gewünschte Endprodukte umgewandelt. Dies umfasst typischerweise das Kompilieren von Quelltexten, Durchführen von Unit- oder Integrationstests, Erstellen von Dokumentation, Paketierung der Ergebnisse in eine Distribution, Verteilen der Distribution usw. Aus Hudsons Sicht handelt es bei diesem Schritt weitestgehend um eine Black-Box. Hudson »zündet hier lediglich die Lunte«, etwa durch Starten eines Build-Werkzeugs wie Ant oder Maven, hat aber kein tiefergehendes Verständnis für dessen internen Aufbau oder Zweck.

Bauen

Abschließende Aktivitäten

Nachdem das Build-Werkzeug sein Feuerwerk abgebrannt hat, folgt der nächste Schritt »Abschließende Aktivitäten«. In diesem Schritt wird in der »Asche des Feuerwerks«, also in den entstandenen Dateien, nach Artefakten, Testberichten, Dokumentationen usw. gesucht. Je nach Anforderung werden diese Dateien archiviert, Benachrichtigungen verschickt, auf andere Server verteilt usw.

Auslösen nachgelagerter Builds

Am Ende eines Builds schließlich können wiederum weitere Builds in nachgelagerten Jobs ausgelöst werden. Mehr über Build-Auslöser (*build trigger*) erfahren Sie in Abschnitt 7.4.

Die weit überwiegende Mehrheit in Hudson angelegter Jobs dürfen Projekte sein. Um gängige Aufgabenstellungen besser zu unterstützen, bietet Hudson momentan Projekte in drei Ausprägungen an: Free-Style, Maven 2 und Multikonfiguration. Diese Typen unterscheiden sich in den Annahmen, die sie über den Build-Vorgang treffen. Bei einem Maven-2-Projekt beispielsweise kann eine POM-Datei vorausgesetzt werden, in der die zu bauende Software näher beschrieben wird. Dies erspart die redundante, manuelle Angabe von Konfigurationsoptionen und erleichtert die Verwaltung eines umfangreichen Hudson-Servers ungemein. Auf der anderen Seite geht ein Free-Style-Projekt von deutlich weniger Annahmen aus, erfordert dafür aber mehr an Konfigurationsarbeit.

7.2.2 Free-Style-Projekt

Dieser Projekttyp hat nicht nur den coolsten Namen, sondern gibt Ihnen auch tatsächlich die meisten Freiheiten im Ablauf Ihres Builds. Sie definieren dazu im Abschnitt *Build-Verfahren* der Projektkonfiguration eine Folge von Build-Schritten, die sequenziell abgearbeitet wird. Shell-Skripte, Batch-Dateien, Ant-Aufrufe, Maven-Builds usw. können dabei nach Belieben kombiniert werden. Buildschritte lassen sich per Drag-and-Drop umsortieren, so dass Sie auch im Nachhinein problemlos zusätzliche Schritte zu Beginn Ihres Build-Verfahrens einfügen können.

Da Sie über Shell-Skripte bzw. Batch-Dateien beliebige, auch proprietäre Werkzeuge in Ihre Builds einbinden können, ist das Free-Style-Projekt das »Schweizer Taschenmesser« des Build-Managers. Tatsächlich wird dieser Projekttyp so auch zur Automatisierung von Aufgaben eingesetzt, die zwar mit dem Bauen und Testen von Software im engeren Sinne nichts zu tun haben, aber trotzdem sehr praktisch für Softwareentwickler sind, beispielsweise für Deployments, das Löschen von temporären Verzeichnissen oder das Einspielen von Testszenarien.

7.2.3 Maven-2-Projekt

Bei diesem Projekttyp wird davon ausgegangen, dass Ihr Build auf Maven 2 basiert. Dies vereinfacht die Konfiguration und erlaubt zusätzliche, speziell auf Maven zugeschnittene Funktionen. Anstatt wie beim Free-Style-Softwareprojekt einzelne Build-Schritte anzugeben, reicht es etwa aus, den Ort der POM-Datei und die gewünschten Maven-Goals anzugeben. Hudson analysiert die POM-Datei und verwendet diese Informationen, um automatisch Vorgänger- und Nachfolgerbeziehungen mit anderen Projekten aufzubauen. Darüber hinaus kann Hudson auch komplette Maven-Multimodulstrukturen einlesen und den Build-Prozess auf Modulebene automatisch parallelisieren.

Im Gegensatz zu anderen CI-Servern wertet Hudson den Inhalt der POM-Datei bei jedem Build erneut aus. Ergeben sich also Änderungen in den Abhängigkeiten eines Projekts, so werden diese von Hudson automatisch aktualisiert. Sie müssen Maven-2-Projekte mindestens einmal bauen, um Hudson Gelegenheit zu geben, die POM-Datei des Projekts zu analysieren. Wundern Sie sich also nicht, wenn direkt nach dem Anlegen – aber vor dem ersten Build – noch keine Abhängigkeiten zu vor- und nachgelagerten Projekten aufgebaut sind.

Intern sind Free-Style- und Maven-2-Projekte sehr unterschiedlich implementiert. Ein Maven-2-Build ist also keineswegs eine Spezialisierung eines Free-Style-Builds. Für den Benutzer spielt dies insofern eine Rolle, als einige Plugins explizit für den einen, für den anderen oder für beide Projekttypen geschrieben sind. Aus diesem Grund finden sich manche Plugins zweimal im Plugin-Manager wieder: einmal unter der thematischen Eingruppierung, z.B. *Build-Berichte*, und ein zweites Mal unter *Maven bzw. Plugins mit besonderer Maven-Unterstützung*. Letzteres bedeutet, dass der Plugin weitergehende Funktionalitäten anbieten kann, wie etwa eine vereinfachte Konfiguration, weil Dateipfade per Maven-Konvention angenommen werden können.

Sollten Sie also Ihre Maven-basierten Entwicklungsprojekte in Hudson am besten immer als Maven-2-Projekt anlegen? Die sehr tiefgehende Unterstützung des Maven-Projektmodells spricht dafür und erspart Ihnen tatsächlich einiges an Konfigurations- und Wartungsaufwand. Auf der anderen Seite ist das Plugin-Angebot für Free-Style-Projekte (noch) deutlich umfangreicher. Sind Sie hier auf ein bestimmtes Plugin angewiesen, kann dies die Entscheidung in Richtung Free-Style-Projekt zwingend vorgeben.

*Ständige Auswertung
der POM-Datei*

*Free-Style- im Vergleich
zu Maven-2-Projekten*

7.2.4 Multikonfigurationsprojekt (»Matrix-Builds«)

Projekte ab einer gewissen Größe müssen oft in mehrere »Geschmacksrichtungen« angepasst werden, etwa für unterschiedliche Betriebssysteme, Datenbankhersteller oder Java-Versionen. Das Bauen und Testen aller möglichen Kombinationen ist dabei manuell nicht mehr zu bewältigen. Hudson erlaubt über die Angabe von sogenannten *Achsen* für Betriebssystem, Datenbankhersteller, Java-Version usw., eine Matrix aller Konfigurationen zu definieren. Diese werden dann, Konfiguration für Konfiguration, automatisch gebaut und das Ergebnis tabellarisch dargestellt. Mehr Details zur praktischen Verwendung von Multikonfigurationsprojekten erfahren Sie in Kapitel 8.

7.2.5 Überwachung eines externen Jobs

Externe Jobs dienen der Überwachung von Abläufen, die nicht innerhalb Hudsons stattfinden, sondern – wie es der Name ja andeutet – extern. Typische Kandidaten für externe Jobs sind Aktivitäten, die per cron ausgeführt werden und mehr als lediglich verschickte E-Mails am Ende der Aktivität zur Protokollierung benötigen. Mithilfe überwachter externer Jobs lässt sich ein zentrales »Cockpit« für diese Aktivitäten einrichten.

Dies können regelmäßige Wartungsaufgaben sein, deren Ergebnisse von Hudson archiviert und visualisiert werden sollen. Oder ein regelmäßig gestartetes Skript, das einen Status (z.B. eines Systems) ermittelt und an Hudson rückmeldet. Bei externen Jobs fungiert Hudson also nicht als aktiver Ausführer eines Jobs, sondern eher als passiver Buchhalter, der den Ausgang eines anderen Prozesses zur Archivierung und Visualisierung mitgeteilt bekommt.

Wie kann ein externer Prozess seine Ergebnisse an Hudson rückmelden? Hudson bietet dazu gleich zwei Möglichkeiten an:

1. durch Ausführen einer Wrapper-Anwendung für externe Jobs sowie
2. durch Kommunikation über die Hudson-REST-Schnittstelle.

Externe Jobs mit Wrapper-Anwendung anbinden

Die Wrapper-Anwendung für externe Jobs ist nicht zu verwechseln mit der Hudson-Kommandozeilenanwendung CLI. Vielmehr handelt es sich um eine Java-Anwendung, die ein angegebenes Programm als Unterprozess ausführt und dessen Ausgaben (STDOUT und STDERR) an einen Hudson-Server übermittelt.

Die Wrapper-Anwendung befindet sich in der Datei `hudson-core-<Hudson-Version>.jar`, die im `WEB-INF/lib`-Verzeichnis der Hudson-Webanwendung existiert. Kopieren Sie diese Datei sowie die weiteren

vier Dateien `remoting-*.jar`, `ant-1.7.0.jar`, `commons-lang-2.4.jar` und `xstream-*.jar` in ein Verzeichnis des Rechners, auf dem Sie das externe Kommando starten möchten. In diesem Verzeichnis starten Sie externe Jobs dann folgendermaßen:

```
$ export HUDSON_HOME=http://user:pw@hudson.acme.de
$ java -jar hudson-core-*.jar "meinJobName"
    <meinProgram arg1 arg2...>
```

Listing 7-1

Ausführen eines externen Jobs (Unix)

```
> set HUDSON_HOME=http://user:pw@hudson.acme.de
> java -jar hudson-core-*.jar "meinJobName" cmd.exe /c
    <meinProgram arg1 arg2...>
```

Listing 7-2

Ausführen eines externen Jobs (Windows)

Beachten Sie, dass Sie bei abgesicherten Hudson-Instanzen Benutzernamen und Kennwort in der `HUDSON_HOME`-Adresse angeben können. Bei Instanzen, die anonymen Zugriff erlauben, können Sie diese Angaben auch weglassen.

Die Ausgaben des ausgeführten Programms werden aufgezeichnet und als Lauf in Hudson archiviert. Der Exit-Code des Programms entscheidet zudem, ob der Lauf in Hudson als Erfolg (Exit-Code 0) oder Fehlschlag (Exit-Code ungleich 0) gilt.

Wenn Ihnen das Einrichten einer Wrapper-Anwendung zu aufwendig erscheint, können Sie als leichtgewichtigere Alternative auf die REST-Schnittstelle zurückgreifen. Hierzu müssen Sie die Ausgabe Ihres Programmes in einem bestimmten XML-Format als Nachricht an Hudson senden. Das folgende Beispiel zeigt das Ergebnis eines erfolgreichen Laufs, der 480 Millisekunden benötigte und den Text »ABC<Neue Zeile>« ausgab:

```
<run>
  <log encoding="hexBinary">4142430A</log>
  <result>0</result>          <!-- 0=Erfolg, sonst Fehlschlag -->
  <duration>480</duration>   <!-- in Millisekunden, optional -->
</run>
```

Externe Jobs mit REST-Schnittstelle anbinden

Listing 7-3

result.xml – Ergebnis eines externen Jobs

Beachten Sie, dass die Konsolenausgabe im Element `<log>` in HexBinary-Kodierung übertragen wird. Dadurch wird vermieden, dass Steuerzeichen in der Programmausgabe die XML-Syntax der Nachricht zerstören. Abschließend übertragen Sie diese XML-Nachricht per HTTP an Ihre Hudson-Instanz:

```
$ curl -X POST -d @result.xml
  http://user:pw@hudson.acme.de/job/meinJobName/postBuildResult
```

Listing 7-4

Übertragung des Ergebnisses eines externen Jobs an Hudson

Diese Art der Anbindung erfordert keinerlei Hudson-spezifische Dateien auf dem Rechner, der den externen Job ausführt. Sie müssen sich aber im Gegenzug um die korrekte Verpackung Ihrer Ergebnisse in das beschriebene XML-Datenformat kümmern.

7.3 Versionsmanagement integrieren

Eine wichtige Voraussetzung für CI lautet, dass alle Quelldateien, die für einen Build benötigt werden, in einem Versionskontrollsysteem verwaltet werden. Im Schnelldurchlauf in Abschnitt 7.1 haben Sie bereits ein Subversion-Repository verwendet, um auf Quelltexte zuzugreifen. In diesem Abschnitt betrachten wir die Integration von Versionsmanagementsystemen.

Breite Unterstützung von
Versionsmanagement-
systemen

CVS

Subversion

Weitere Versionsmanage-
mentsysteme

Hudson unterstützt in der Basisdistribution bereits CVS und Subversion.

Die CVS-Anbindung erfolgt durch Aufruf der CVS-Kommandozeilenanwendung. Eine lokale Installation eines CVS-Clients (cvs bzw. cvs.exe) ist daher Voraussetzung. Sie konfigurieren den Pfad zum CVS-Komandozeilenprogramm unter *Hudson → Hudson verwalten → System konfigurieren → CVS*.

Die Subversion-Anbindung wird durch ein Subversion-Plugin ermöglicht, das in der Hudson-Basisdistribution mit ausgeliefert wird. Das Plugin enthält SVNKit, eine Subversion-API, die zu 100% in Java realisiert ist. Sie benötigen – im Gegensatz zur CVS-Anbindung – also keine weitere lokale Installation einer Subversion- Komandozeilenanwendung.

Für rund 20 weitere Hersteller existieren Plugins, so dass Hudson eines der CI-Systeme mit der breitesten Unterstützung an Versionsmanagementsystemen sein dürfte – unter anderem für ClearCase, Git, Mercurial, Perforce, Synergy, Team Foundation Server oder Visual Source Safe. Eine vollständige Liste finden Sie im Plugin-Manager im Bereich *Verfügbar → Versionsverwaltung*. Da für die meisten Versionsmanagementsysteme keine zu 100% in Java implementierten APIs verfügbar sind, ist in der Regel die lokale Installation einer Komandozeilenanwendung notwendig (analog zum oben beschriebenen CVS-Client).

7.3.1 Konfiguration eines Projekts

Da Subversion momentan das mit Abstand populärste Versionsmanagementsystem im Zusammenspiel mit Hudson darstellt, werden in den folgenden Abschnitten Beispiele mit der Subversion-Anbindung gezeigt.

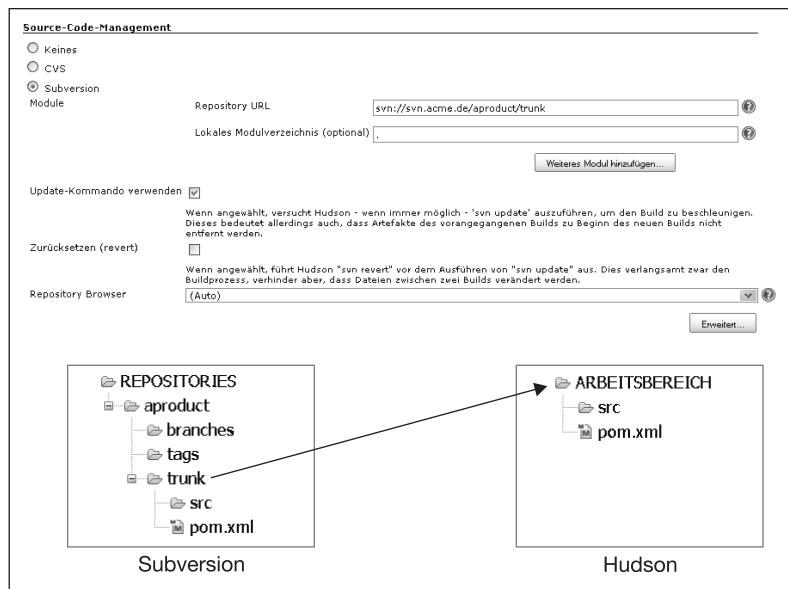
Die Quellen eines Projektes werden in der Projektkonfiguration im Abschnitt »Source Code Management« angegeben. Pro Projekt müssen Sie sich für ein bestimmtes Versionsmanagementprodukt entscheiden (also Subversion *oder* CVS *oder* Perforce usw.). Sie können Ihre

Quellen aber aus mehreren Zweigen eines Repositorys oder gar aus unterschiedlichen Repositories gleichen Typs beziehen. Jede Quelle wird in der Hudson-Oberfläche als Modul bezeichnet.

Verwenden Sie nur ein Modul, so legt Hudson die Revisionsnummer der ausgecheckten Revision in der Umgebungsvariable SVN_REVISION ab. Diese können Sie dann in Ihrem Build-Skript beispielsweise in Protokollen oder für Manifest-Dateien verwenden. Die Angabe eines lokalen Modulverzeichnisses ist optional. Bei nur einem Modul kann aber ein Eintrag von »..« (ein Punkt) praktisch sein, weil dann der angegebene Subversion-Pfad direkt in den Arbeitsbereich ausgecheckt wird. Im Beispiel in Abbildung 7–10 wird dadurch die Anlage eines Unterverzeichnisses `trunk` im Arbeitsbereich vermieden.

Arbeiten mit einem Modul

Abb. 7–10
Projektkonfiguration mit einem Modul

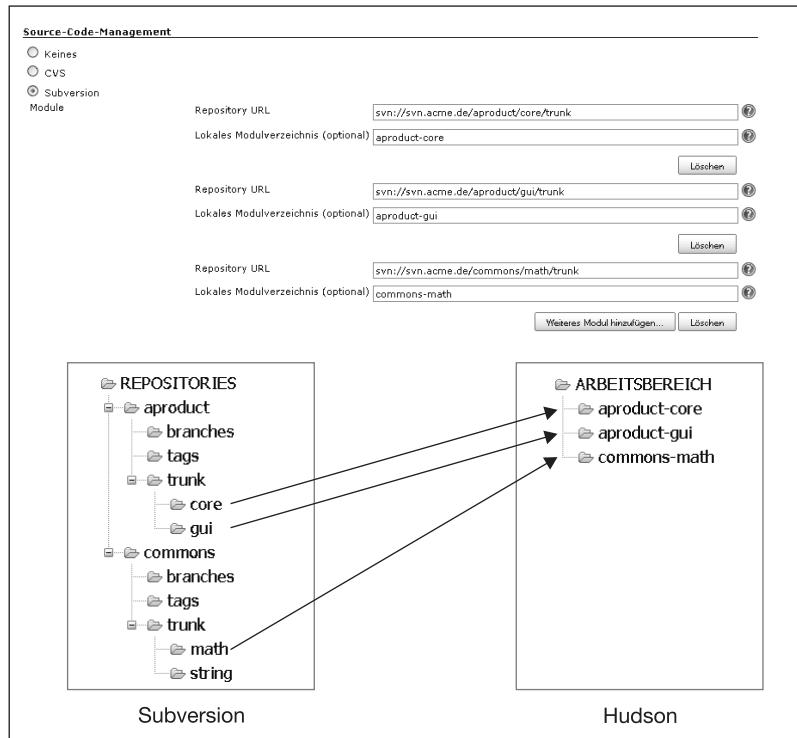


Verwenden Sie hingegen mehrere Module, so ist die Angabe von lokalen Modulverzeichnissen ausdrücklich empfohlen, um Übersicht im Arbeitsbereich Ihres Projekts zu behalten (Abb. 7–11). Lösen Sie neue Builds bei Änderungen im Versionsmanagementsystem aus, so reicht es bei mehreren Modulen übrigens aus, wenn sich mindestens eine davon verändert (Oder-Verknüpfung).

Arbeiten mit mehreren Modulen

Abb. 7-11

Projektkonfiguration mit mehreren Modulen



Subversion-spezifische Optionen

Speziell für Subversion-Repositories werden noch zwei zusätzliche Optionen angeboten, um die Build-Dauer zu verkürzen:

Update
Revert

Update-Kommando verwenden: Ist diese Option angewählt, werden die Quelldateien nicht mit dem Subversion-Kommando checkout, sondern mit update heruntergeladen. Dies beschleunigt Builds umfangreicher Projekte enorm, da in diesem Falle nicht alle Quelldateien erneut komplett übertragen werden. Es werden stattdessen nur die Änderungen heruntergeladen, die notwendig sind, um die lokale Arbeitskopie auf dem Hudson-Server auf den neuesten Stand zu bringen. Diese Optimierung bietet sich an, wenn Ihr Build-Anlauf die ausgescheckten Quellen *nicht* modifiziert. Nehmen Sie hingegen Änderungen an den Quelltexten vor, z.B. Ersetzen von Platzhaltern mit Zeit- und Revisionsstempeln, sollten Sie zusätzlich die Option *Revert* aktivieren.

Zurücksetzen (revert): Ist diese Option angewählt, wird vor dem Subversion-Kommando update zusätzlich ein revert ausgeführt. Dies nimmt zunächst alle Änderungen zurück, die Ihr Build-Prozess an Ihrer lokalen Arbeitskopie vorgenommen hat, bevor die Aktualisierungen aus dem Versionsmanagement heruntergeladen und eingefügt werden. Der zusätzliche Schritt erhöht Ihre Build-Dauer etwas. Die Kom-

bination aus revert und update kann aber immer noch deutlich schneller sein als ein umfangreiches checkout-Kommando vor jedem Build.

In der Praxis werden viele Projekte so konfiguriert, dass Änderungen in einem Subversion-Repository neue Builds auslösen. In manchen Fällen möchte man zwar komplette Module auschecken und bauen, jedoch einschränken, was darin als »Änderung« verstanden wird. Warum? Einige Build-Werkzeuge greifen etwa auch schreibend auf ein Repository zu. Dies könnte im ungünstigsten Falle einen sich immer wieder selbst auslösenden Endlos-Build verursachen. Oder es befinden sich im Repository Dateien, die zwar versioniert werden sollen, aber nicht direkt in das Endprodukt eingehen und deren Veränderung daher keine neuen Builds erfordert (z.B. interne Dokumentationen).

Einschränken, was als »Änderung« verstanden wird

In den erweiterten Subversion-Einstellungen können Sie daher nach fünf Kriterien filtern, welche Modifikationen im Repository als »auslösende Änderung« betrachtet werden. Alle Pfadangaben sind dabei relativ zur Modul-URL anzugeben. Das sehr konstruierte Beispiel in Abbildung 7-12 zeigt eine Konfiguration, die neue Builds nur auslöst bei (Und-Verknüpfung):

- Änderungen unterhalb von /aprogram/core,
- die keine HTML- oder JPEG-Dateien betreffen,
- nicht von den Benutzern hudson oder svnhook eingecheckt wurden,
- deren Commit-Beschreibung kein »IGNORE« enthält und
- deren Revision keine Eigenschaft¹ hudson:ignore trägt.

Excluded Regions	/aprogram/core/.*\html /aprogram/core/.*\jpeg
Included Regions	/aprogram/core/.*
Excluded Users	hudson svnhooks
Excluded Commit Messages	.*IGNORE.*
Exclusion revprop name	hudson:ignore

Abb. 7-12
Filtern der Änderungen in den erweiterten Subversion-Projekteinstellungen

Tipp: Wenn Sie Filterkriterien einrichten und deren Funktionsweise überprüfen möchten, hilft Ihnen Hudsons Protokollierung der Kommunikation mit Subversion. Sie erreichen es auf der Projektseite über die Funktion *Subversion Abfrage-Protokoll* am linken Seitenrand.

1. Erfordert einen Subversion-Server Version 1.5 oder neuer. Mehr zu Revisionseigenschaften (*revprop*) unter <http://svnbook.red-bean.com/en/1.5/svn.advanced.props.html>

**Bauen einer bestimmten
Revision**

Gelegentlich kann es vorkommen, dass man eine ganz bestimmte Revision aus der Vergangenheit nochmals bauen möchte. Es gibt für diesen Fall zwar kein Eingabefeld »Revisionsnummer« oder Ähnliches – aber ein besonderes Subversion-Feature kommt hier zur Hilfe: Sie können an eine Modul-URL die gewünschte Revisionsnummer in der Form @<revision> anhängen, also zum Beispiel für Revision 57 `svn://svn.acme.de/aproduct/trunk@57`. Selbst wenn inzwischen neuere Revisionen existieren, wird Hudson Revision 57 auschecken und verwenden. Sie sollten in diesem Beispiel außerdem einen Wert für das lokale Modulverzeichnis angeben (hier etwa `trunk`), da Hudson sonst per Default in ein Unterverzeichnis `trunk@57` auscheckt.

7.3.2 Repository-Browser anbinden

Welche Änderungen im Versionsmanagementsystem sind in einen bestimmten Build eingeflossen? Hudson stellt Ihnen von Haus aus bereits eine Liste dieser Änderungen über die Funktion *Änderungen* der Build-Seite dar, inklusive der Autoren, Revisionsnummern, Commit-Beschreibungen und betroffenen Dateipfade (Abb. 7–13, links oben).

Ohne weiteres können Sie sich jedoch nicht die geänderten Quelltexte in einer Vorher/Nachher-Gegenüberstellung anzeigen lassen. Diese Art der Visualisierung und Navigation in einem Versionsmanagementsystem ist die Domäne einer eigenen Softwaregattung, der sogenannten Repository-Browser, kurz: Repo-Browser. Bekannte Open-Source-Vertreter sind WebSVN, Sventon bzw. Atlassian FishEye als kommerzielles Produkt. Repo-Browser sind meist als Webanwendungen realisiert, die eine interaktive, grafische Benutzeroberfläche auf ein Repository bereitstellen. Diesen Umstand nutzt Hudson elegant, indem es alle Elemente aus einem Repository (Dateien, Revisionen, Change-Sets usw.) in seinen Ansichten mit HTML-Links in den Repo-Browser anreichern kann. Das geschieht so einfach und unauffällig, dass es von vielen Benutzern schlichtweg übersehen wird (siehe Abb. 7–13, rechts oben)! Beachten Sie, dass in der rechten Darstellung Dateinamen und Revisionsnummern mit einem Link hinterlegt sind. Zusätzlich ist ein Link (*diff*) hinzugekommen, der direkt in eine Vorher/Nachher-Vergleichsansicht des Repo-Browsers führt.

<p>Ohne Repository-Browser</p>	<p>Mit Repository-Browser</p>
<p>HTML-Links zu Quelltext, Revision und Differenzansicht</p>	
<p>Repository-Browser (hier: Sventon 2.1.4)</p>	

Abb. 7-13

Revisionen vergleichen
durch Anbindung eines
Repository-Browsers

Sie konfigurieren die Anbindung eines Repo-Browsers in der Projektkonfiguration im Abschnitt Source-Code-Management. In der Regel ist lediglich die URL Ihres Repo-Browsers notwendig, alles Weitere übernimmt Hudson für Sie. Abb. 7-14 zeigt die Konfiguration für Sventon 2.x.

Repository Browser	<input type="text" value="Sventon 2.x"/>
URL	<input type="text" value="http://ng-luna:9999/sventon/"/>
Repository Instance	<input type="text" value="greetr"/>
<input type="button" value="Erweitern..."/>	

Abb. 7-14

Konfiguration eines
Repository-Browsers

Die Anbindung von Repository-Browsern ist Aufgabe der Hudson-Plugins für das jeweilige Versionsmanagementprodukt, also des Subversion-Plugins, des Mercurial-Plugins usw. Dies leuchtet ein, da sich Versionsmanagementprodukte konzeptionell deutlich unterscheiden können und diese Unterschiede auch von den jeweiligen Repo-Browsern abgebildet werden müssen. Tabelle 7–1 zeigt eine Auswahl der gängigsten Versionsmanagementsysteme und der momentan unterstützten passenden Repo-Browser.

Tab. 7-1

*Versionsmanagement-
systeme mit unterstützten
Repository-Browsern
(Auswahl)*

Versionsmanagementsystem	Durch Plugin unterstützte Repo-Browser
CVS	FishEye, ViewCVS, ViewVC (eigener Hudson-Plugin)
Subversion	FishEye, WebSVN, Sventon 1.x/2.x, ViewSVN, CollabNet, ViewVC (eigener Hudson-Plugin)
Git	gitweb
Mercurial	bitbucket, hgweb, googlecode
Perforce	FishEye, P4Web
Team Foundation Server	Team System Web Access

*Automatische
Konfiguration eines
Repo-Browsers*

Die Anbindung des Repo-Browsers wird auf Projektebene vorgenommen. Da in den meisten Arbeitsgruppen ein zentraler Repo-Browser für alle Repositories zuständig ist, bietet Hudson in der Repo-Browser-Auswahlliste als Vorgabe *Auto* an. Hierbei versucht Hudson, die Repo-Browser-Konfiguration eines Projekts aus einem anderen Projekt abzuleiten, welches dasselbe Versionsmanagementsystem nutzt. Beispiel: Sie haben mehrere Projekte in Hudson angelegt, die Subversion verwenden. Eines der Projekte haben Sie bereits an einen Repo-Browser angebunden. Alle weiteren Projekte belassen Sie auf dem Vorgabewert »Auto«. Die richtige Konfiguration wird dort von Hudson »automatisch« ergänzt.

7.4 Auslösen von Builds

Builds können – je nach Anwendungsfall – entweder manuell oder durch Hudson ausgelöst werden. Im letzteren Fall konfigurieren Sie dieses Verhalten auf Projektebene im Abschnitt *Build-Auslöser* oder durch zusätzlich installierte Plugins.

7.4.1 Manuell (build trigger)

Dies ist die Voreinstellung. Projekte, die nur selten oder zu besonderen Anlässen laufen sollen, lösen Sie manuell über die grafische Oberfläche aus, z.B. ein Deployment auf Produktivserver.

Interaktiv auslösen

Alternativ können Sie einen Build auch über Hudsons REST-Schnittstelle starten, indem Sie eine URL nach dem Muster `http://hudson.acme.de/job/<meinProjekt>/build` abrufen. Auf diese Weise können Sie auch sehr einfach bestehende Skripte oder Web-Cockpits um »Hudson-Fähigkeiten« erweitern.

Per REST auslösen

Tipp: Zelebrieren Sie doch mal ein Deployment-Ritual.

Zum Abschluss eines Projekts, Meilensteins oder Sprints sollten Sie sich und Ihrem Team ein kleines Ritual gönnen!

Über den Versandhandel sind wichtig aussehende »Panik-Knöpfe« mit USB-Abschluss erhältlich (googeln Sie nach »USB panic button«). Mit einem kleinen Skript können diese auf Knopfdruck eine beliebige URL aufrufen – warum nicht auf diese Weise ein Deployment über Hudson auslösen? Sie erhalten so mit minimalem Aufwand eine dekorative Fernsteuerung Ihres Hudson-Servers, um die Sie Ihre Kollegen beneiden werden...

7.4.2 Zeitgesteuert

Sehr lang laufende Builds oder regelmäßige Wartungsaufgaben sollen meistens nur in festen Intervallen ausgeführt werden, z.B. einmal pro Nacht.

Hudson erlaubt unter *Build-Auslöser* → *Builds zeitgesteuert starten* die Konfiguration komplexer Zeitpläne. Diese werden in der cron-Syntax angegeben, die aus der Unix-Welt bekannt ist. Mit einem Satz aus fünf Angaben für Minute, Stunde, Tag des Monats (1–31), Monat und Wochentag (0–7, 7=Sonntag) geben Sie die Ausführungszeitpunkte an. Sie können in Ihrem Zeitplan auch mehrere Sätze angeben (Oder-Verknüpfung). Beispiele finden Sie in Tabelle 7–2. Eine vollständige Beschreibung der unterstützten Syntax erreichen Sie in der Online-Hilfe über das blaue Fragenzeichen-Symbol rechts neben dem Feld *Zeitplan*.

Zeitplanung über cron-Syntax

Zeitplan	Bedeutung
* * * * *	Jede Minute
15 * * * *	Jeweils 15 Minuten nach der vollen Stunde (Achtung: Also nicht alle 15 Minuten!)
*/15 * * * *	Alle 15 Minuten

Tab. 7–2
Beispiele für Zeitpläne
in cron-Syntax

Zeitplan	Bedeutung
0,15,30,45 * * * *	Alle 15 Minuten (Äquivalente Schreibweise zu vorhergehender Zeile)
0 2 * * 0-5	Montag–Freitag um 2 Uhr morgens
0 8-20 * * 0-5 0 9-14 * * 6	Montag–Freitag zwischen 8–20 Uhr, Samstag zwischen 9–14 Uhr, jeweils zur vollen Stunde
@daily	Einmal am Tag, wobei die genaue Uhrzeit nicht vorgegeben ist und von Hudson frei gewählt werden kann

7.4.3 Bei Änderungen im Versionsmanagementsystem

Projekte nach einem starren Zeitraster zu bauen, ist aus CI-Sicht nicht optimal. Warum sollte ein Projekt immer wieder neu gebaut werden, obwohl sich in der Zwischenzeit nichts im Versionsmanagement verändert hat?

Regelmäßig abfragen,
aber nur bei Änderung
bauen

Die Option *Build-Auslöser → Source Code Management System abfragen* veranlasst Hudson, zwar nach einem vorgegebenen Zeitplan das Versionsmanagement zu befragen, aber nur bei Veränderungen einen neuen Build auszulösen. Die Syntax ist identisch zur Option *Build-Auslöser → Builds zeitgesteuert starten*, daher ein Hinweis aus der Praxis: Manche Benutzer tragen irrtümlich ihren Zeitplan statt bei *Source Code Management System abfragen* ein Feld höher in *Builds zeitgesteuert starten* ein. Hudson beginnt dann tapfer Minute für Minute neue Builds zu bauen, obwohl sich gar nichts im Versionsmanagementsystem getan hat...

Sie möchten nicht bei
jeder Änderung bauen?

Möchten Sie zwar häufig auf Änderungen prüfen, aber inhaltlich einschränken, was überhaupt als »Änderung« angesehen wird (etwa die überwachten Dateien einschränken), so finden Sie in Abschnitt 7.3.1 interessante Konfigurationsmöglichkeiten für Subversion-Projekte. Auf diese Weise können Sie unnötige Builds einsparen und die Ressourcen Ihres Build-Systems sinnvolleren Tätigkeiten zukommen lassen.

Sollte man jede neue
Revision bauen?

Betrachten wir abschließend das andere Extrem: Sollte man nicht idealerweise *jede* neue Revision bauen? Wie lässt sich dies in Hudson konfigurieren (mein Chef möchte das unbedingt)? Die Frage taucht immer wieder auf den Hudson-Mailinglisten auf, weshalb ich hier eine Antwort zu geben versuche. Ja, idealerweise würde jede neue Revision gebaut, wenn dadurch die Geschwindigkeit der Rückmeldung nicht leidet. Es geht hier also um einen Kompromiss, denn: Würde *jede* neue Revision gebaut, könnte es im ungünstigsten Falle zu einem langen Rückstau an Builds kommen, d.h., die Zeit zwischen Commit und Rückmeldung zieht sich für den einzelnen Entwickler immer mehr in

die Länge. Nach meiner persönlichen Erfahrung hingegen bevorzugen Entwickler hier lieber schnellere Rückmeldungen, auch wenn dann pro Build mehrere neue Revisionen zusammengefasst werden müssen. Eine spezielle Option *Baue jede Revision* gibt es in Hudson nicht. Sie könnten allerdings – wenn Ihr Versionsmanagementsystem das anbietet – als eine Post-Commit-Operation einen Build über Hudsons REST-Schnittstelle auslösen (siehe Abschnitt 7.4.1). Subversion bietet solche Funktionalität über sogenannte *hooks* an.

7.4.4 Abhängigkeiten zu anderen Projekten

Builds können sich auch gegenseitig starten und dabei regelrechte »Build-Kaskaden« lostreten. Ein Projekt zur Kompilierung einer Bibliothek kann beispielsweise automatisch ein zweites Testprojekt starten. Dieses wiederum löst bei Erfolg ein drittes Projekt zur Veröffentlichung der Bibliothek auf einen FTP-Server aus. Das auslösende Projekt wird als *vorgelagertes* Projekt bezeichnet, das ausgelöste als *nachgelagertes* Projekt. Im Beispiel in Abbildung 7-15 ist TEST dem Projekt COMPILE vorgelagert, aber PUBLISH nachgelagert. Abhängige Projekte sind ein sehr wichtiges Werkzeug zur Optimierung von Build-Zeiten, wie wir in Kapitel 8 noch näher sehen werden (*build pipelines*).

Vor- und nachgelagerte Projekte

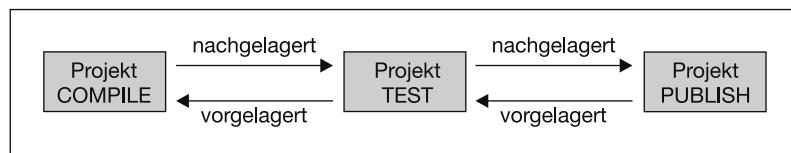


Abb. 7-15
Abhängigkeiten zwischen Projekten

In Free-Style-Projekten werden Abhängigkeiten im Abschnitt *Build-Auslöser → Starte Build, nachdem andere Projekte gebaut wurden* konfiguriert. Hier geben Sie eine Liste der *vorgelagerten* Projekte an. Sie können jedoch auch umgekehrt unter *Post-Build-Aktionen → Weitere Projekte bauen* eine Liste *nachgelagerter* Projekte eintragen. Da diese Beziehung symmetrisch ist, werden Eintragungen in den korrespondierenden Projekten automatisch reflektiert.

Manuelle Abhängigkeiten bei Free-Style-Projekten

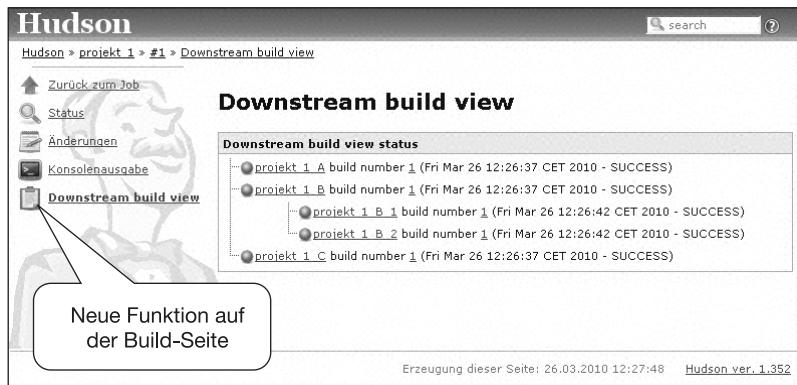
Eine Besonderheit des Maven-2-Projekttyps ist die automatische Verwaltung dieser Abhängigkeiten. Werden beispielsweise in der gleichen Hudson-Instanz die Maven-Module `de.acme.aproduct.core` und `de.acme.aproduct.gui` in zwei getrennten Projekten gebaut und ist `core` in der POM-Datei von `gui` als Abhängigkeit deklariert, so startet ein neuer Build von `core` im Anschluss automatisch einen neuen Build von `gui`.

Automatische Abhängigkeiten bei Maven-2-Projekten

Plugin »Downstream Build View«

In der Projektkonfiguration sehen Sie jeweils nur die direkt vor bzw. nachgelagerten Projekte. Haben Sie sich hingegen eine hübsche Kaskade über mehrere Projekte hinweg ausgeheckt, verlieren Sie an dieser Stelle den Überblick. Abhilfe kann das Plugin »Downstream Build View« schaffen, das Ihnen nach Ablauf eines Builds eine grafische Visualisierung der abgearbeiteten Kette erzeugen kann. Das Plugin installiert auf der Build-Seite die Funktion *Downstream Build View* am linken Seitenrand nach (Abb. 7–16).

Abb. 7–16
Plugin »Downstream Build View«



Plugin »Downstream-Ext«

Wenn Sie mehr Kontrolle über das Auslösen nachgelagerter Projekte benötigen, sollten Sie das Plugin »Downstream-Ext« in Betracht ziehen (Abb. 7–17). Im Vergleich zu Hudsons eingebauter Funktion für nachgelagerte Projekte können Sie hier präziser festlegen, bei welchem Build-Ergebnis nachfolgende Projekte ausgelöst werden sollen, z. B. nur bei Erfolg, nur bei Fehlschlag usw. Außerdem können Sie nachgelagerte Projekte nur dann bauen lassen, wenn für diese Projekte im Versionsmanagementsystem Änderungen vorliegen. Das verwandte Plugin »Parameterized Trigger« erlaubt ebenfalls das Auslösen nachgelagerter Projekte vom Build-Ergebnis abhängig zu machen, kann aber zusätzlich noch Parameter an nachgelagerte Projekte übergeben.

Abb. 7–17
Plugin »Downstream-ext«



7.4.5 Auslösung durch Plugins

Die meisten Builds werden auf die bisher beschriebenen Arten ausgelöst. Dank Hudsons Plugin-Konzept sind aber natürlich auch die kreativsten Alternativen denkbar – und teilweise auch bereits verfügbar! Um das Spektrum aufzuzeigen, finden Sie in Abbildung 7–3 eine Auswahl an Plugins, die Builds auslösen können.

Plugin	Arbeitsweise
URL Change Trigger	Startet einen neuen Build, wenn sich der Inhalt einer URL verändert.
Files Found Build Trigger	Durchsucht regelmäßig ein lokales Verzeichnis auf dem Server und startet bei Vorhandensein bestimmter Dateinamen einen neuen Build.
Startup Trigger	Startet einen Build einmalig beim Hochfahren des Hudson-Systems.
Join Plugin	Startet ein Projekt, nachdem <i>alle</i> seine nachgelagerten Projekte gebaut wurden. Ein typischer Anwendungsfall wäre das »Einsammeln« von Teilergebnissen aus parallel ausgeführten Builds.
Centralized Job(Re)Action	Startet ein Projekt nach einer einstellbaren Verzögerung neu, wenn in der Konsolenausgabe eines Builds ein bestimmtes Textmuster vorkommt.
Naginator	Startet Projekte nach fehlgeschlagenen Builds sofort neu. Das kann praktisch sein, wenn der Build von unzuverlässigen externen Systemen abhängt oder man prinzipiell den Leidensdruck auf die Entwickler erhöhen möchte (Naginator bedeutet auf Deutsch etwa »Nervinator«). Persönlich würde ich Ihnen allerdings dringend raten, lieber Ihren Build auf stabilere Füße zu stellen, als Entwickler mit einer Flut von E-Mails zu reizen...
Retry Failed Builds	Startet Projekte mit fehlgeschlagenen Builds nach einer einstellbaren Pause erneut. Als etwas »weichere« Alternative zum Paginator kann man dabei auch ein Maximum an Wiederholungsversuchen angeben.
Jabber/IRC	Chatten Sie mit Hudson! Sie bekommen Meldungen aus Ihrem Hudson-System per Instant Messenger, können aber auch Textkommandos wie <code>!build project 'foobar'</code> absetzen. Dieses Kommando würde etwa einen neuen Build des Projekts »foobar« anstoßen.

Tab. 7-3
Plugins, die weitere Builds auslösen können (Auswahl)

7.5 Testwerkzeuge integrieren

»Continuous Integration« ist mehr als nur das häufige Kompilieren einer Software. Der Witz besteht ja vielmehr darin, auch vollautomatisierte Schritte zur Qualitätssicherung einzubauen, etwa Unit-, Integrations- und Systemtests. Hudson Stärke liegt hier in der Visualisierung der Testergebnisse sowohl eines einzelnen Builds als auch projektweit über mehrere Builds hinweg.

7.5.1 JUnit, TestNG

Konfiguration bei Free-Style-Projekten

JUnit und TestNG unterstützen Hudson bereits in der Basisdistribution. Sie müssen lediglich zwei Dinge in Ihren eigenen Projekten sicherstellen:

Erstens müssen nach Beendigung der Build-Werkzeuge die Testergebnisse als XML-Datei im Arbeitsbereich vorliegen. Ant-Benutzer wählen hierzu im Ant-Skript die Ausgabe des XML-Formatters (Listing 7–5). Bei Maven ist dies bereits die Voreinstellung.

Listing 7–5

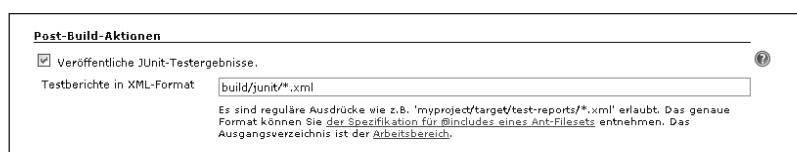
Konfiguration des XML-Formatters innerhalb eines Ant-Tasks

```
<junit . . . >
    <formatter type="xml"/>
    .
    .
    </junit>
```

Zweitens muss in der Projektkonfiguration spezifiziert sein, wo Hudson diese Ergebnisse zur Auswertung abholen soll. Bei Free-Style-Projekten geben Sie dies unter *Post-Build-Aktionen → Veröffentliche JUnit-Testergebnisse* an (Abb. 7–18). Tragen Sie dort den Namen der XML-Datei ein. Sie können auch Wildcard-Zeichen verwenden, wenn Sie die Ergebnisse mehrerer Test-Suites zusammenfassen möchten.

Abb. 7–18

Einbindung von JUnit-Ergebnissen im Free-Style-Projekt



Automatische Konfiguration bei Maven-2-Projekten

In Maven-2-Projekten werden Sie die Option *Post-Build-Aktionen → Veröffentliche JUnit-Testergebnisse* vergeblich finden. Warum? Weil sie nicht notwendig ist. Hudson weiß ja durch Analyse der POM-Datei bereits Bescheid, ob Sie JUnit einsetzen und wo Ihre Ergebnisdateien zum Liegen kommen. JUnit-Berichte erscheinen bei diesem Projekttyp also automatisch ohne weiteres Zutun.

Ist Ihr Projekt korrekt konfiguriert und hat der Build JUnit-Testergebnisse geliefert, so erscheint auf der Projektseite eine Trendgrafik

mit den JUnit-Ergebnissen und auf der jeweiligen Build-Seite ein neues Symbol *Testergebnisse* mit einer kurzen Zusammenfassung. Über dieses Symbol können Sie hierarchisch in Ihren Testergebnissen navigieren – bis hin zum Einzeltest (Beispiele siehe Abb. 7–19). Sie bekommen zusätzlich auch Angaben zur Laufzeit der Tests und gegebenenfalls der Anzahl der Builds, seit denen ein bestimmter Test fehlschlägt.

Die Visualisierung der JUnit-Testergebnisse ist in Hudson so ausgereift, dass selbst Intensiv-Anwender immer wieder neue Funktionen, Übersichten und pfiffige Querverweise entdecken. Hier ein paar Vorschläge, die Sie ausprobieren sollten:

- Alle Tabellen lassen sich neu sortieren, indem Sie auf den Kopf der jeweiligen Spalte klicken. Eine praktische Anwendung: Sortieren Sie Tests nach der Laufzeit und entdecken Sie die Langläufer.
- Testergebnisse lassen sich mit einer Beschreibung versehen. Dies funktioniert auf der Ebene eines einzelnen Tests, einer Suite, eines Packages bis hin zum gesamten Build. Dadurch können Sie im Nachhinein Ihre Testergebnisse bequem kommentieren (»Netzwerk war nicht erreichbar. Daher sind alle Datenbanktests in diesem Build fehlgeschlagen. Ursache: Praktikant über Kabel gestolpert«).
- Ebenfalls auf allen Ebenen – also Einzeltest, Suite, Package und Build – lässt sich die Entwicklung der Ergebnisse über die vergangenen Builds visualisieren (über die Funktion »Verlauf« am linken Seitenrand). Sie können dabei zwischen der Anzeige der Laufzeit und Anzahl durchgeföhrter Tests umschalten.
- Für fehlschlagende Tests wird deren »Alter« als die Anzahl der Builds seit dem letzten Erfolg angegeben. Das Alter ist mit einem Link zum Build hinterlegt, in dem der Test zum ersten Mal versagte.
- Nutzen Sie die Pfad-Navigation (*bread crumb*) am oberen Seitenrand, um schnell wieder von der Einzeltest-Darstellung auf Package- oder Build-Ebene »aufzutauchen«.
- Hängen Sie an die URL einer Testübersicht ein /api/xml an, so bekommen Sie die Testergebnisse in maschinenlesbarer Form – z.B. für komplexere Auswertungen durch eigene Skripte.

Nutzen Sie das volle Potenzial der JUnit-Visualisierung.

Abb. 7-19

Visualisierung von JUnit-Ergebnissen: Einstieg über die Build-Seite (oben), Testergebnisse eines Builds (Mitte), Verlaufsdiagramm einer Testsuite (unten)

The figure consists of three vertically stacked screenshots of the Hudson web interface, each showing a different aspect of JUnit test results:

- Top Screenshot (Build Page):** Shows the main Hudson dashboard for 'Build #24 (26.03.2010 17:24:15)'. A callout bubble points to the 'Testergebnisse' link in the top right, which leads to the detailed test results page.
- Middle Screenshot (Test Results Page):** Shows the 'Testergebnis' page for 'Build #24'. It displays a summary table of failed tests ('Alle fehlschlagenden Tests') and a full table of all tests ('Alle Tests'). A callout bubble points to the 'Fehlertabelle' section, which lists a single failing test: 'Greeting of a known person is incorrect. expected=<Hello, Duke> but was=<Hello, Duke>'. Below this is a chart titled 'Verlauf von GreeteTest' showing the execution time of the test over several builds.
- Bottom Screenshot (Test Suite History):** Shows the 'Verlauf von GreeteTest' chart in more detail, with a callout bubble pointing to it. Below the chart is a table summarizing the execution times for various builds of the 'GreeteTest' suite.

Beschreibung	Dauer	Fehlgeschlagen	Ausgelassen	Summe
greet: #24	0 ms	0	0	4
greet: #23	16 ms	0	0	4
greet: #22	Breiter Build auf neuem Testserver, daher Zeiten niedriger.	14 ms	0	4
greet: #19	0 ms	0	0	4
greet: #18	31 ms	0	0	4
greet: #12	31 ms	0	0	4

Nachgelagerte Tests zusammenfassen

Wenn Sie mit nachgelagerten Builds arbeiten, möchten Sie manchmal der Übersichtlichkeit halber die Testergebnisse der nachgelagerten Projekte zusammengefasst an einer Stelle dargestellt sehen. Beispiel: Sie haben ein Projekt A, in dem eine Anwendung erstellt wird. Anschließend wird diese in den nachgelagerten Projekten B1, B2, B3, B4 mit vier unterschiedlichen Datenbanken getestet. Zur Betrachtung der Testergebnisse möchten Sie sich nicht durch fünf Projekte hangeln (A, B1, B2, B3, B4), sondern alle Ergebnisse zusammengefasst in Projekt A sehen. Dies erreichen Sie durch die Option **Post-Build-Aktionen → Nachgelagerte Testergebnisse zusammenfassen** (Abb. 7-20). Diese Option ist sowohl für Free-Style- als auch Maven-2-Projekte verfügbar.

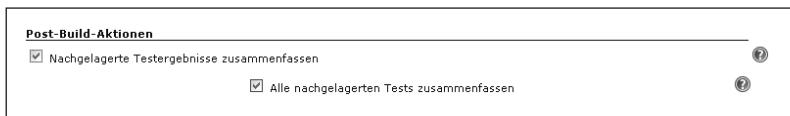


Abb. 7-20
Konfiguration
aggregierter
Testergebnisse

Plugin »JUnit
Attachments«

Werden während eines Tests neue Dateien erstellt, ist es oftmals wünschenswert, diese zusammen mit den Testergebnissen zu archivieren. Beispiel: Sie entwickeln eine Statistiksoftware, die Diagramme als PDF-Dateien ausgibt. Um im Falle eines Falles fehlgeschlagene Tests besser analysieren zu können, möchten Sie für jeden automatisierten Test auch die dabei erzeugten PDF-Dateien archivieren.

Das Plugin »JUnit Attachments« erweitert dazu Hudsons eingebaute JUnit-Berichtsfunktionen und archiviert am Ende eines Builds alle Dateien eines Ordners, der im gleichen Verzeichnis wie die XML-Datei mit den JUnit-Ergebnissen liegen und den vollen Namen der Java-Testklasse tragen muss (z.B. de.acme.project123.RenderTest). Die gesicherten Dateien werden dann innerhalb der Weboberfläche bei den Testergebnissen aufgeführt und sind mit Links zum Herunterladen versehen.

7.5.2 Weitere Test-Frameworks der xUnit-Familie

Neben JUnit existieren für andere Programmiersprachen und Testmethoden zahlreiche Frameworks, die JUnit konzeptionell sehr ähnlich sind und als xUnit-Familie bezeichnet werden. Geben diese Frameworks ein JUnit-kompatibles XML-Format aus, so können Ihre Ergebnisse direkt von Hudson ausgewertet werden (z.B. FlexUnit). Falls nicht, bleiben immer noch drei Möglichkeiten:

- Es existiert bereits ein eigens dafür vorgesehenes Plugin für das verwendete Format – durchsuchen Sie dazu den Plugin-Manager nach dem Namen des Frameworks.
- Die Ausgabe des Frameworks kann vollautomatisch in das JUnit-Format transformiert werden. Diese Idee steckt hinter dem Plugin »xUnit«, das über diesen »Trick« die folgenden Frameworks unterstützt: MSTest, NUnit, UnitTest++, Boost Test Library, PHPUnit sowie Free Pascal Unit. Darüber hinaus gibt es noch spezialisierte Plugins, die »xUnit« erweitern. Auf diesem Wege können zusätzlich CppUnit, Parasoft C++Test, Gallio, JUnit und AUnit angebunden werden.
- Sie modifizieren ein bestehendes Plugin. Oftmals muss nur der Dateiparser auf das eigene Format zugeschnitten werden. Wenn Sie ein gängigeres Test-Framework auf diese Weise erschlossen haben,

könnten Sie ja auch darüber nachdenken, Ihre Schöpfung der Hudson-Gemeinde zu stiften.

Tipp: Eigenes Test-Framework? Legen Sie ein Kuckucksei!

Hudson findet weit über die Java-Welt hinaus Verwendung. Auch Arbeitsgruppen, die in völlig anderen Sprachen entwickeln, setzen Hudson erfolgreich ein. Die Anbindung bereits bestehender und oft individuell entwickelter Test-Frameworks kann durch folgenden »Trick« schnell und elegant umgesetzt werden:

Statt wie bisher Text-, CSV- oder HTML-Berichte zu erzeugen, wird eine Ausgabe im JUnit-XML-Format generiert und Hudson als JUnit-»Kuckucksei« zur Visualisierung untergeschoben. Da die baumartige Organisation der Tests nach Java-Packages eine reine Konvention darstellt, kann man in anderen Umgebungen die Punkt-Notation zweckentfremden, um eigene Testhierarchien aufzubauen, z.B. `integrationtests.pdf-ausgabe.layout.din-a4`.

7.6 Benachrichtigungen verschicken

Im Idealfall leistet Hudson seine Dienste im Hintergrund zuverlässig, konstant und unauffällig. Wenn jedoch Builds fehlschlagen, sollten Sie so schnell wie möglich auf angemessene Weise davon erfahren. Dazu stehen Ihnen bereits in der Basisdistribution mehrere Kanäle zur Verfügung. Über Plugins – Sie ahnen es bereits – kommen noch weitere dazu.

7.6.1 E-Mail

Twitter und iPhone zum Trotz: E-Mail ist nach wie vor der gebräuchlichste Weg, sich über Hudsons Aktivitäten auf dem Laufenden zu halten. Hudson bringt deshalb bereits in der Basisdistribution die Möglichkeit mit, E-Mails zu versenden. Diese E-Mails enthalten – neben Namen und Status des Projekts – lediglich einen Link auf die Übersichtsseite des beendeten Builds. Das klingt zunächst spartanisch, reicht oftmals jedoch völlig aus. Die interaktiven Analysemöglichkeiten in Hudsons Weboberfläche sind damit schließlich nur einen Klick entfernt.

Nachdem Sie auf systemweiter Ebene Ihren E-Mail-Server konfiguriert haben, können Sie pro Projekt den E-Mail-Versand aktivieren (Abb. 7–21). Bei Free-Style-Projekten geschieht das unter *Post-Build-Aktionen* → *E-Mail-Benachrichtigung*, bei Maven-2-Projekten unter *Build-Einstellungen* → *E-Mail-Benachrichtigung*.

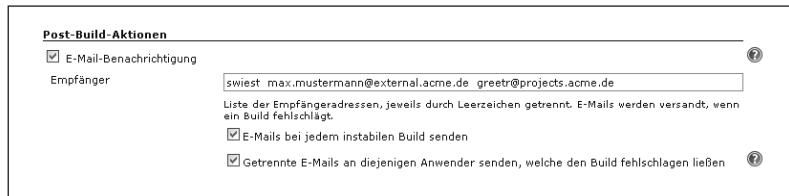


Abb. 7-21
Konfiguration der
E-Mail-Einstellungen
(Projektebene)

Die Liste der E-Mail-Empfänger wird pro Projekt fest angegeben. In der Regel wird man hier entweder wenige Senior-Entwickler angeben oder die E-Mail-Adresse eines projektweisen Verteilers. Hudson sendet dann E-Mails nach folgendem Schema:

- Jeder fehlgeschlagene Build löst eine E-Mail aus.
- Ein erfolgreicher Build, der auf einen fehlgeschlagenen oder instabilen folgt, löst ebenfalls eine E-Mail aus, um über die nun ausgestandene Krise zu informieren.
- Ein instabiler Build nach einem erfolgreichen löst eine E-Mail aus, um über einen Regressionsfehler zu informieren.

Vielleicht vermissen Sie bei dieser Aufstellung die Möglichkeit, bei *jedem* Build eine E-Mail abzusetzen, also unabhängig von dessen Ausgang. Dies ist mit Hudson Bordmitteln nicht möglich, wohl aber mit dem Plugin Email-ext aus dem folgenden Abschnitt 7.6.2. Bitte widerstehen Sie trotzdem der Versuchung, sich selbst und Ihre Kollegen rund um die Uhr mit Hudson-E-Mails zu beglücken. Meist erreichen Sie nur, dass im E-Mail-Programm zügig eine Filterregel angelegt wird, die solche E-Mails einer schnellen Beseitigung zuführen ...

Per Vorgabe löst jeder instabile Build eine E-Mail aus, um über anhaltende Regressionsfehler zu informieren. Für nachlässige Projekte, in denen instabile Projekte der Regelfall sind, kann das zu viel des Guten sein. Wählen Sie in diesem Fall die Option *E-Mail für jeden instabilen Build senden* ab, um nicht in E-Mails zu ertrinken.

Die Option *Getrennte E-Mails an diejenigen Anwender senden, welche den Build fehlschlagen ließen* erlaubt es, die Liste der Empfänger dynamisch um alle Entwickler zu erweitern, die Code zu einem fehlgeschlagenen Build beigetragen haben. Dies ist sinnvoll, denn meist haben diese fachlich die größten Chancen, das aufgetretene Problem wieder zu beheben. Hudson bestimmt diese Empfänger automatisch aus den Revisionsinformationen des Versionsmanagementsystems.

Wie ermittelt Hudson E-Mail-Adressen seiner Benutzer, insbesondere von Entwicklern, die in ein überwachtes Versionsmanagementsystem eingecHECKT haben?

E-Mail für jeden instabilen Build senden

E-Mails an Verursacher instabiler oder fehlerhafter Builds

Ermittlung der E-Mail-Adressen

- Vollständige E-Mail-Adressen (z.B. mustermann@acme.de) werden direkt verwendet.
- Unvollständige E-Mail-Adressen (z.B. mustermann) bekommen das E-Mail-Suffix nachgestellt, das Sie in den E-Mail-Einstellungen auf der systemweiten Konfigurationsseite angegeben haben (z.B. @acme.de). Wenn die Kontennamen auf E-Mail-Server und im Versionsmanagementsystem übereinstimmen, wird so dem Committer mustermann automatisch die richtige E-Mail-Adresse mustermann@acme.de zugeordnet.
- Weichen die Kontennamen ab, können Sie auf der Konfigurationsseite des betreffenden Benutzers (z.B. <http://<ihr.hudson.server>/user/mustermann/configure>) individuell eine E-Mail-Adresse eintragen (z.B. max.mustermann@a0815.acme.de).
- Für größere Installationen kann Hudson E-Mail-Adressen auch aus einem LDAP-Verzeichnis ermitteln.

7.6.2 Email-ext (Plugin)

Wenn Sie mehr Kontrolle über den E-Mail-Versand wünschen, hilft Ihnen das Plugin »Email-ext« weiter – eines der populärsten Hudson-Plugins überhaupt. »Email-ext« ist momentan noch nicht ins Deutsche lokalisiert, daher sind im Folgenden jeweils die englischen Feldbezeichnungen angegeben. Das Plugin arbeitet unabhängig von Hudsons eingebauten E-Mail-Fähigkeiten. In der Praxis werden Sie also *entweder* die eingebauten Funktionen nutzen (Abschnitt 7.6.1) *oder* das Plugin »Email-ext«. Dieses eröffnet Ihnen zusätzliche Einstellungen hinsichtlich:

- *Auslösung*: Wann soll eine E-Mail verschickt werden?
- *Verteiler*: An wen werden die E-Mails verschickt?
- *Inhalt*: Welchen Betreff und inhaltlichen Aufbau hat die E-Mail?

Zur Konfiguration geben Sie auf Projektebene zunächst eine globale Liste aus Empfängern an (*Global Recipient List*). Als nächsten Schritt wählen Sie das Format der E-Mails (Nur-Text oder HTML) aus und passen den Betreff und Aufbau der E-Mail projektbezogen an. Sie können diese Werte aber auch auf den Default-Einstellungen belassen. In diesem Fall greifen die Werte, die in der systemweiten Konfiguration eingetragen sind – dies ist praktisch, wenn Sie für alle Projekte einen einheitlichen Aufbau der verschickten E-Mails erzielen möchten. Abschließend geben Sie pro Ereignis an, an wen E-Mails verschickt werden sollen (*trigger*).

Als Auslöser können Sie nicht nur die unterschiedlichen Build-Ergebnisse (fehlgeschlagen, instabil, erfolgreich) verwenden, sondern auch Übergänge zwischen den Zuständen (z.B. *instabil* → *erfolgreich* oder *fehlgeschlagen* → *fehlgeschlagen*).

Auslöser anpassen

Der Verteiler einer E-Mail kann sich dynamisch aus der globalen Empfängerliste (*Send to Recipient List*), aus den beteiligten Committern eines Builds (*Send To Committers*) und den »Übeltätern« (*Include Culprits*) zusammensetzen. Die »Übeltäter« sind alle Entwickler, die seit dem letzten erfolgreichen Build Änderungen ins Versionsmanagement eingechekkt haben.

Verteiler anpassen

Betreff und Inhalt der E-Mails lassen sich aus knapp 20 Bausteinen (*Content Tokens*) zusammenstellen – und sogar je nach auslösendem Ereignis individuell anpassen. Solche Bausteine sind beispielsweise Projektname, Build-Nummer, Liste der Änderungen im Versionsmanagementsystem, die letzten Zeilen der Build-Ausgabe, fehlgeschlagene Tests, verwendete Subversion-Revision, URL des Builds, Umgebungsvariablen usw. Eine vollständige Liste der unterstützten Content Tokens können Sie der Online-Hilfe des Plugins entnehmen (Sie müssen dazu das Plugin also installieren).

Betreff und Inhalt der Emails anpassen

Sie können den Inhalt der E-Mails auch mit HTML-Elementen anreichern, um eine strukturiertere Darstellung zu erzielen. Eine beispielhafte Konfiguration mit zugehörigem Resultat in einem E-Mail-Programm sehen Sie in Abbildung 7–22 und Abbildung 7–23.

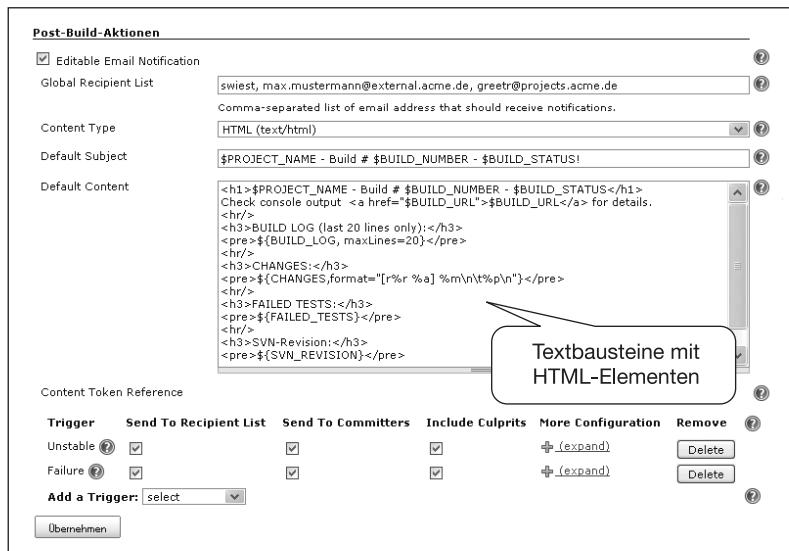


Abb. 7–22

Plugin »Email-ext« – Konfiguration einer HTML-E-Mail

Abb. 7-23

Plugin »Email-ext« –
Ergebnis im E-Mail-
Programm (Ausschnitt)

From: Hudson Admin <swiest@buildbox.simonwiest.de>
To: swiest@buildbox.simonwiest.de, swiest@buildbox.simonwiest.de
Subject: greetr - Build # 61 - Unstable!
Date: Sat, 27 Mar 2010 16:06:23 +0100 (CET)

greetr - Build # 61 - Unstable

Check console output <http://ubuntu:8082/job/greetr/61/> for details.

BUILD LOG (last 20 lines only):

```
[...truncated 210 lines...]
[INFO] Generating "Source Xref" report.
log4j:WARN No appenders could be found for logger (org.apache.commons.digester.Digester.sax).
log4j:WARN Please initialize the log4j system properly.
[FINDBUGS] Successfully parsed file /data/hudson/jobs/greetr/workspace/greetr/target/findbugsXml.xml
[CHECKSTYLE] Successfully parsed file /data/hudson/jobs/greetr/workspace/greetr/target/checkstyle-report.xml
[PMD] Successfully parsed file /data/hudson/jobs/greetr/workspace/greetr/target/pmd.xml of module greetr
[HUDSON] Archiving site from /data/hudson/jobs/greetr/workspace/greetr/target/site to /data/hudson/jobs/greetr/...
[INFO] -----
[INFO] BUILD SUCCESSFUL
[INFO] -----
[INFO] Total time: 31 seconds
[INFO] Finished at: Sat Mar 27 16:06:23 CET 2010
[INFO] Final Memory: 62M/148M
[INFO] -----
channel stopped
Started indexing related Jira issue keys
Successfully indexed related Jira issue keys
Email was triggered for: Unstable
Sending email for trigger: Unstable
```

CHANGES:

```
[r103 Simon Wiest] Extended unit testing for international greetings.
    /src/test/java/de/simonwiest/greetr/GreetrTest.java
```

FAILED TESTS:

```
1 tests failed.
REGRESSION: de.simonwiest.greetr.GreetrTest.testComposeGreetingRussian
```

Obwohl die zahlreichen Einstellungsmöglichkeiten dazu verlocken, sollten Sie in der Praxis darauf achten, so wenig E-Mails wie möglich abzusetzen. Bedenken Sie: Aus großer Macht wächst große Verantwortung ...

7.6.3 RSS-Feeds

Statt sich E-Mails schicken zu lassen, können Sie auch den Status Ihrer Builds per RSS-Feeds (*Really Simple Syndication*) abfragen. Dazu können Sie eigenständige RSS-Reader verwenden. Die meisten E-Mail-Programme und Webbrower stellen aber inzwischen ebenfalls RSS-Feeds dar. Hudson bietet auf der Übersichtsseite drei Feeds an: alle Builds, nur Fehlschläge oder nur die jeweils letzten Builds aller Projekte. Zusätzlich können Sie auf allen Projektseiten je zwei projektbezogene Feeds abonnieren (*Alle Builds, nur Fehlschläge*). RSS-Feeds können auch elegant zur Anbindung von Drittsystemen verwendet werden. So bezieht der Hudson Build Monitor (Abschnitt 7.6.4) seine Informationen beispielsweise per RSS-Feed.

7.6.4 Hudson Build Monitor (Firefox Add-on)

Der »Hudson Build Monitor« ist ausnahmsweise kein Plugin für Hudson, sondern ein Add-on für den beliebten Webbrowser Firefox. Das Add-on zeigt die Aktivität eines Hudson-Systems in der Statusleiste von Firefox an (Abb. 7-24). Überstreichen Sie dort die jeweiligen Projekte mit dem Mauszeiger, bekommen Sie zusätzlich deren letzte Builds bzw. die momentane Auslastung Ihres Hudson-Systems angezeigt. Bei Veränderungen können Sie zusätzlich durch ein visuelles und akustisches Signal darauf hingewiesen werden.

Technisch ist die Kommunikation mit Hudson durch die regelmäßige Abfrage der relevanten RSS-Feeds realisiert. Folgerichtig erweitert das Firefox-Add-On Kontextmenüs des Browsers beim Anklicken von RSS-Links um den Eintrag »Add link to Hudson Build Monitor«. Sie müssen dann nur noch den gewünschten Klartextnamen angeben, der in der Statuszeile angezeigt werden soll – und schon sind Sie auch während Ihrer Surf-Sitzungen allzeit informiert.

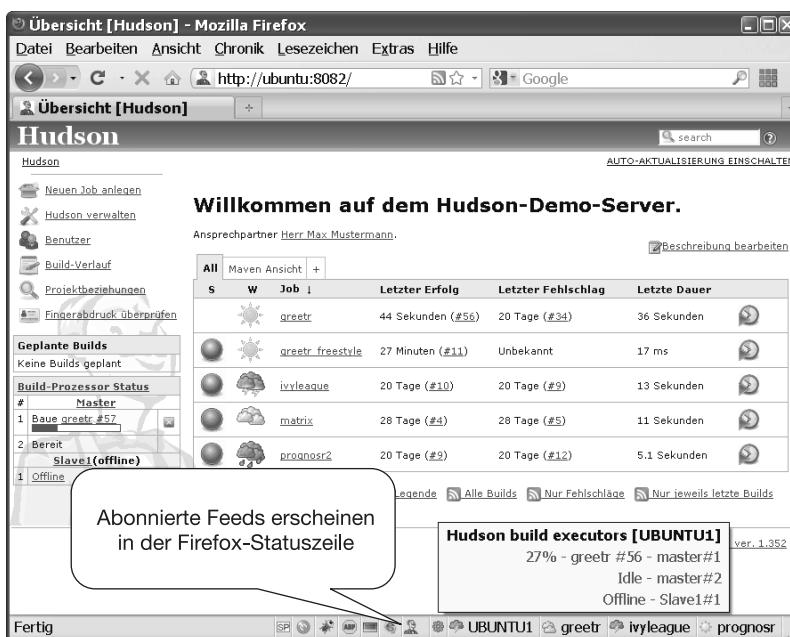


Abb. 7-24

Hudson Build Monitor
(Firefox Add-On)

7.6.5 Twitter (Plugin)

Über Sinn oder Unsinn mag man sich streiten, aber Hudson kann auch twittern – die Installation des Plugins »Twitter« vorausgesetzt. Jede Nachricht (*tweet*) enthält dann das Build-Ergebnis, die Build-Nummer

und den Namen des Projekts. Darüber hinaus kann optional auch noch ein Link zum Build in der Weboberfläche des Hudson-Servers angehängt werden. Diese URL wird vorher – aufgrund der Längenbeschränkung auf 140 Zeichen pro Tweet – noch automatisch über den Dienst tinyurl.com in eine Kurz-URL umgewandelt.

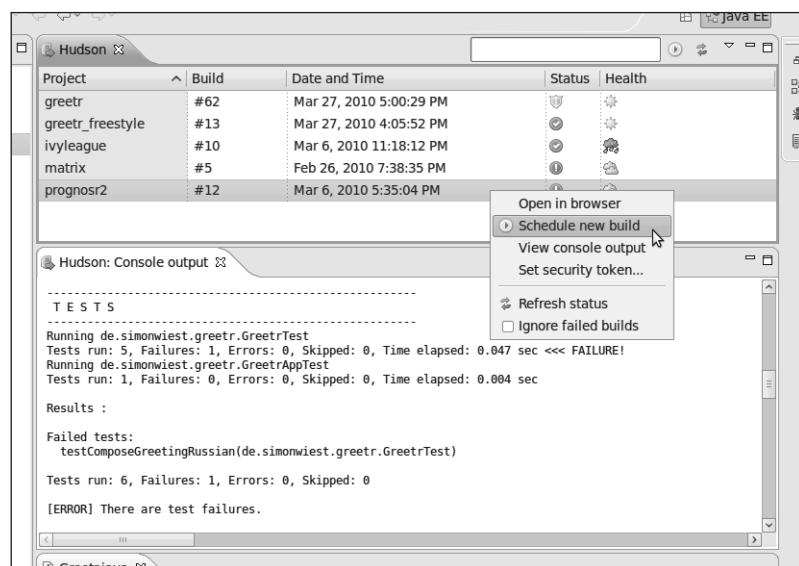
7.6.6 hudson-eclipse (Eclipse-Plugin)

Entwickler, die mit Eclipse arbeiten und ungern ihre IDE verlassen, können sich ein Hudson-Eclipse-Plugin installieren (<http://code.google.com/p/hudson-eclipse>). So erhalten sie eine spezielle Hudson-Ansicht, welche alle angelegten Projekte und deren aktuellen Status zeigt (Abb. 7–25). Per Kontextmenü können unter anderem neue Builds gestartet, Build-Logs eingesehen oder es kann zur korrespondierenden Projektseite in der Hudson-Weboberfläche gesprungen werden.

Momentan ist allerdings der Funktionsumfang des Eclipse-Plugins einfacher mit einem parallel geöffneten Browserfenster zu erzielen. Lediglich bei Hudson-Systemen mit einer großen Anzahl an Projekten (über 100) könnte die inkrementelle Filterfunktion über alle Projektnamen einen kleinen Effizienzgewinn erzielen. Eine weitergehende Integration, wie das Anspringen fehlgeschlagener Unit-Tests oder das Verfolgen von Build-Logs in Echtzeit, sucht man jedoch leider noch vergebens.

Abb. 7–25

Hudson-Eclipse-Plugin



7.6.7 Informationsradiatoren

Neben den Nachrichten, die ein CI-System individuell an seine Benutzer verschiickt, können Informationsradiatoren (*information radiators*) wertvolle Dienste leisten. Der Begriff wurde von Alistair Cockburn geprägt [Cockburn01] und beschreibt eine Visualisierung kritischer Betriebsgrößen,

Sinnvolle Ergänzung zu individuellen Nachrichten

- die ständig aktualisiert wird und
- die man blitzschnell im Vorübergehen wahrnehmen kann.

Darüber hinaus würde ich aus meinen Praxiserfahrungen noch ein drittes Kriterium für einen wirksamen Radiator hinzufügen:

- Der Betrachter muss sofort wissen, ob er *ganz persönlich* handeln muss oder nicht. Es geht also nicht darum, möglichst viele Informationen anzugeben (»das Unternehmen auf einen Blick«), sondern bei Problemen möglichst schnell Gegenmaßnahmen auszulösen.

Ein Informationsradiator kann eine schlichte Pinnwand mit bunten Kärtchen sein, aber auch ein großformatiges Display, das in Echtzeit Informationen aus IT-Systemen anzeigt. Ein wichtiger Erfolgsfaktor für einen guten Informationsradiator ist der richtige Standort: Hier eignen sich vor allem Plätze mit hohem Verkehr, also der Empfang, Abteilungsflure oder die Kaffeeküche. Die grafische Ausgabe auf einem solchen Display muss natürlich sehr plakativ gestaltet sein. Der Betrachter muss blitzschnell aus dem Augenwinkel heraus entscheiden können, ob alles im sprichwörtlichen »grünen« Bereich ist oder ein Problem seine Aufmerksamkeit benötigt. Tiefergehende Analysen wird der aus der Kaffeeküche an seinen Rechner zurückhastende Entwickler lieber dort vornehmen.

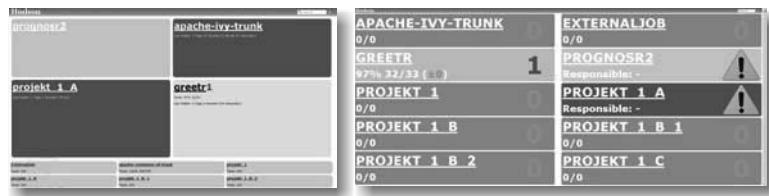
Technisch einfach oder komplex: Erlaubt ist, was funktioniert.

Die Statusdaten eines Hudson-Systems lassen sich über die REST-API leicht anzapfen und mit wenig Aufwand in hübsche HTML-Darstellungen verpacken. Anregungen können Sie sich bei den beiden Plugins »Radiator View« und »eXtreme Feedback Panel« holen (Abb. 7–26). Diese Plugins rüsten jeweils einen eigenen Ansichtstyp (*view type*) nach. Um zur Darstellung des Informationsradiators zu gelangen, legen Sie eine neue Ansicht (*view*) an. Dies geschieht auf der Hudson-Übersichtsseite durch Anklicken des »+«-Reiters oberhalb der Job-Liste (oder durch direkte Eingabe der URL <http://<ihr.hudson.server>/newView>). Wie bei jeder Ansicht können Sie auch hier auswählen, welche Jobs enthalten sein sollen. Dadurch können Sie auf einfache Weise eine individuelle Abteilungsansicht zusammenstellen, welche nur die Jobs Ihres Teams enthält. Abschließend rufen Sie die neue Ansicht auf und schalten den Browser in den Vollbildmodus um.

Plugins »Radiator View« und »eXtreme Feedback Panel«

Abb. 7-26

Plugins »Radiator View«
 (links) und »eXtreme
 Feedback Panel« (rechts)



7.6.8 eXtreme-Feedback-Geräte anbinden

XFDs: Mehr als eine
 »Nerd-Spielerei«

Ähnlich wie Informationsradiatoren signalisieren *eXtreme Feedback Devices* (XFD) den Systemzustand des CI-Servers. Allerdings wird hier die Information noch weiter verdichtet und meist sogar binär angezeigt: »Alles o.k.« oder »Problem aufgetreten«. Dies muss kein Nachteil denn. Letzten Endes geht es bei XFDs darum, allen Beteiligten schnell mitzuteilen, ob Handlungsbedarf besteht oder nicht. Zur Steigerung des Spaßfaktors wird der Status des CI-Systems mit Gegenständen aus der realen Welt angezeigt. XFDs sind dabei jedoch weit mehr als eine »Nerd-Spielerei«:

- Sie schaffen eine Kultur gemeinsamer Verantwortung für den fehlerfreien Build.
- Sie sind bei Kundenbesuch ein ideales Konversationsstück (»Das ist unser Qualitätsmanagement-Tool«).
- Sie ermöglichen es auch nicht unmittelbar Beteiligten, am Geschehen teilzunehmen – vom Geschäftsführer bis zum Reinigungspersonal.

Im Hudson-Wiki² finden Sie dazu Anregungen mit Bildern, Filmen und Code, beispielsweise:

- *Lava-Lampen:*

Der Klassiker unter den XFDs dürfte ein Paar von Lava-Lampen sein, die über den CI-Server geschaltet werden. Die grüne Lava-Lampe leuchtet, solange sich alle Projekte im buchstäblich »grünen Bereich« befinden. Schlägt hingegen ein Build fehl, wird die grüne Lavalampe ausgeschaltet und dafür die rote eingeschaltet. Da eine Lava-Lampe ein paar Minuten benötigt, um das enthaltene Wachs richtig in Wallung zu bringen, hat man zusätzlich einen visuellen Indikator, ob ein Farbwechsel erst vor kurzem stattgefunden hat.

2. <http://wiki.hudson-ci.org/display/HUDSON/Use+Hudson>

■ *Verkehrsampel:*

Andere Softwareteams lassen sich Ihren Hudson-Status auf einer echten Verkehrsampel visualisieren. Die Ampeln wurden dabei offiziell im Internet ersteigert. Hoffentlich stimmt's ...

■ *Nabaztag:*

Das kybernetische Häschen »Nabaztag« kann nicht nur blinken und mit den Ohren wackeln – per Text-to-Speech liest es sogar Meldungen in verschiedenen Landessprachen und Stimmen vor. Entwickler scheinen eine hohe Affinität zu dem Häschen zu haben, denn es gibt ein ausgefeiltes Hudson-Plugin zur Anbindung des Langohrs, über das sich Beginn und Ausgang von Builds signalisieren lassen. Bei Fehlschlägen lässt Nabaztag die Ohren hängen – stellt sie aber auch wieder auf, wenn alles in Ordnung gebracht wurde. Leider ist es seit Ende 2009 ruhiger um den Hersteller violet.net geworden, so dass die Verfügbarkeit neuer Geräte momentan unklar ist.

■ *Duftlampen:*

Ein sehr ausgefallenes XFD besteht aus einem Paar von elektrisch betriebenen Duftlampen. Jeweils mit unterschiedlichen Duftölen bestückt, verbreiten die Lampen nach Aufheizphasen von wenigen Minuten den passenden »code smell«.

■ *Die Bären-Ampel:*

Eine Variation der Ampel sind Leuchtbären, wie man sie aus Kinderzimmern der 90er Jahre kennt. Es gibt sie in vielen Farben, so dass sich aus drei Bären eine Ampel zusammenstellen lässt. Der grüne Bär beispielsweise steht für »Alle Projekte o.k.«, der rote hingegen für »Mindestens ein Projekt ist fehlgeschlagen.« Der gelbe Bär signalisiert, dass gerade ein Projekt gebaut wird, und dient somit als vereinfachte Aktivitätsanzeige des Hudson-Servers. Neue Bären sind zurzeit nur in Fachgeschäften für Designer-Leuchten zu kaufen. Eine größere Auswahl an Farben und Modellen findet man zu wesentlich niedrigeren Preisen in den gängigen Online-Auktionshäusern.

Die Ansteuerung von XFDs lässt sich am einfachsten durch sogenannte IP-Steckdosen realisieren. Dabei handelt es sich um Steckdosenleisten mit eingebautem Webserver und eigener IP-Adresse. Jede Steckdose kann dann über TCP/IP an- bzw. ausgeschaltet werden. Die meisten IP-Steckdosen bieten dazu eine einfache CGI-Schnittstelle an, über die beispielsweise die Steckdose Nr. 3 über den Aufruf `http://<steckdosenleiste.acme.de>/?F3=0` an- bzw. mit `http://<steckdosenleiste.acme.de>/?F3=1` ausgeschaltet werden kann. Mehr kon-

Ansteuerung per
IP-Steckdosenleiste

krete Tipps zum Thema »Ansteuerung von XFDs« finden Sie in der oben angegebenen Stelle im Hudson-Wiki.

*Auch bei XFDs:
Weniger ist mehr*

Noch ein Hinweis aus der Praxis: Rotierende Alarmlichter, Schiffs sirenen und USB-Raketenwerfer machen zweifelsohne mächtigen Eindruck. Im Alltag haben sich jedoch die dezenteren Signalgeber als wesentlich praxistauglicher herausgestellt. Stellen Sie sich vor, Ihr Build-System hat ein *wirklich* schwerwiegendes Problem entdeckt. Ein penetrantes Warn-Blitzlicht mit schriller Alarm wäre vermutlich so ziemlich das Letzte, was einem Entwickler bei der hochkonzentrierten Fehlersuche helfen würde ...

7.7 Dokumentationswerkzeuge integrieren

Neben ausführbarer Software wird während eines Builds oftmals auch Dokumentation erzeugt. Die vollautomatische Erstellung hat den Vorteil, dass die Dokumentation immer aktuell ist, exakt zu der jeweils erstellten Software passt und kostengünstig produziert werden kann.

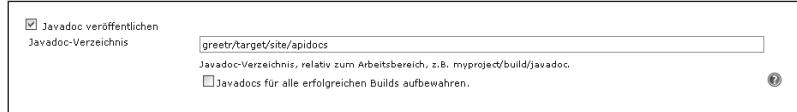
7.7.1 Javadoc

Im Java-Umfeld stellt Javadoc (<http://java.sun.com/j2se/javadoc>) die gängigste Dokumentationsform dar. Wenn Sie während des Builds »Javadocs« erstellen, z.B. über den Ant-Task javadoc oder das Maven-Goal javadoc:jar, müssen Sie in einem Hudson-Free-Style-Projekt lediglich unter *Post-Build-Aktionen* → *Javadocs veröffentlichen* das Wurzelverzeichnis der erstellten Dokumentation angeben (Abb. 7-27). Dies bewirkt zweierlei:

- Hudson kopiert automatisch nach Abschluss jedes Builds die Javadoc-Dateien in das Verzeichnis <Arbeitsbereich>/javadoc. Dort liegen also immer nur die Javadocs des jeweils letzten Builds. Optional können Sie auch die Javadocs aller erfolgreichen Builds konservieren, was allerdings mehr Speicherplatz verbraucht.
- Auf der Projektseite erscheint ein neues Symbol *Javadocs* mit einem Link auf die Dokumentation. Sie können dadurch also direkt vom Projekt in die zugehörige Dokumentation springen.

Abb. 7-27

Einbindung von Javadoc-API-Dokumentationen



Sie können die Option *Veröffentliche Javadocs* auch für andere Dokumentationswerkzeuge als Javadoc verwenden. Solange die komplette Dokumentation in einem Verzeichnis erzeugt wird, geben Sie einfach dieses an. Das Symbol auf der Projektseite wird dann von Hudson automatisch von *Javadocs in Dokumentation* umbenannt.

Javadocs enthalten von Haus aus keine UML-Diagramme. Wenn Sie Ihre Dokumentation mit Schaubildern anreichern möchten, sei Ihnen das Open-Source-Projekt UMLGraph (<http://www.umlgraph.org>) empfohlen. Über ein eigenes Doclet erstellt es vollautomatisch ansehnliche UML-Klassendiagramme und fügt diese an der passenden Stelle in die bestehenden Javadoc-Seiten ein (Abb. 7–28).

UMLGraph

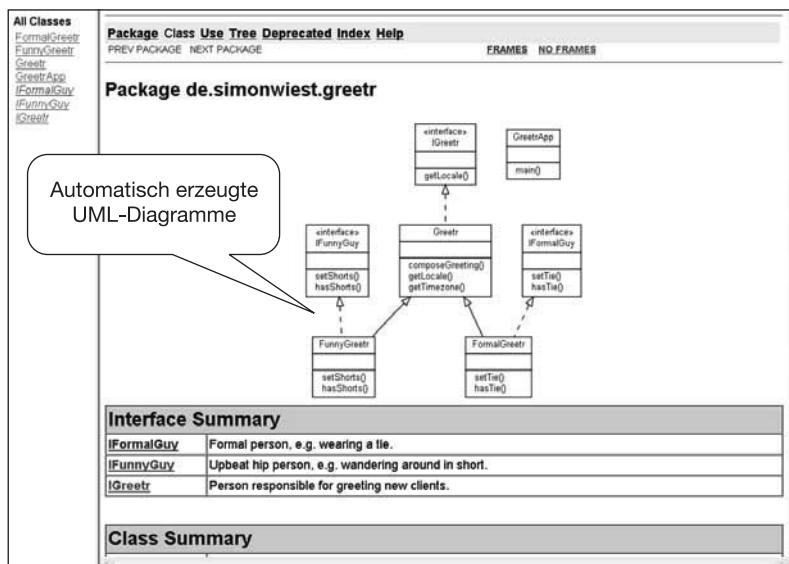


Abb. 7–28

Javadoc-API-Dokumentation mit UML-Diagrammen

Technisch betrachtet baut UMLGraph auf die Bibliothek GraphViz (<http://www.graphviz.org>) auf, die es daher als installiert voraussetzt. Verwenden Sie Maven, so können Sie in der POM-Datei die Erstellung von UML-Diagrammen als Option des maven-javadoc-Plugins aktivieren:

```

<reporting>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-javadoc-plugin</artifactId>
      <configuration>
        <doclet>org.umlgraph.doclet.UmlGraphDoc</doclet>
        <docletArtifact>
          <groupId>org.umlgraph</groupId>
  
```

Listing 7–6

Konfiguration von UMLGraph in der POM-Datei

```
<artifactId>doclet</artifactId>
<version>5.1</version>
</docletArtifact>
<additionalparam>-operations</additionalparam>
<useStandardDocletOptions>true</useStandardDocletOptions>
</configuration>
</plugin>
[...]
</plugins>
</reporting>
```

Die Generierung der Diagramme erfordert allerdings einiges an Rechenzeit, so dass Sie prüfen sollten, ob der Nutzen der Schaubilder die Verlängerung der Build-Zeit rechtfertigt. Vielleicht können Sie ja auch eine langlaufende Generierung der Dokumentation in ein eigenständiges Projekt auslagern, das mit niedriger Frequenz gebaut wird, etwa einmal nachts.

7.7.2 Maven Site

Wenn Sie Mavens Dokumentationsmechanismus `site` verwenden, gehen Sie für Free-Style-Projekte wie unter Abschnitt 7.7.1 (Javadoc) vor. Statt des Wurzelverzeichnisses der erzeugten »Javadocs« geben Sie das Site-Verzeichnis an (üblicherweise `<Projekt>/target/site`).

In Maven-2-Projekten hingegen müssen Sie nichts zusätzlich konfigurieren. Hudson detektiert automatisch die Ausführung des Maven-Goals `site` und fügt ein Symbol »Maven Site« mit hinterlegtem Link in die Projektübersicht ein.

7.7.3 Einbindung beliebiger HTML-Berichte

Einbindung von HTML-Berichten

Es gibt noch eine ganze Reihe weiterer nützlicher Dokumentationswerkzeuge, die sich sehr gut in einen CI-Build einbauen lassen.

Für viele ist noch kein spezielles Hudson-Plugin zur Visualisierung verfügbar. In den meisten Fällen können diese Werkzeuge jedoch HTML-Berichte ausgeben. Somit reduziert die Anbindung auf die Frage, wie diese Berichte am elegantesten von der Hudson-Oberfläche aus zu erreichen wären. Dazu haben Sie mindestens zwei Möglichkeiten: Entweder fügen Sie HTML-Links in die Projektbeschreibung ein, oder Sie verwenden das Plugin »Doc-Links«. Abbildung 7–29 zeigt beide Möglichkeiten gleichzeitig.

The screenshot shows the Hudson interface for the 'greetr' project. On the left, there's a sidebar with various project management links like 'Zurück zur Übersicht', 'Status', 'Änderungen', 'Arbeitsbereich', 'Jetzt bauen', 'Wählen', 'Konfigurieren', 'Checkliste Warnings', 'Analysen', 'Nachschauen', and 'Jazzbeam'. The main content area has a header 'Projekt greetr' with a placeholder text about Latin. Below it, there's a 'Verfügbare Dokumentationen' section with a 'Classische Auszeichnungen' tab selected. A callout box labeled 'HTML-Link in Beschreibung' points to the 'Schema Spy-Dokumentation' link. To the right, there's a 'Tabelle Codeanalyse Trend' with a chart showing trends over time. At the bottom, there's a 'Document links' section with a 'Permalinks' heading and a list of links, with another callout box labeled 'Plugin Doc-links' pointing to it.

Abb. 7-29
Einbindung von
HTML-Berichten

Der einfachste Weg, HTML-Berichte in Hudsons Weboberfläche einzubinden, führt über die Projektbeschreibung, die am Kopf der Übersichtsseite eines Projekts dargestellt wird. Diese Beschreibung kann nicht nur Text, sondern auch HTML-Auszeichnungen enthalten. Dadurch lassen sich Beschreibungen typografisch ansprechender gestalten. Vor allem aber sind dadurch auch anklickbare Links möglich. Links gelten dabei immer relativ zum Projektverzeichnis, in dem ja auch der Arbeitsbereich des Projekts mit den erstellten Berichten liegt. Die Verknüpfung `Schema Spy-Dokumentation` führt also zur Datei `<Arbeitsbereich>/target/schemaspy/index.html` (das Kürzel ws in der URL wird intern auf das Unterverzeichnis workspace im Jobverzeichnis abgebildet).

Diese Methode funktioniert ad hoc und kommt ohne zusätzliche Plugins aus. Allerdings wird vorausgesetzt, dass der Benutzer auf die Dateien des Arbeitsbereichs direkt zugreifen darf. Dieses Zugriffrecht kann ihm in abgesicherten Hudson-Instanzen eventuell aus Sicherheitsgründen fehlen.

Wenn Sie mehrere Dokumentationswerkzeuge einsetzen, könnte das Plugin »Doc-Links« interessant sein: Dieses Plugin fügt der Projektübersicht einen neuen Abschnitt *Document links* hinzu, der Verweise auf die jeweiligen Berichte aufführt. Welche Links hier zu sehen sein sollen, geben Sie in der Projektkonfiguration im Abschnitt *Post-Build-Aktionen → Publish documents* an. Im Gegensatz zu HTML-Links in der Projektbeschreibung funktionieren die *Document links* auch mit abgesichertem Arbeitsbereich, da das Plugin am Ende eines Builds die konfigurierten Dokumentationen in ein eigenes Verzeichnis

HTML-Links in der Projektbeschreibung

Plugin »Doc-Links«

doclinks außerhalb des Arbeitsbereiches kopiert (ganz ähnlich wie Hudson dies für Javadocs realisiert).

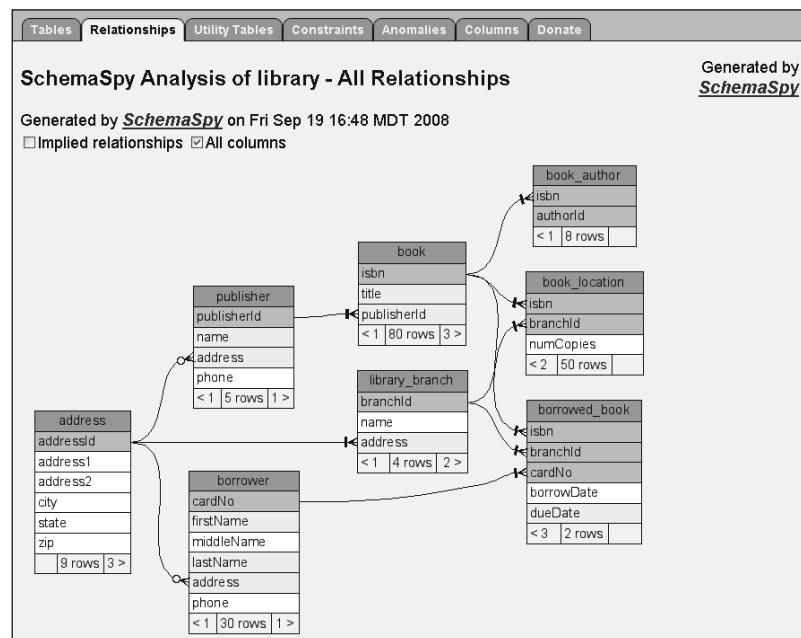
Anregungen für weitere Dokumentationswerkzeuge

SchemaSpy

Abschließend ein paar Anregungen für nützliche Dokumentationswerkzeuge, die momentan noch ohne eigene Hudson-Plugin-Anbindung auskommen müssen und daher am besten mit dem oben beschriebenen Plugin »Doc-Links« eingebunden werden:

SchemaSpy (<http://schemaspy.sourceforge.net>) dokumentiert vollautomatisch Datenbankschemas durch Auslesen der Metadaten einer laufenden Datenbankinstanz. Die Dokumentation umfasst nicht nur eine übersichtliche Auflistung aller Datenbankobjekte (Tabellen, Felder, Indizes usw.), sondern auch interaktive Entitätsdiagramme, die das komplette Schema auf einen Blick zeigen (Abb. 7–30).

Abb. 7–30
Automatisch mit
SchemaSpy erstelltes
Entitätsdiagramm



JDepend, Classycle

JDepend (<http://clarkware.com/software/JDepend.html>) und Classycle (<http://classycle.sourceforge.net>) untersuchen die Abhängigkeiten von Klassen und Java-Packages untereinander und finden beispielsweise zyklische Abhängigkeiten, die eine klare Modularisierung eines umfangreichen Softwareprojekts deutlich erschweren.

Depdometer

Depdometer (<http://source.valtech.com/display/dpm>) dient der Architekturvalidierung und kann Programmcode auf Einhaltung einer angestrebten Architektur überprüfen. Beispielsweise können so unerwünschte »Abkürzungen« über mehrere Ebenen einer Schichtenarchitektur hinweg entdeckt werden.

7.8 Analysewerkzeuge integrieren

Bisher haben wir Testverfahren angesprochen, die Code ausführen, um dessen Funktionstauglichkeit zu überprüfen, wie etwa JUnit oder TestNG.

Daneben existiert eine weitere Gattung an Qualitätssicherungswerkzeugen, welche auf den Quelltexten bzw. den daraus kompilierten Binärdateien arbeitet. Diese Werkzeuge werden *statische* Analyseverfahren genannt, weil sie den untersuchten Code nicht ausführen. Neben einfachen Prüfungen, etwa nach der Einhaltung von Code-Konventionen, können diese Werkzeuge auch mit einem überraschend tiefen Verständnis problematische Stellen in Quelltexten finden, etwa mögliche Probleme bei nebenläufiger Ausführung. Stellen Sie sich einfach vor, Sie hätten einen Senior-Entwickler auf Abruf, der Ihnen kostenlos, in Windeseile und ohne zu ermüden Ihre Quelltexte auf mehrere hundert typische Programmierfehler abklopft.

Statische Codeanalyse

Wie bei JUnit und Konsorten ist es auch bei diesen Werkzeugen Aufgabe des Builds, die eigentliche Analyse durchzuführen. Hudson hingegen sammelt nach Abschluss des Builds die Ergebnisse ein, archiviert diese mit den anderen Build-Resultaten und visualisiert sie über die Weboberfläche. Wenn also von einem »FindBugs-Plugin für Hudson« die Rede ist, so müsste man eigentlich treffender von einem »FindBugs-Visualisierungs-Plugin für Hudson« sprechen. Diese Unterscheidung ist wichtig, da immer wieder Hudson-Einsteiger erwartungsfroh Codeanalyse-Plugins installieren, sich aber anschließend wundern, warum Hudson diese Analysen nicht automatisch *ausführt*.

7.8.1 Checkstyle, PMD, FindBugs

Checkstyle, PMD und FindBugs gehören zu den bekanntesten statischen Analysewerkzeugen in der Java-Welt. Die drei Werkzeuge dekken jeweils einen unterschiedlichen Themenbereich aber, haben aber auch Schnittmengen. Alle drei können interaktiv in den gängigen IDEs eingesetzt werden, funktionieren aber auch selbstständig als Kommandozeilenanwendung, Ant-Task oder Maven-Goal. Damit eignen sie sich hervorragend zur Einbindung in Continuous-Integration-Builds.

Wer zum ersten Mal in Kontakt mit statischer Codeanalyse kommt und mal eben sein aktuelles Projekt überprüfen lässt, erhält nicht selten eine drei- bis vierstellige Anzahl an Warnungen. Viele Entwickler schalten an diesem Punkt leider ab. Für die Einführung statischer Codeanalyse als Routineprüfung im Rahmen vom CI-Builds kann es daher helfen, zunächst mit einem sehr laxen Regelwerk zu

beginnen, um dann sukzessive die Latte höher und höher zu legen. Oft entspinnen sich auch Diskussionen über Sinn und Unsinn der einzelnen Regeln. Wird in diesen Diskussionen fundiert argumentiert und abgewogen, kann bereits dieser Prozess ein Gewinn für ein Team sein, weil das »Warum?« hinter der Regel abgeklopft wird. Auf keinen Fall sollten Sie »lästige« Regeln vorschnell deaktivieren – sie sind nicht ohne Grund in die genannten Werkzeuge aufgenommen worden. Idealerweise dokumentieren Sie mit Kommentaren in den Konfigurationsdateien der jeweiligen Werkzeuge, warum die eine oder andere Regel deaktiviert wurde.

Checkstyle

Checkstyle (<http://checkstyle.sourceforge.net>) überprüft – wie der Name es andeutet – Quelltexte auf rund 150 stilistische Aspekte, etwa Einrückungen, ausreichende Kommentierung oder das Vorhandensein von vorgeschriebenen Copyright-Kopfzeilen. Checkstyle ist hochgradig konfigurierbar und wird mit einer Beispielkonfiguration ausgeliefert, welche die Code-Konventionen von Sun abbildet. Ausgehend von diesem Startpunkt lassen sich auf einfache Weise laxere oder schärfere hausinterne Konventionen definieren. Ursprünglich wurde Checkstyle entwickelt, um Code auf ein einheitliches Layout zu überprüfen. Inzwischen kann Checkstyle deutlich mehr, etwa Code-Doubletten aufspüren oder problematisches Klassen-Design erkennen.

PMD

Auch PMD (<http://pmd.sourceforge.net>) unterstützt das Einhalten von Quellcode-Konventionen, etwa der konsistenten Benennung von Klassen, Methoden und Feldern, und überlappt so mit Checkstyle. PMDs Schwerpunkt hingegen dürfte im Auffinden von trügerischen Java-Code-Fragmenten sein, die entweder bereits Fehler beinhalteten oder aber einen unachtsamen Entwickler zu fehlerhaftem Code verleiten könnten. Bei den über 250 Fehlermustern, die überprüft werden, dürfte es sich um die geronnene Erfahrung aus unzähligen Nachschichten handeln, in denen obskure Fehler gejagt werden mussten. Der Name »PMD« ist übrigens nach Aussagen der Entwickler völlig sinnfrei und wurde des guten Klanges wegen gewählt. Nachträgliche Auslegungen durch Anwender wie »Project Mess Detector« oder »Project Meets Deadline« mögen einen Hinweis auf die große Nützlichkeit und den guten Ruf des Werkzeugs geben.

FindBugs

FindBugs (<http://findbugs.sourceforge.net>) unterscheidet sich von den beiden bereits angesprochenen Werkzeugen in inhaltlicher und technischer Sicht. FindBugs sucht nach über typischen 400 Fehlermustern, die aus der falschen Verwendung von schwierigen Java-Sprachkonstrukten oder APIs herrühren. Darüber hinaus findet es auch die »alltäglichen Unfälle«, wie die Verwechslung von Variablen, den falschen Einsatz von booleschen Vergleichen oder Ausgabekanäle, die nicht geschlossen werden. FindBugs erstaunt immer wieder, wenn es

unscheinbare, aber problematische Passagen im Code zutage fördert. Oft kann man in diesen Fällen bereits durch das Studium der ausführlichen Fehlerbeschreibung einiges dazulernen. Technisch unterscheidet sich FindBugs von PMD und Checkstyle, da es nicht auf Quelltexten, sondern auf den kompilierten .class-Dateien arbeitet. Es lässt sich also sogar auch Java-Code überprüfen, zu dem kein Quelltext vorhanden ist.

Ein Hinweis für Maven-Benutzer: Das Hudson-Plugin für FindBugs benötigt einen FindBugs-Bericht im XML-Format. Da das findbugs-maven-plugin per Default HTML liefert, müssen Sie den Formatwunsch in Ihrer POM-Datei explizit angeben:

```
<reporting>
  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>findbugs-maven-plugin</artifactId>
      <configuration>
        <findbugsXmlOutput>true</findbugsXmlOutput>
      </configuration>
    </plugin>
    [...]
  </plugins>
</reporting>
```

Listing 7-7
Konfiguration des
findbugs-maven-plugin
in der POM-Datei

Die große Popularität der drei vorgestellten Werkzeuge und die exzellenten Visualisierungsmöglichkeiten der Analyseergebnisse haben die Hudson-Plugins »Checkstyle«, »PMD« und »FindBugs« zu den drei meistinstallierten Hudson-Erweiterungen gemacht. Sie bieten nicht nur die Anzeige der gefundenen »Problemzonen« pro Build, sondern auch Zusammenfassungen nach Ort (Package, Klasse), Fehlerart oder Priorität. Darüber hinaus werden auch Veränderungen zwischen den Builds berechnet, also ob die Quelltexte an Codequalität zu- oder abnehmen. Zusätzlich ist es möglich, sich direkt in Hudson die betreffenden Stellen hervorgehoben im Quelltext zeigen zu lassen – inklusive der kompletten Fehlerbeschreibung des jeweiligen Analysewerkzeugs. Ihre Qualitätsziele können Sie durch die Angabe der maximal erlaubten Warnungen pro Wetterlage formulieren (0 Warnungen entsprechen »sonnig«, 10 Warnungen »bewölkt« usw.). Bei Überschreiten eines selbst gewählten Maximalwerts können Sie den Build als instabil markieren lassen (gelber Ampelball) und so manuellen Eingriff verlangen. Zusammenfassungen wie *Neuer Rekord: Seit 45 Tagen keine FindBugs-Warnung mehr gefunden!* sollen Entwickler zusätzlich mit einem kleinen Augenzwinkern zu sauberem Code anspornen. Die Beispiele in Abbildung 7-31 zeigen exemplarisch die Möglichkeiten des Checkstyle-Plugins. Die Darstellung erfolgt analog für FindBugs, PMD und eine Reihe weiterer Analysewerkzeuge.

Visualisierung in Hudson

Abb. 7-31

Visualisierung der Checkstyle-Analyse für ein Projekt (oben), einen Build (Mitte) und eine Klasse (unten)

Hudson

Projekt greetr

Entwicklung der CheckStyle-Analysen

Hudson

CheckStyle Ergebnis

Vergleich mit letzter Analyse

Problemstellen, nach Aspekten gegliedert

Hudson

Inhalt der Datei Greetr.java

Problemstellen im Quelltext hervorgehoben

The screenshots illustrate the Hudson interface for a project named "greetr". The top screenshot shows a "Checkstyle Trend" chart with a bar graph titled "Checkstyle Trend" and a line graph titled "Trend der Testergebnisse". A callout bubble points to the top chart with the text "Entwicklung der CheckStyle-Analysen". The middle screenshot shows a "CheckStyle Ergebnis" page with a table comparing "Alle Warnungen", "Neue Warnungen", and "Behobene Warnungen". A callout bubble points to the table with the text "Problemstellen, nach Aspekten gegliedert". The bottom screenshot shows the "Inhalt der Datei Greetr.java" page with code snippets and annotations. A callout bubble points to the annotated code with the text "Problemstellen im Quelltext hervorgehoben". The code snippets show Java code for a "Greetr" class with annotations like "Checkstyle: Verborgenes Feld", "Checkstyle: Verborgene Variable", and "Checkstyle: Verborgener Parameter".

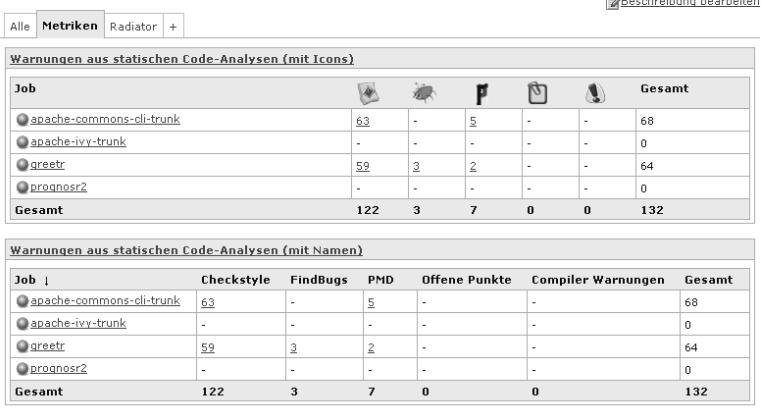
Da alle drei Werkzeuge vom selben Autor stammen und thematisch Verwandtes leisten, verwenden sie intern denselben Kern, der als eigenes Plugin »Static Analysis Utilities« gekapselt ist – man könnte daher sogar von einer Plugin-Familie sprechen. Der Plugin-Manager installiert beim Herunterladen der obigen Plugins die »Static Analysis Utilities« als Vorbedingung automatisch mit. Aus Benutzersicht ist diese gemeinsame Basis ein immenser Vorteil, weil die Bedienung aller darauf basierender Visualisierungen ähnlich abläuft.

Setzen Sie mehrere Plugins aus der Familie der »Static Analysis Utilities« ein, sollten Sie zusätzlich einen Blick auf das Plugin »Static Analysis Collector« werfen. Dieses Plugin kombiniert die Resultate der einzelnen Werkzeuge in aggregierte Ansichten. Sie haben somit alle Analysen platzsparend und schnell erfassbar auf einen Blick dargestellt. Zusätzlich stellt der »Static Analysis Collector« ein sogenanntes *Portlet* für die Dashboard-Anzeige bereit, das tabellarisch die Analyseergebnisse für mehrere Projekte gegenüberstellt (Abb. 7–32). Beachten Sie, dass die meisten Informationen mit Links hinterlegt sind. Ein Klick auf die 63 in Spalte *Checkstyle* und Zeile *apache-commons-cli-trunk* lässt den Browser automatisch zu den 63 Checkstyle-Warnungen aus dem letzten Build dieses Projekts springen.

Plugin-Familie
»Static Analysis Utilities«

Plugin
»Static Analysis Collector«

Abb. 7–32
Portlet-Ansicht des
»Static Analysis Collector«

Metriken über alle Projekte									
Alle	Metriken	Radiator	+						
<u>Warnungen aus statischen Code-Analysen (mit Icons)</u>									
									
Job	Checkstyle	FindBugs	PMD	Offene Punkte	Compiler Warnungen	Gesamt			
apache-commons-cli-trunk	63	-	5	-	-	68			
apache-ivy-trunk	-	-	-	-	-	0			
greetr	59	3	2	-	-	64			
prognosr2	-	-	-	-	-	0			
Gesamt	122	3	7	0	0	132			
<u>Warnungen aus statischen Code-Analysen (mit Namen)</u>									
Job	Checkstyle	FindBugs	PMD	Offene Punkte	Compiler Warnungen	Gesamt			
apache-commons-cli-trunk	63	-	5	-	-	68			
apache-ivy-trunk	-	-	-	-	-	0			
greetr	59	3	2	-	-	64			
prognosr2	-	-	-	-	-	0			
Gesamt	122	3	7	0	0	132			

Ähnlich dem »Static Analysis Collector« fasst auch das Plugin »Violations« die Ergebnisse aus unterschiedlichen Analysewerkzeugen zusammen. Das Plugin ist in den Möglichkeiten der Visualisierung nicht so ausgefeilt wie die »Static Analysis Utilities«, bringt aber dafür alles in einem einzigen Plugin mit.

Plugin »Violations«

7.8.2 Warnings (Plugin)

In Builds wird typischerweise zwischen Fehlern (*errors*) und Warnungen (*warnings*) unterschieden. Fehler führen dabei zum Abbruch des Builds, etwa weil Klassen nicht kompiliert werden können oder in Zielverzeichnisse nicht geschrieben werden kann. Warnungen stellen hingegen eine »weichere« Form von Fehlern dar. Sie weisen zwar auf mögliche Probleme hin, brechen den Build aber nicht ab. Genau hier liegt das Problem: Warnungen werden dadurch oft übersehen oder deren Behebung auf »später« aufgeschoben. Um die Sichtbarkeit von Warnungen zu erhöhen, deren Auftreten im Verlauf der Builds zu protokollieren und letztendlich Maßnahmen seitens der Entwickler einzufordern, wurde das Plugin »Warnings« entwickelt. Es ist Teil der Familie der bereits beschriebenen »Static Analysis Utilities«.

Plugin »Warnings«

Das Plugin untersucht die Konsolenausgabe eines Builds sowie optional weitere Textdateien des Arbeitsbereichs nach Warnungen per Mustervergleich. Es unterstützt von Haus bereits rund 20 Compiler und Build-Werkzeuge – nicht nur aus der Java-Welt, sondern unter anderem auch AcuCobol, Ada, Buckminster, Erlang, Flex SDK, GCC, MSBuild, Oracle Invalids, PHP Laufzeitwarnungen, Sun C++ oder Texas Instruments Code Composer Studio (C/C++).

Setzen Sie in Ihrem Umfeld hauseigene Prüfwerkzeuge ein, die gezielt zu einzelnen Quelldateien Warnungen ausgeben können, dürfte »Warnings« der ideale Ausgangspunkt zur Einbindung sein: Sie müssen lediglich eine eigene Parser-Klasse schreiben, die Warnungen aus Ihrem existierenden Testberichtformat extrahieren kann und dieses dann in »Warnings« einhängen – quasi als Plugin-im-Plugin. Visualisierung, Verlaufs berechnung und alle weiteren Funktionen der »Static Analysis Utilities« bekommen Sie dann ohne weiteren Aufwand geschenkt. Hinweise für Entwickler finden sich dazu auf der Wiki-Seite des Plugins (<http://wiki.hudson-ci.org/display/HUDSON/Warnings+Plugin>).

7.8.3 Task Scanner (Plugin)

Sie kennen das sicher: In der Eile des nächsten Releases bleibt nicht immer Zeit, alle Unschönheiten im Code so zu beheben. Also schnell ein Kommentar mit den Signalwörtern TODO, FIXME oder einfach XXX eingefügt und weiter. Dadurch lassen sich diese Stellen nach dem Release wiederauffinden und in Ruhe angemessen bearbeiten. So weit die Theorie.

In der Praxis werden mit solchen Kommentaren »Hypothesen« aufgenommen, die nur selten zurückgezahlt werden. Wäre es nicht schön, wenn Hudson für Sie solche Stellen im Auge behielte?

Das Plugin »Task Scanner« durchsucht ausgewählte Dateien des Arbeitsbereiches nach Signalwörtern (z.B. FIXME, TODO oder XXX) und erstellt einen Bericht über die Fundstellen. Auch dieses Plugin gehört zur Familie der »Static Analysis Utilities« – was für Sie als Benutzer vor allem bedeutet, dass Sie von umfangreichen Visualisierungen mit einheitlicher Bedienung profitieren.

Sie richten das Plugin in der Projektkonfiguration unter *Post-Build-Aktionen → Suche im Arbeitsbereich nach offenen Punkten* ein (Abb. 7-33). Sie können dort mit regulären Ausdrücken in der Ant-FileSet-Notation angeben, welche Dateien analysiert werden sollen und welche Signalwörter gesucht werden sollen. Diese Wörter lassen sich in drei unterschiedlichen Prioritäten einteilen, was eine verfeinerte Darstellung in den Berichten zur Folge hat. Ähnlich wie bei den Plugins für statische Codeanalyse lassen sich auch hier wieder Qualitätsziele in Form von Schwellwerten für Wetterlage oder instabile Builds angeben. Nutzen Sie diese Möglichkeit unbedingt! Eine Metrik ohne Konsequenz entfaltet im Alltag nur sehr schwer Wirkung. Tipp: Beginnen Sie lieber mit einem laxen Analyseprofil, aber verwenden Sie stets einen Schwellwert von null.

Plugin »Task Scanner«

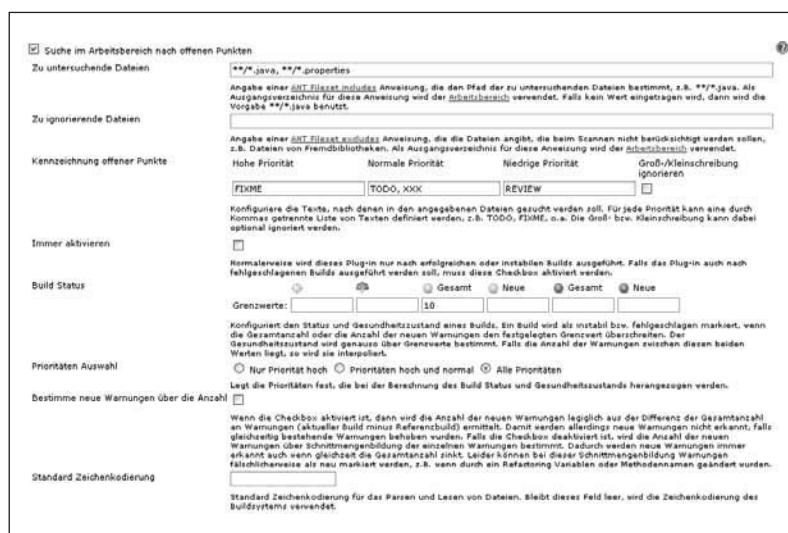


Abb. 7-33
Konfiguration des Plugins
»Task Scanner«

7.8.4 Cobertura (Plugin)

*Codeabdeckung
(code coverage)*

Haben Sie sich schon einmal gefragt, wie viel Prozent Ihres Codes während der Unit-Tests wirklich durchlaufen werden? Softwaretechniker bezeichnen diese Größe als Codeabdeckung (*code coverage*).

*Abgrenzung zur
Testabdeckung
(test coverage)*

Die Codeabdeckung wird im informellen Gespräch gerne mit der Testabdeckung (*test coverage*) verwechselt. Während erstere misst, wie viel des vorhandenen Codes getestet wird, geht es bei letzterer darum, wie viele aller möglichen Eingaben in ein System getestet werden. Beispiel: Sie haben eine einfache Taschenrechner-Anwendung entwickelt. Ihre Codeabdeckung ist 100 %, das bedeutet, alle Zeilen des Codes werden während des Testens mindestens einmal durchlaufen. Haben Sie allerdings vergessen, eine spezielle Behandlung für das Teilen durch 0 zu implementieren, und gibt es dafür keinen Test, so ist Ihre Testabdeckung unter 100 %. Da eine wirklich vollständige Testabdeckung meist nur in der Theorie möglich ist, werden in der Praxis die Testfälle so gewählt, dass sie den zu testenden Bereich möglichst gut abdecken. Eine Codeabdeckung von 100 % ist hingegen möglich – wenngleich meist mit hohem Aufwand verbunden. Viele Arbeitsgruppen teilen deshalb ihren Code in unterschiedlich wichtige Bereiche ein und fordern jeweils unterschiedliche Codeabdeckungen, z.B. mindestens 30 % für alle Bereiche, 60 % für wichtigere und sogar 90 % für kritische.

Behalten Sie bei aller Begeisterung für Metriken stets im Hinterkopf, dass eine hohe Codeabdeckung nur aussagt, dass der *vorhandene* Code umfassend getestet wurde. Es lässt sich hingegen nicht ableiten, ob die Software im Einsatz *auf allen Eingaben* fehlerfrei arbeiten wird.

Cobertura

Cobertura³ (<http://cobertura.sourceforge.net>) ist ein Werkzeug zur Berechnung der Codeabdeckung. Weitere bekannte Vertreter sind das Open-Source-Projekt EMMA (<http://emma.sourceforge.net>) und das kommerzielle Atlassian Clover (<http://www.atlassian.com>). Im Folgenden werden wir Cobertura als Repräsentant dieser Werkzeuggattung im Zusammenspiel mit Hudson betrachten.

*Instrumentierung
von Testklassen*

Falls Sie noch nie mit Codeabdeckung zu tun hatten, fragen Sie sich vielleicht, wie Cobertura technisch wissen kann, an welchen Ecken des Codes Ihre Unit-Tests vorbeikommen. Die Antwort: Ihre zu testenden Klassen werden vor dem Testen instrumentiert, also mit zusätzlichem Cobertura-Code »geimpft«. Dieser zusätzliche Code arbeitet wie ein Fahrtenschreiber und zeichnet während der Tests auf,

3. »Cobertura« bedeutet im Spanischen »Decke«, »Abdeckung«, »Zuckerguss«. Hat man sich das einmal bildlich vorgestellt, so ist der Name des Werkzeugs leicht zu merken.

welche Zeilen zur Ausführung gekommen sind. Nach Beendigung der Unit-Tests werden diese Aufzeichnungen ausgewertet und in Form eines Berichts ausgegeben. Aus diesem lässt sich für jede Zeile des Codes ersehen, ob sie während des Tests passiert wurde. Dieses zusätzliche Wissen hat allerdings auch einen Preis: Die Protokollierung benötigt zusätzliche Rechenzeit, so dass instrumentierte Test länger laufen als nicht instrumentierte.

Wie bei allen anderen Analyseverfahren ist es zunächst auch hier Aufgabe des Build-Werkzeugs, Cobertura während des Build-Prozesses auszuführen. Für gängige Build-Werkzeuge liegen dazu Anbindungen vor, also etwa Ant-Tasks bzw. Maven-Goals. Nach Abschluss des Builds sammelt Hudson dann »lediglich« die Ergebnisse ein und visualisiert und archiviert diese.

Das Visualisierungsplugin »Cobertura« fügt Hudson gleich mehrere grafische Auswertungen hinzu, vom Trendverlauf auf Projektebene, über detaillierte Berichte zu Packages und Klassen bis hin zum farblich markierten Quelltext, der ausgeführte und nicht ausgeführte Bereiche darstellt (Abb. 7–34).

Plugin »Cobertura«

Sie konfigurieren das Plugin auf Projektebene unter »Post-Build-Aktionen → Publish Cobertura Coverage Report«. Sie können hier neben Coberturas Ausgabedatei auch Schwellwerte für die Codeabdeckung setzen. Bei Unterschreiten dieser Werte verändert sich die Wetterlage Ihres Projekts bzw. werden Builds als instabil gekennzeichnet. Das Plugin verlangt eine Ausgabe von Cobertura im XML-Format. Da das cobertura-maven-plugin per Default HTML liefert, müssen Sie den Formatwunsch in Ihrer POM-Datei explizit angeben:

```
<reporting>
  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>cobertura-maven-plugin</artifactId>
      <configuration>
        <format>xml</format>
      </configuration>
    </plugin>
    [...]
  </plugins>
</reporting>
```

Listing 7–8
Konfiguration des
cobertura-maven-plugin
in der POM-Datei

Abb. 7-34
Visualisierung der Cobertura-Analysen im Verlaufe eines Projekts (oben), für ein Java-Package (Mitte) und eine Klasse (unten)

The figure illustrates the visual representation of Cobertura analysis results over time for a project, at the package level, and at the class level using the Hudson interface.

Top Panel (Project greetr):

- Left sidebar:** Shows the build history with builds #40 to #55, each with a timestamp.
- Middle section:** Displays the "Coverage Report" and "Last Änderungen".
- Right section:** Two charts: "Trend der Testergebnisse" (Test results trend) showing coverage increasing from ~10% to ~30% over 15 builds, and "Code Coverage" showing detailed metrics for Classes, Packages, Conditionsals, Files, Lines, Methods, and Packages.
- Annotation:** A callout bubble points to the "Code Coverage" chart with the text "Entwicklung der Codeabdeckung" (Development of code coverage).

Middle Panel (Code Coverage for de.simonwiest.greetr):

- Left sidebar:** Shows the build history for build #45.
- Middle section:** Displays the "Code Coverage" report for the package "de.simonwiest.greetr".
- Right section:** A chart showing the trend of code coverage for the package, starting at 0% and rising to approximately 75% over 10 builds.
- Annotation:** A callout bubble points to the chart with the text "Codeabdeckung eines Java-Packages" (Code coverage of a Java package).
- Bottom section:** Shows a "Coverage Breakdown by File" table and a "Package Coverage summary" table.

Bottom Panel (Source code of Greeter.java):

- Left sidebar:** Shows the build history for build #45.
- Middle section:** Displays the source code of the Greeter.java file.
- Right section:** A callout bubble points to the code with the text "Codeabdeckung Zeile für Zeile" (Line-by-line code coverage).

7.8.5 Sonar (Plugin)

Sonar (<http://sonar.codehaus.org>) kommt eine Sonderrolle unter den bisher vorgestellten Werkzeugen zu: Es handelt sich dabei nicht um ein weiteres Analyseverfahren, sondern um einen eigenständigen Server zur zentralen Archivierung und Visualisierung von Softwaremetriken. Ziel ist die Zusammenführung dieser Informationen an einer Stelle, ganz unabhängig davon, ob diese automatisch oder manuell erhoben wurden. Sonar betrachtet dabei sieben Dimensionen der Softwarequalität: Architektur/Design, Komplexität, Unit-Tests, Codekonventionen, mögliche Programmierfehler, Kommentierung sowie Codeduplizierung (»Copy-Paste-Doubletten«). Sonar bedient sich dabei bekannter Werkzeuge wie Checkstyle oder FindBugs, um Metriken für die obigen Dimensionen zu erheben.

Der Vorteil von Sonar gegenüber der individuellen Ausführung einzelner Werkzeuge liegt zum einen in der einfacheren Konfiguration. Maven-Benutzer müssen im Idealfall überhaupt keine zusätzlichen Angaben machen, sondern zünden mit dem schlichten Goal `sonar:sonar` ein ganzes Feuerwerk an Codeanalysen. Zum anderen speichert Sonar die Analyseergebnisse dauerhaft in einer relationalen Datenbank ab und generiert aus dieser Datenbasis Visualisierungen wie in Abbildung 7-35. Ein Highlight sind sicherlich animierte Diagramme (*motion charts*), mit denen sich die Entwicklung der Metriken im Zeitverlauf studieren lassen. Einen Demo-Server mit Analysen über viele bekannte Open-Source-Projekte aus dem Apache-Universum finden Sie unter <http://nemo.sonarsource.org>.

Neben der »passiven« Visualisierung kann Sonar auch aktiv Alarne auslösen, etwa bei Verfehlern selbst gewählter Qualitätsziele.

Sonars Leistungsbeschreibung dürfte für Sie sehr vertraut klingen. Sind das nicht auch Funktionen, die Hudson anbietet – insbesondere mit seinen mächtigen Möglichkeiten zur Visualisierung? Warum sollte man also *zusätzlich* zu Hudson noch Sonar einsetzen?

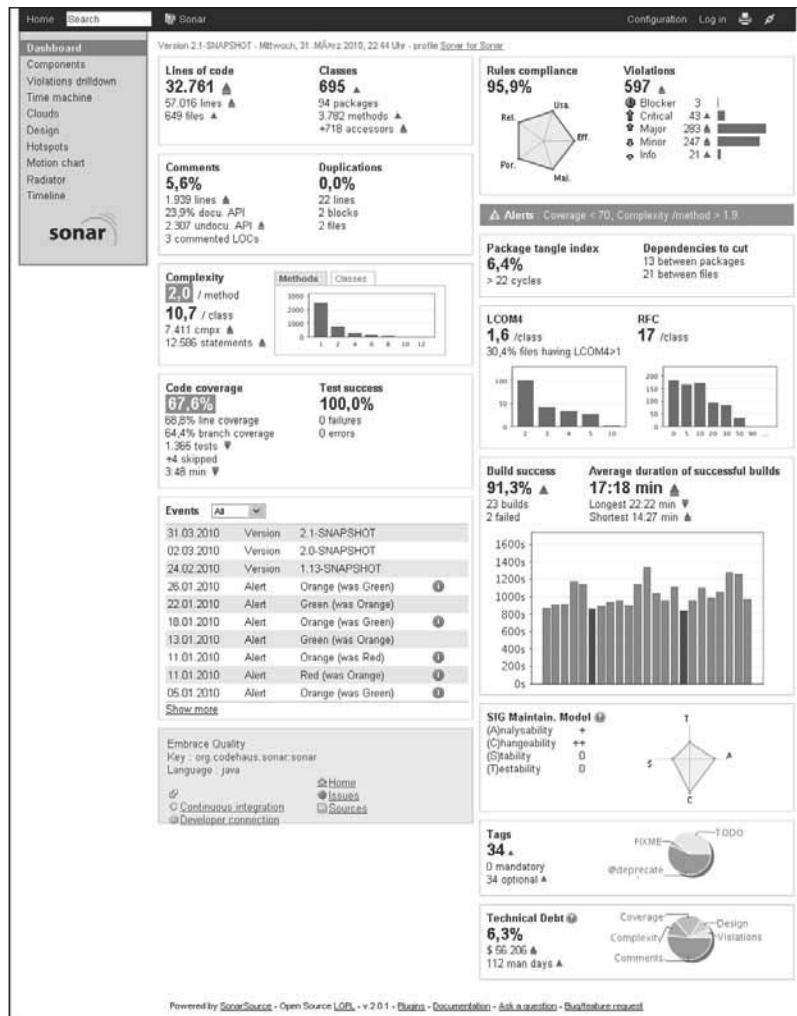
Obwohl beide Systeme teilweise sogar dieselben Metriken darstellen, liegt der Fokus bei Hudson vor allem auf der kurzfristigen Betrachtung der letzten Builds: Hier möchten Entwickler schnelle Rückmeldungen im Minutenbereich. Sonar hingegen versteht sich als Langzeitgedächtnis der Softwaremetriken. Die Sonar-Entwickler empfehlen sogar ausdrücklich, nicht jeden Build aus einem CI-System in Sonar zu erfassen, sondern höchstens einmal pro Tag einen Datensatz abzuliefern. Die meisten Auswertungen, die man später mit einem Sonar-System machen wird, verwenden in der Praxis ohnehin nur eine Genauigkeit von Wochen.

Warum Sonar?

Hudson und Sonar

Abb. 7-35

Darstellung von Analyseergebnissen in Sonar. Die meisten Zahlen und Diagramme sind mit Links zu tiefergehenden Analysen hinterlegt.



Plugin »Sonar«

Die Brücke zwischen Hudson und Sonar schlägt das Hudson-Plugin »Sonar«. Im Gegensatz zu den bisher betrachteten Plugins löst »Sonar« aktiv die Erhebung der Softwaremetriken aus und basiert nicht nur auf dem Einsammeln von Ergebnisdateien, nachdem sich der Staub eines Build-Durchlaufs gelegt hat. Liegen Metriken vor, werden sie an den Sonar-Server übermittelt und dort gespeichert. Eine Darstellung der Sonar-Ergebnisse innerhalb Hudsons findet nicht statt. Das Plugin fügt lediglich einen zusätzlichen Link zur Sonar-Weboberfläche auf der Projektseite ein. Dies verwundert aber kaum, schließlich ist die grafische Aufbereitung ja gerade die Stärke des Sonar-Systems.

Um nicht bei jedem CI-Build eine aufwendige Sonar-Analyse durchzuführen und einen neuen Datensatz zu übermitteln, richten die meisten Teams in Hudson (mindestens) zwei Projekte ein: Im ersten Projekt wird der Code gebaut und getestet, im zweiten Projekt hingegen erfolgt ein Sonar-Durchlauf, bei dem die gewünschten Metriken erhoben werden. Während das erste Projekt durch Codeänderungen ausgelöst wird, startet das zweite einmal am Tag, typischerweise in der Nacht. Die Aufspaltung in zwei Projekte klingt umständlicher, als es sich in der Praxis darstellt. Build-Abläufe werden ab einer gewissen Größe sowieso in mehrere Projekte aufgespalten (Kompilieren, Testen, Inspizieren, Dokumentieren usw.). Ein zusätzliches Projekt für den Sonar-Durchgang mit festem Zeitplan fällt dabei nur unerheblich ins Gewicht.

Konfiguriert wird das Plugin »Sonar« an zwei Stellen: Unter *Hudson verwalten* → *System konfigurieren* → *Sonar* geben Sie alle Sonar-Instanzen an, die Sie anbinden möchten. Damit das Hudson-Plugin einer Sonar-Instanz Daten übermitteln kann, müssen Sie nicht nur Adressen der Benutzeroberfläche, sondern auch der darunterliegenden Datenbank angeben. Anschließend können Sie den Projektkonfigurationen unter *Post-Build-Aktionen* → *Sonar* projektspezifische Angaben machen, etwa die zu verwendende Maven-Installation.

7.9 Issue-Tracker integrieren

Neben Versionsmanagement- und CI-System benötigt professionelle Softwareentwicklung mindestens ein weiteres essenzielles IT-System: einen Issue-Tracker (je nach Firmenkultur auch als Bug-Tracker, Ticket-System oder Fallbearbeitungssystem bekannt).

Neue Softwarefunktionen durchlaufen bis zu ihrer Fertigstellung einen Weg durch alle drei Systeme:

1. von der Änderungsanforderung im Issue-Tracker
2. über die Codeerweiterungen im Versionsmanagementsystem
3. bis hin zu den zugehörigen Builds auf dem CI-Server.

*Nachvollziehbarkeit
durch Verknüpfung von
IT-Systemen*

Daher liegt es nahe, diese Systeme auch untereinander zu vernetzen. So lassen sich schnell Fragen beantworten wie »Wurde die Umsetzung eines Tickets bereits im CI-System getestet?« (Issue-Tracker → CI-System) oder »Welches Ticket war die Ursache für diesen Build?« (CI-System → Issue-Tracker).

Die verbindende Klammer zwischen allen Systemen stellt dabei die Ticket-ID dar, die jedes Ticket eindeutig identifiziert. Typischerweise sind dies numerische Schlüssel (z.B. #1892378) oder Nummernkreise mit vorangestelltem Projektbezeichner (z.B. GREETR-925). Checkt

Ticket-ID als Klammer

ein Entwickler seine Änderungen ins Versionsmanagement ein, so baut er diese Ticket-ID in seinen Commit-Kommentar ein, z.B. *Unit-Tests für Anforderung GREETR-925 ergänzt*. Hudson erkennt beim Auschecken der Quelldateien derartige Ticket-IDs in den Informationen des Versionsmanagements und kann dadurch Ticket, Revision und Build miteinander in Verbindung bringen. Wie elegant dies in der Praxis umgesetzt ist, hängt von der jeweiligen Anbindung ab. Im Folgenden betrachten wir die Integration von Hudson mit zwei populären Issue-Trackern, JIRA und Mantis.

7.9.1 Atlassian JIRA

JIRA (<http://www.atlassian.com>) aus dem Hause Atlassian ist eines der Issue-Tracking-Systeme mit der weitesten Verbreitung. Als webbasierte Anwendung ermöglicht es nicht nur verteilten Entwicklerteams eine zentrale Ablage von Fehlerberichten und Feature-Requests (Wünschen nach zusätzlichen Funktionen). In vielen Fällen erhalten auch Kunden über das Internet direkten Zugriff auf den Issue-Tracker, etwa um neue Tickets anzulegen oder sich über den Bearbeitungsstatus bestehender Tickets zu informieren. JIRA verwendet – ähnlich wie Hudson – ein Plugin-Konzept, das eigene Erweiterungen des Funktionsumfangs erlaubt.

Zur Anbindung von Hudson und JIRA stehen gleich zwei Optionen mit unterschiedlichem Funktionsumfang zur Auswahl:

- über das Hudson-Plugin »JIRA«
- über die »JIRA-Hudson-Integration« von Marvelution

Plugin »JIRA«

Das Hudson-Plugin »JIRA« analysiert Commit-Kommentare des Versionsmanagements und stellt bei gefundenen JIRA-Ticket-Schlüsseln Verknüpfungen in beide Richtungen her (Abb. 7–36): Zum einen werden Schlüssel innerhalb der Hudson-Weboberfläche automatisch mit einem Link zum korrespondierenden JIRA-Ticket hinterlegt (Hudson → JIRA). Zum anderen fügt Hudson optional beim jeweiligen Ticket einen Kommentar mit Link zum korrespondierenden Build ein (JIRA → Hudson). Dadurch können Beobachter des Tickets beispielsweise sehr schnell herausfinden, ab welchem Build ein Problem behoben wurde.

Hudson

Build #64 (02.04.2010 15:56:24)

JIRA

Issue Details (200x West | Pastoral)

Key: GREETR_1
Type: New Feature
Status: Reopened
Priority: Major
Assignee: Simon West
Reporter: Simon West
Votes: 0
Watchers: 0

Available Workflow Actions

- Resolve Issue**
- Close Issue**
- Start Progress**
- Operations**
- Assign this issue**
- Attach file to this issue**
- Attach screenshot to this issue**
- Close this issue**
- Comment on this issue**
- Delete this issue**
- Edit this issue**
- Move this issue**
- Vote**

greeter
Description:
 Greetings are in english only at the time being. This should be extended to the following locales DE, FR, ES - controlled by an environment variable GREETER_LANGUAGE with the corresponding locale as value.

All | **Comments** | **Change History** | **Activity Stream** | **Source**

Hudson added a comment - 02/04/10 04:10 PM
 Integrated in greeter #5 ([http://subbu0010209.nethome.vch5.am](#))
 Fixed implementation for greeting in Russian (was GREETER_1)
swieli ([http://ubunu:8080/changeLog?cs=106](#))
Files:
 * /src/main/java/de/simonwest/greeter/Greet.java
 * /src/test/java/de/simonwest/greeter/GreetTest.java

Simon West added a comment - 25/04/10 03:32 PM
 Our solution seems already work with Chinese characters anyway.

Abb. 7-36
Bidirektionale Verknüpfung von Hudson und JIRA mit dem Plugin »JIRA«

Der Weg von JIRA nach Hudson erfordert drei Schritte der Vorbereitung:

1. Damit Hudson Kommentare in JIRA anlegen kann, muss JIRAs Remote-API aktiviert werden (dazu in JIRA **Administration** → **Global settings** → **General Configuration** → **Options** → **Accept remote API calls** auf ON stellen).
2. In Hudson hinterlegen Sie unter **Hudson verwalten** → **System konfigurieren** → **JIRA** die Anmelde- und Rechte eines JIRA-Benutzers, der ausreichend Rechte zur Kommentierung von Tickets besitzt (166). In der Praxis hat es sich bewährt, für diesen Zweck einen eigenen Benutzer »Hudson« in JIRA anzulegen. Dadurch ist in JIRA besser nachvollziehbar, über welchen Mechanismus die Kommentare entstanden sind. Beachten Sie, dass das Hudson-Plugin auch mehrere JIRA-Server ansprechen kann.

3. Schließlich aktivieren Sie für jedes Hudson-Projekt individuell in den Projektkonfigurationen die Option *Post-Build-Aktionen* → *JIRA-Issues aktualisieren*.



Der Reiz des Plugins »JIRA« ist die bidirektionale Verbindung von JIRA und Hudson, die keinerlei Installation auf JIRA-Seite erfordert – von der Aktivierung der Remote-Schnittstelle und ggf. dem Anlegen eines Hudson-Benutzerkontos abgesehen.

JIRA-Hudson-Integration (Marvelution)

Auch die kostenlose »JIRA-Hudson-Integration« von Marvelution (www.marvelution.com/atlassian/jira-hudson-integration) durchsucht Commit-Kommentare nach Schlüsseln von JIRA-Tickets. Werden solche Schlüssel gefunden, stellt JIRA bei den korrespondierenden Tickets unter einem eigenen Reiter *Hudson Builds* Kurzzusammenfassungen zugehöriger Builds dar (Abbildung 7-38). Diese sind mit Links zur Hudson-Instanz hinterlegt. Die Navigation ist in der aktuellen Version (3.2.0) aber nur in Richtung JIRA → Hudson möglich.

Im Gegensatz zum Hudson-Plugin »JIRA« stellt die Marvelution-Integration aber zusätzlich wichtige Hudson-Informationen als *Portlets* auf den JIRA-Übersichtsseiten (*dashboards*) dar. Portlets sind rechteckige Kacheln, aus denen JIRA-Benutzer sich ihre Übersichtsseiten zusammenstellen können (Abb. 7-39).

Technisch besteht die Marvelution-Integration aus zwei Plugins: Das *Hudson-Plugin* erlaubt es, in der Projektkonfiguration korrespondierende JIRA-Projekt-Schlüssel zuzuordnen. Des Weiteren exponiert es Daten aus dem Hudson-System über eine eigene REST-Schnittstelle. Über diese Schnittstelle lässt sich beispielsweise von außen abfragen, welche Builds für ein bestimmtes JIRA-Projekt relevant sind. Die andere Hälfte der Integration stellt das *JIRA-Plugin* dar, welches per REST diese Daten aus Hudson abruft und visualisiert.

The screenshot shows the JIRA interface for issue GREETR-1. The ticket details include: Key: GREETR-1, Type: New Feature, Status: Reopened, Priority: Major, Assignee: Simon Wiest, Reporter: Simon Wiest, Votes: 0, Watchers: 0. The ticket description is: "Implement greeting in multiple languages". Below the ticket, there is a "Buildbox Hudson" portlet displaying two Hudson builds:

- Build #30:** Started 3 days ago, Duration: 2 min 59 sec. Tests: Total: 4, Passed: 4, Failed: 0, Skipped: 0. Build artifacts: greetr-1.1-SNAPSHOT.jar, pom.xml. Build triggers: Code change build.
- Build #27:** Started 1 mo 3 days ago, Duration: 1 min 40 sec. Tests: Total: 2, Passed: 2, Failed: 0, Skipped: 0. Build artifacts: greetr-1.0.jar, pom.xml. Build triggers: Code change build.

Abb. 7-38

Anzeige eines JIRA-Tickets mit korrespondierenden Hudson-Builds der »JIRA-Hudson-Integration« (Marvelution)

Beide Plugins sind nur auf der Marvelution-Website verfügbar, also nicht im Plugin-Center in Hudson aufgeführt.

The screenshot shows the Marvelution interface with several portlets:

- Hudson Status: Buildbox Hudson:** Displays a list of recent Hudson builds:
 - greetr #83: Run: 22 min ago | Updated by anonymous | Duration: 57 sec
 - matrix #6: Run: 1 mo 3 days ago | Updated by anonymous | Duration: 11 946 sec
 - Ivyleague #10: Run: 26 days ago | Updated by anonymous | Duration: 13 sec
 - greetr_freestyle #13: Run: 5 days 20 hr ago | Dependency build greetr_50 | Duration: 20 min
 - proxmo: Run: 1 mo 3 days ago | Updated by anonymous | Duration: 11 946 sec
- Buildbox JIRA:** Shows a summary of recent JIRA activity: April, 2010. Simon Wiest reopened GREETR-1 (Implement greeting in multiple languages).
- Hudson Build Trend: greetr:** A bar chart showing build duration over time. The Y-axis is Duration (1m to 12m) and the X-axis is build number (5 to 60). The chart shows a sharp peak at build 10 followed by a general downward trend.
- Assigned to Me:** A list of issues assigned to the user:
 - GREETR-1 Implement greeting in multiple languages
 - GREETR-2 Unknown guests are greeted as "null"
- Favorite Filters:** A section stating "You have no favorite filters at the moment." with links to "Create Filter" and "Manage Filters".

Abb. 7-39

Portlets der »JIRA-Hudson-Integration« (Marvelution)

7.9.2 Mantis

Wie JIRA ist auch Mantis (<http://www.mantisbt.org>) ein webbasiertes System, das als Bug-Tracker und zur Verwaltung von Feature-Requests eingesetzt wird. Die niedrige aktuelle Versionsnummer 1.2.0 lässt vermuten, dass es sich um ein sehr junges Produkt handelt. Im Gegenteil: Mantis ist bereits seit 10 Jahren verfügbar, die Sprünge in den Versionsnummern wurden jedoch in einem Anflug von Understatement sehr klein gewählt. Mantis ist Open Source und kostenlos erhältlich. Durch seine leichtgewichtige Implementierung in PHP ist es einfach zu installieren und sehr flink in der Bedienung. Bereits in den Vorgängerversionen wurde häufig durch Patches individuell angepasst, ab Version 1.2.0 besitzt Mantis dafür ein explizites Plugin-Konzept.

Plugin »Mantis«

Die Anbindung über das Plugin »Mantis« erfolgt analog zum Plugin »JIRA«. Hudson hinterlegt den Mantis-Ticketnummern Links auf die Mantis-Instanz. Umgekehrt verweisen Kommentare in den Mantis-Tickets auf korrespondierende Builds (Abb. 7–40). Die Konfiguration ist sehr ähnlich:

1. In Hudson hinterlegen Sie unter *Hudson verwalten* → *System konfigurieren* → *Mantis* die Anmelddaten eines Mantis-Benutzers, der mindestens »Developer«-Rechte zur Kommentierung von Tickets besitzen muss. Auch hier hat es sich in der Praxis bewährt, für diesen Zweck einen eigenen Benutzer »Hudson« in Mantis anzulegen.
2. Schließlich aktivieren Sie für jedes Hudson-Projekt individuell in den Projektkonfigurationen die Option *Post-Build-Aktionen* → *Update relevant Mantis issues*. Zusätzlich geben Sie im Kopfbereich der Projektkonfiguration im Abschnitt »Mantis« an, in welcher Notation Mantis-Ticketnummern in Commit-Kommentaren auftauchen. In vielen Arbeitsgruppen ist dies ein Gatter (»#«) mit nachfolgender Mantis-Ticketnummer, z.B. »Unit-Tests für Anforderung #925 ergänzt«. Die korrekte Konfiguration im Feld *Issue id pattern* wäre dann »%ID%«. Für kompliziertere Schreibweisen können Sie alternativ in den erweiterten Einstellungen im Feld *Regexp pattern* auch ausgewachsene reguläre Ausdrücke verwenden.

Hudson

Build #72 (02.04.2010 18:30:14)

Mantis

View Advanced | Issue History | Print

ID	Category	Severity	Reproducibility	Date Submitted	Last Update
0000003	[GREETR]	feature	N/A	2010-04-02 18:25	2010-04-02 18:31

Notes

(0000002) hudson (developer) 2010-04-02 18:31

Integrated in greetr:72 See <http://ubuntu:8080/job/greetr/72/> [^].

Revision/Changset: 113 <http://ubuntu:8080/job/greetr/72/changeset/113> [^]

Author: swiest

Fixed implementation for greeting in Russian (fixes 0000003).

Changed paths:

- M ./src/main/java/de/simonwiest/greetr/Greetr.java
- M ./src/test/java/de/simonwiest/greetr/GreetrTest.java

Abb. 7-40
Bidirektionale
Verknüpfung von Hudson
und Mantis mit dem
Plugin »Mantis«

7.10 Zusammenfassung

In diesem Kapitel haben Sie die wichtigsten Werkzeuge und Handgriffe kennengelernt, die Sie als Entwickler bei Ihrer täglichen Arbeit benötigen: Sie können nun eigene Projekte in Hudson anlegen und mit umfangreichen Test-, Analyse- und Dokumentationswerkzeugen ausstatten. Durch Alarne erfahren Sie per E-Mail, Informationsradiator oder eXtreme-Feedback-Device automatisch von Problemen, die Ihre Aufmerksamkeit erfordern. Dadurch können Sie Ihre wertvolle Arbeitskraft – befreit von vielen lästigen manuellen Arbeitsschritten – für kreativere und herausfordernde Aufgaben einsetzen.

Im folgenden Kapitel wenden wir uns fortgeschrittenen Themen zu, die als Build- und Release-Manager auf Sie zukommen. Dazu gehören beispielsweise die Optimierung von Build-Zeiten, verteiltes Testen auf heterogenen Architekturen oder die Absicherung eines Hudson-Systems mit einem Rechte- und Rollenkonzept.

8 Hudson für Fortgeschrittene

Das Aufsetzen und Einrichten eines Projektes geht mit Hudson flott von der Hand und liefert prompte Erfolgsergebnisse. Gerade weil CI in Entwicklerteams in der Regel schnelle Akzeptanz findet und dann immer weitere Kreise zieht, stehen Hudson-Administratoren nach ein paar Monaten vor Fragen wie diesen:

- Wie kann ein Projekt in unterschiedlichen Parametrisierungen ausgeführt werden?
- Wie behält man bei immer mehr Projekten den Überblick?
- Wie skaliert die Gesamtleistung des CI-Systems?
- Wie kann der Zugriff auf bestimmte Benutzergruppen eingeschränkt werden?
- Welche nützlichen Plugins sollte man kennen?

In diesem Kapitel beschäftigen wir uns daher mit den fortgeschrittenen Funktionsmerkmalen Hudsons und setzen dabei voraus, dass Sie inzwischen schon ausreichend Hudson-Erfahrung sammeln konnten und in Navigation und Begriffswelt dieses CI-Servers sattelfest sind.

8.1 Parametrisierte Builds

In einigen Fällen wäre es ungemein praktisch, wenn Hudson vor der Ausführung eines Projektes ein paar Parameter in einem Formular abfragen und diese im Build-Prozess berücksichtigen würde (Abb. 8–1). Typische Parameter wären etwa der Server, auf den eine neue Software ausgebracht werden soll, ein Passwort, das während des Builds verwendet werden soll, oder die Subversion-Revision, deren Quelltexte verwendet werden sollen. Alle diese Fälle lassen sich mit *parametrisierten Builds* elegant lösen.

Abb. 8-1
Beispielabfrage beim
Start eines
parametrisierten Builds

Projekt prognosr

Dieser Build erfordert Parameter:

DEPLOYMENT_SEVER	server-production-001
PASSWORD_TOMCAT	Wählen Sie oben den Server, auf den die Buildergebnisse ausgebracht werden sollen. *****
SUBVERSION_REV	Geben Sie oben das Passwort des Kontos 'tomcat' an. @3525
Geben Sie oben die Revision im Format "@1234" an oder lassen Sie das Feld leer.	
<input type="button" value="Build"/>	

8.1.1 Definition von Parametern

Sie definieren neue Parameter in der Jobkonfiguration, indem Sie zunächst die Option *Dieser Build ist parametrisiert* aktivieren und dann die gewünschten Parameter anlegen. Parameter sind dabei keinesfalls auf Textketten beschränkt, wie Sie dies etwa von Umgebungsvariablen kennen. Stattdessen stehen mehrere Datentypen zur Verfügung. Tabelle 8-1 gibt eine Übersicht über die »eingebauten« Datentypen, Abbildung 8-2 zeigt deren Darstellung in der Benutzeroberfläche.

Tab. 8-1
Parameter-Datentypen

Datentyp	Beschreibung
Boolescher Wert	Ja-Nein-Entscheidung
Auswahl	Auswahl aus einer konstanten Liste an Optionen
Text-Parameter	Zeichenkette
Kennwort-Parameter	Zeichenkette, die nicht am Bildschirm dargestellt und verschlüsselt archiviert wird
Run-Parameter	Bestimmter Build eines festgelegten Projekts
Datei-Parameter	Hochzuladende Datei aus dem lokalen Dateisystem

Abb. 8-2
Darstellung
unterschiedlicher
Parameter-Datentypen

Projekt prognosr

Dieser Build erfordert Parameter:

INCLUDE_DEBUG_INFO	<input checked="" type="checkbox"/>
DEPLOYMENT_SEVER	server-production-001
CODENAME	FunkyChicken
PASSWORD	*****
RUN	prognosr #2
TEST_FILE	U:\autorun.inf
<input type="button" value="Durchsuchen..."/>	
<input type="button" value="Build"/>	

Parameterwerte werden nur dann per Formular abgefragt, wenn der Build interaktiv über die Benutzeroberfläche gestartet wurde. Wurde der Build hingegen automatisch ausgelöst, etwa durch ein vorgelagertes Projekt, so werden Vorgabewerte verwendet, die Sie pro Parameter individuell festlegen können.

Vorgabewerte

Über Plugins lassen sich weitere Datentypen definieren (mehr dazu in Kapitel 9, Erweiterungspunkt `ParameterDefinition`): Man könnte sich etwa einen Datentyp »Server« vorstellen, der eine Auswahl aus einer automatisch erstellten Liste momentan verfügbarer Server erfordert. Ein anderes Beispiel stellt das Plugin »Validating String Parameter« dar. Es erweitert den eingebauten Text-Parametertyp und überprüft, ob Texteingaben einen vorgegebenen Aufbau erfüllen. So lassen sich sehr einfach konsistente Schreibweisen für Datumsangaben, Versionslabels, Servernamen usw. erreichen.

Erweiterung der eingebauten Datentypen

Wie kann während eines Builds auf die »hereingereichten« Parameterwerte zugegriffen werden? Und wie kann später im Nachhinein kontrolliert werden, mit welchen Parametern ein Build gestartet wurde? Dazu stehen mehrere Wege zur Verfügung:

Wie werden die Parameter im Build verwendet?

- Parameter werden als Umgebungsvariablen sichtbar. Die Umsetzung erfolgt je nach Datentyp unterschiedlich: Textparameter werden direkt in Umgebungsvariablen gewandelt, boolesche Parameter in der Form `INCLUDE_DEBUG_INFO=false`, Run-Parameter als URL, z. B. `RUN=http://hudson.acme.de/job/foobar/2/`. Die Umkehrung, dass also alle Umgebungsvariablen automatisch wie Build-Parameter behandelt werden, gilt übrigens nicht.
- Plugins können über Hudsons Datenmodell auf die Parameter eines Builds zugreifen und diese als Eingaben verwenden.
- Parameterwerte werden mit jedem Build archiviert und sind über das Symbol *Parameter* am linken Seitenrand einer Build-Ansicht abrufbar.

8.1.2 Typische Anwendungsfälle

Die folgenden Anwendungsfälle kommen in der Praxis oft vor und können als Anregung für eigene Parametrisierungen dienen:

Ein Projekt soll mit einer ganz bestimmten Subversion-Revision gebaut werden. Dazu legen Sie einen Textparameter an und nennen diesen beispielsweise `MY_REVISION`. Im SCM-Abschnitt der Jobkonfiguration erweitern Sie die Subversion-Adresse im Feld »Repository URL« um `${MY_REVISION}`, aus `svn://localhost/foobar` wird dann also `svn://localhost/foobar${MY_REVISION}`. Ignorieren Sie die Warnmeldung hinsichtlich eines nicht existierenden Subversion-Pfades. Wenn

Auswahl einer Subversion-Revision

Sie jetzt den Build manuell starten, geben Sie die gewünschte Revision in der Form @123 in das Formularfeld `MY_REVISION` ein – oder belassen es leer, um die neueste Revision (`HEAD`) zu bauen. Dieser Kniff nützt aus, dass ein Subversion-Server ein Adressanhänsel der Form @123 als Revisionsangabe interpretiert. Es handelt sich dabei also nicht um ein Funktionsmerkmal von Hudson, sondern von Subversion.

Auswahl eines Hosts

Am Ende eines Builds soll eine frisch erstellte Software auf einem von zehn möglichen Testservern installiert werden. Dazu legen Sie einen Auswahlparameter, etwa `TEST_SERVER`, an und geben die Namen aller zehn Server im Textfeld *Auswahlmöglichkeiten* an. Wenn Sie jetzt den Build manuell starten, werden Sie zur Auswahl eines Servers aufgefordert. Ihr Build-Skript bekommt diese Auswahl als Umgebungsvariable hereingereicht und kann so die Software auf dem gewählten Server installieren.

Sicherheitsabfrage

Sie möchten die versehentliche Ausführung eines Projekts verhindern (»Hoppla, habe ich wirklich gerade das Projekt `wipe-out-all-customer-databases` gestartet?«). Dazu legen Sie einen booleschen Parameter, etwa `REALLY_BUILD_NOW`, an und fragen dessen Wert in Ihrem Build-Skript ab. In der Praxis hat sich gezeigt, dass meist bereits die Anzeige des Formulars ausreicht, um versehentliche Ausführungen zu vermeiden.

Eingeben eines Kennworts

Ihr Build-Prozess benötigt ein Kennwort, aber Sie möchten es nicht dauerhaft in der Jobkonfiguration hinterlegen. Dazu legen Sie einen Kennwortparameter, etwa `PASSWORD`, und greifen in Ihrem Build-Skript darauf zu. Achtung: Gibt Ihr Build-Prozess das Passwort während des Bauens auf die Konsole aus, wird es dauerhaft im Konsolenprotokoll gespeichert! Eine Alternative zur Übergabe sensibler Informationen ist übrigens das Plugin »Build Secret«, das Passwörter zu Beginn eines Builds aus einer Datei einlesen kann.

Übergabe einer Datei als Eingabe eines Tests

Sie stellen eine Software zur Verarbeitung von PDF-Dokumenten her. Ihre Tester möchten individuell präparierte Dateien probeweise von den neuesten Programmversionen verarbeiten lassen. Dazu legen Sie einen Dateiparameter, etwa `PDF_FILE`, an. Beim manuellen Starten des Builds laden Sie über das Browserformular die zu verarbeitende PDF-Datei auf den Hudson-Server hoch. Diese wird dort als Datei `PDF_FILE` im Arbeitsbereich Ihres Projekts abgelegt und steht somit dem Build-Prozess zur Verfügung. Somit gelangen die Daten zum Programm.

Natürlich ist auch das umgekehrte Szenario denkbar: Das Programm soll zu den Daten kommen. In diesem Fall sind Tests und Testdaten fest im Hudson-Projekt angelegt, und die *zu testende Software* wird manuell hochgeladen. Dies ist beispielsweise für Entwickler inter-

ressant, die eine individuell erstellte Version ihrer Software einer einheitlichen Qualitätsüberprüfung unterziehen möchten.

8.2 Ansichten (views) und Dashboards

Mit zunehmender Anzahl an Projekten neigt Hudsons Übersichtsseite dazu, leider alles andere als übersichtlich zu bleiben. Glücklicherweise gibt es einige Kniffe und nützliche Plugins, mit denen sich selbst mehrere Hundert Projekte sicher im Griff behalten lassen.

8.2.1 Benutzerdefinierte Listenansichten

Die tabellarische Anzeige aller Projekte auf Hudsons Übersichtsseite wird durch eine Ansicht vom Typ *Listenansicht* realisiert. Sie können sich neben dieser voreingestellten Ansicht *Alle* aber auch eigene, benutzerdefinierte Ansichten definieren. Abbildung 8–3 zeigt hierzu ein Beispiel. Benutzerdefinierte Ansichten legen Sie über das Plus-Symbol (»+«) im Karteireiter über der Projektliste an.

Meine Beispielansicht						
Diese Ansicht zeigt, wie die Spalten der Listenansicht angepasst werden können. Manche Spalten (Maven Modules, Artifakt Größe) wurden ergänzt.						
Alle	Meine Ansicht	Projekt Bar	Projekt Foo	Beschreibung bearbeiten		
Job	Letzter Erfolg	Letzter Fehlstart	Letzte Dauer	Maven Modules (Short)	Artifact Size	
apache-commons-io	8 Stunden 21 Minuten (#1)	Unbekannt	1 Minute 25 Sekunden	commons-io:2.0-SNAPSHOT	Not recorded	
commons	9 Stunden 12 Minuten (#1)	Unbekannt	1 Minute 51 Sekunden		Not recorded	
freestyle	7 Tage 22 Stunden (#10)	Unbekannt	1,7 Sekunden		7 MB	
gitter	6 Tage 22 Stunden (#11)	Unbekannt	0,51 Sekunden		2 MB	
gitter2	7 Tage 0 Stunden (#1)	Unbekannt	0,18 Sekunden		Not recorded	
greentrace	6 Tage 22 Stunden (#12)	Unbekannt	0,31 Sekunden		0 bytes	
greentrace2	6 Tage 4 Stunden (#13)	6 Tage 4 Stunden (#13)	6 Sekunden		0 bytes	
proxmod	2 Stunden 54 Minuten (#14)	Unbekannt	29 Sekunden		Not recorded	

Symbol: L

Legende: Alle Builds Nur Fehlstarts Nur jeweils letzte Builds

Abb. 8–3
Beispiel einer benutzerdefinierten Ansicht mit angepassten Spalten

Neben dem Namen der Ansicht, welcher im Karteireiter angezeigt werden soll, können Sie die anzuzeigenden Projekte festlegen. Sie können dadurch Ansichten als Filter auf die Gesamtheit alle angelegten Projekte verwenden. In der Konfigurationsseite der Ansicht (zu erreichen über *Ansicht bearbeiten*) sind dazu alle Projekte einzeln an- bzw. abwählbar.

Konfiguration einer Ansicht

Verwenden Sie ein systematisches Namensschema für Ihre Projekte (was dringend angeraten sei), so können Sie diese Auswahl auch dynamisch über einen regulären Ausdruck steuern: Ein Wert von `foo-.*`*

schließt beispielsweise die Projekte `foo-release` und `foo-test` ein, aber `bar-release` aus. Besonders praktisch dabei ist, dass ein neu angelegtes Projekt `foo-deployment` automatisch in diese Ansicht aufgenommen würde.

Darüber hinaus können Sie in der Konfigurationsseite die darzustellenden Spalten ausdünnen und in der Reihenfolge umsortieren. Über Plugins lassen sich sogar neuartige Spalten hinzufügen. In Abbildung 8–3 sehen Sie so eine neue Spalte, welche die Version des gebauten Maven-Moduls ausgibt (Plugin »Hudson Maven Info«), sowie eine weitere Spalte, welche die Gesamtgröße der erzeugten Artefakte angibt (dieses Plugin werden wir in Kapitel 9 gemeinsam entwickeln).

Tipp: Standardansicht anpassen

Immer wieder fragen Benutzer, wie man denn die Standardansicht »Alle« anpassen könne. Antwort: Man kann es nicht. Sie können sich aber problemlos zunächst eine benutzerdefinierte Ansicht nach Ihrem Geschmack einrichten, dann unter *Hudson → Hudson verwalten → System konfigurieren* im Feld »Standardansicht« diese neue Ansicht festlegen und abschließend (optional) die Ansicht »Alle« löschen. Im Endergebnis können Sie so das Ziel erreichen, Ihren Anwendern eine benutzerdefinierte Ansicht vorzustellen.

8.2.2 Projekte filtern und gruppieren

Plugin »Personal View«

Das Plugin »Personal View« erlaubt es Ihnen, benutzerspezifische Ansichten anzulegen. Das ist aber längst nicht alles: Der Name des Plugins verschweigt leider drei äußerst interessante weitere Funktionsmerkmale dieses Plugins: die schnelle Filterung von Projekten, die automatische Gruppierung bei systematischer Benennung sowie – als kleines Bonbon – die neue Spalte *Konsole* für Listenansichten.

Schnelle Filterung

Für Listenansichten können Sie reguläre Ausdrücke zur Filterung der angezeigten Projekte verwenden. Möchten Sie allerdings nur für den Moment übergangsweise die Liste der Projekte ausdünnen, ist der Weg über die Konfigurationsseite einer Ansicht zu mühsam. Mit dem Plugin »Personal View« erreichen Sie das Gleiche über ein Eingabefeld direkt über der Liste der Projekte. Sie sparen sich so also den lästigen Umweg über die Konfigurationsseite der Ansicht (Abb. 8–4, oben).

The screenshot shows a Hudson interface. At the top, there is a search bar with 'apache,*' and a 'Filter It' button. Below it is a 'Filtered ListView' header with 'Step in View'. The main area displays a table with columns: S, W, Job, Letzter Erfolg, Letzter Fehlschlag, Letzte Dauer, and Console 1. The 'Console 1' column is circled. The table contains five rows of build information for projects like apache-commons-cli, apache-commons-io, apache-commons-math, apache-wicket-stable, and apache-wicket-trunk. At the bottom, there is a legend with three icons: 'Alle Builds', 'Nur Fehlschläge', and 'Nur jeweils letzte Builds'.

Abb. 8–4
Schnelle Filterung
der Projektliste

Wenn Sie eine große Anzahl an Projekten angelegt haben, folgen diese vermutlich bereits einer gewissen Namenskonvention, etwa <KUNDE>-<PROJEKT>-<RELEASE>. Diesen Umstand macht sich das »Personal View Plugin« zunutze und gruppiert Ihre Projekte hierarchisch anhand eines einstellbaren Trennzeichens – das Ergebnis ist vergleichbar mit der Darstellung geschachtelter Unterverzeichnisse, so wie Sie es aus Dateisystemen kennen (Abb. 8–5).

Automatische
Gruppierung
(step in view)

The screenshot shows a Hudson interface with a 'tree' header. A search bar with '*' and a 'Filter It' button is present. Below it is a 'Filtered ListView' header with 'Step in View'. The main area displays a table with columns: S, W, Job, Letzter Erfolg, Letzter Fehlschlag, Letzte Dauer, and Console 1. The 'Console 1' column is circled. The table contains the same five rows of build information as in Abb. 8–4. At the bottom, there is a legend with three icons: 'Alle Builds', 'Nur Fehlschläge', and 'Nur jeweils letzte Builds'.

Abb. 8–5
Automatische
Gruppierung per
Namenskonvention

Sie können das Trennzeichen unter *Hudson* → *Hudson verwalten* → *System konfigurieren* im Feld *Filtered View* → *Character where to cut project names into parts for tree views* festlegen – alternativ kann dies auch benutzerspezifisch auf den Konfigurationsseiten des Benutzers erfolgen. Als Vorgabewert wird ein Trennstreich »-« verwendet.

Mit der neuen Spalte *Konsole* können Sie direkt aus der Listenansicht in die Konsolenausgabe eines Builds springen (in Abb. 8–4 rechts hervorgehoben). Der Mehrsprung über Listenansicht zur Build-Ansicht zur Konsolenausgabe wird damit um einen Zwischenhalt kürzt.

Spalte »Konsole«

zer. Sie sparen damit zwar nur einen Klick – doch dieser liegt in einem der meistgenutzten Navigationspfade.

8.2.3 Verschachtelte Ansichten

Plugin »Nested View«

Nimmt die Anzahl der Karteireiter in Hudsons Listenansicht überhand, können Sie auf das Plugin »Nested View« zurückgreifen: Dieses kann Ansichten verschachtelt darstellen. So können Sie auf oberster Ebene beispielsweise nur eine Handvoll von Ansichten für Ihre wichtigsten Projekte anlegen. Innerhalb dieser Ansichten auf oberster Ebene erzeugen Sie dann beliebig viele weitere Unteransichten, etwa für unterschiedliche Zweige des Projekts im Versionskontrollsystem. Ein Beispiel sehen Sie in Abbildung 8–6.

Abb. 8–6

Beispiel einer verschachtelten Ansicht
(nested view)

The screenshot shows a Hudson interface with a title bar 'Beispiel einer verschachtelten Ansicht'. Below it is a navigation bar with tabs: 'Alle', 'Projekt Bar', 'Projekt Foo', 'Projekt Zot' (which is highlighted), and '+'. To the right of the tabs is a link 'Beschreibung bearbeiten'. The main content area displays a 'View' section for 'Projekt Zot'. This section contains a list of build branches: 'Trunk', 'Release 2010-06', 'Release 2010-07', and 'Release 2010-08'. A large black oval is drawn around the 'View' section and the list of branches. A callout bubble originates from the bottom right of this oval and points to the list of branches, containing the text: 'Diese Ansichten sind nur innerhalb von "Projekt Zot" sichtbar.' At the bottom of the interface are buttons for 'Symbol: S M L', a legend with four items ('Alle Builds', 'Nur Fehlschläge', 'Nur jeweils letzte Builds'), and links for 'Beschreibung bearbeiten' and 'Beschreibung bearbeiten'.

8.2.4 Hudson als Ihr Build-Portal

Plugin »Dashboard View«

Mit dem Plugin »Dashboard View« können Sie die Projektansicht zu einem richtigen »Build-Portal« ausbauen: Aus rechteckigen »Informationskacheln« (portlets) können Sie sich Ihre Startseite nach Belieben zusammenkonfigurieren. Jedes Portlet stellt dabei einen bestimmten Aspekt Ihrer Hudson-Instanz dar: Testergebnisse mit Trends, Berichte aus Codeinspektionen, die letzten fehlgeschlagener Projekte usw. Das Besondere an der »Dashboard View« ist, dass hier Informationen aus den einzelnen Builds *projektübergreifend* verglichen werden können (Abb. 8–7). Beachten Sie außerdem, dass viele der tabellarisch angezeigten Werte mit Links hinterlegt sind, welche auf entsprechende, detailliertere Seiten führen.

Mein Dashboard

Beschreibung bearbeiten

Alle Mein Dashboard Projekt Bar Projekt Foo Projekt Zot +

S	W	Job	Letzter Erfolg	Letzter Fehlschlag	Letzte Dauer
		apache-commons-cli	12 Stunden (#1)	Unbekannt	0,75 Sekunden
		apache-wicket-trunk	Unbekannt	13 Minuten (#3)	1 Minute 58 Sekunden
		greetr-trunk	14 Minuten (#4)	Unbekannt	39 Sekunden

Symbol: S M L Legende Alle Builds Nur Fehlschläge Nur jeweils letzte Builds

Warnungen aus statischen Code-Analysen

Job				Gesamt
apache-commons-cli	-	-	-	0
apache-wicket-trunk	-	-	-	0
greetr-trunk	2	0	4	13
Gesamt	9	0	4	13

Test Statistics Chart

Test Statistics Grid

Job	Success	Failed	Skipped	Total	
apache-commons-cli	# 0	% 0%	# 0	% 0%	# 0
apache-wicket-trunk	# 0	% 0%	# 0	% 0%	# 0
greetr-trunk	# 6	% 86%	# 1	% 14%	# 7
Total	# 6	% 86%	# 1	% 14%	# 7

Instabile und fehlschlagende Projekte

-
-
- apache-wicket-trunk
- greetr-trunk

Das Plugin »Dashboard View« stellt dazu die Portalansicht mit Konfigurationsmöglichkeiten bereit und bringt gleich ein paar wichtige Portlets mit. Bereits 10 Plugins steuern zurzeit weitere bei: Die Familie der »Static Code Analysis Plugins« beispielsweise bietet Portlets an, die tabellarisch die Resultate von PMD, Checkstyle, FindBugs und anderen Werkzeugen über Projektgrenzen hinweg vergleichbar machen.

Abb. 8-7

Dashboard-Ansicht:
Hudson als »Build-Portal«

Plugins können weitere Portlets beisteuern.

8.3 Parallelisierung der Build-Aktivitäten

Builds dauern – gefühlt – eigentlich immer zu lange. Aber noch schlimmer: Sie tendieren auch dazu, im Laufe der Zeit zu wachsen! Da CI vor allem auf schneller Rückmeldung nach Codeänderungen beruht, sind Methoden zur Verkürzung der Build-Dauer von enormer Wichtigkeit. Sehr naheliegend ist dabei die Parallelisierung der Build-Aktivitäten.

Parallelisierung auf drei Ebenen

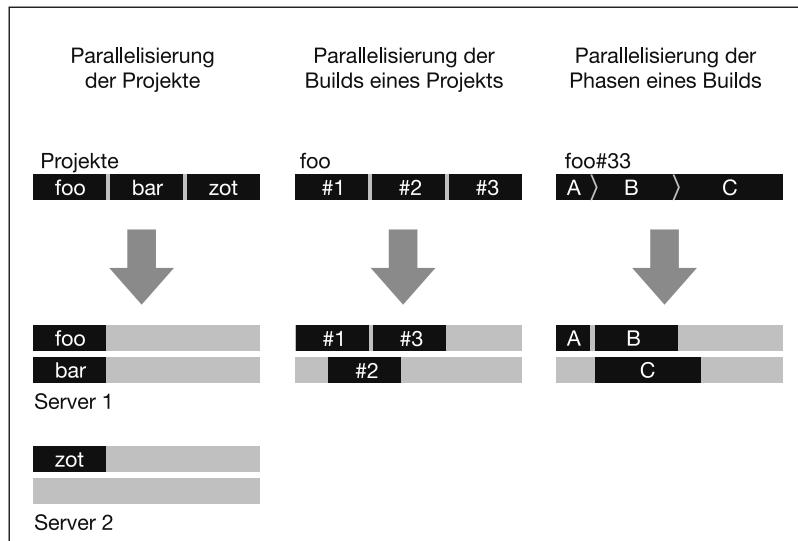
Hier sind drei Ansätze der Parallelisierung auf verschiedenen Ebenen zu differenzieren (Abb. 8–8):

- Parallelisierung der Projekte
- Parallelisierung der Builds eines Projekts
- Parallelisierung der Phasen eines Builds

Diese schließen sich nicht gegenseitig aus, sondern lassen sich sogar sehr gut miteinander kombinieren. Betrachten wir jeden der drei Ansätze kurz genauer.

Abb. 8–8

Parallelisierung auf drei unterschiedlichen Ebenen



Parallelisierung der Projekte

Großes Parallelisierungspotenzial – gerade bei einer hohen Anzahl an Projekten – bietet die Verteilung von Projekten auf mehrere Rechner (Abb. 8–8, links). Neben verkürzten Build-Zeiten bringt dieser Ansatz eine ganze Reihe weiterer Vorteile mit sich, etwa die Möglichkeit, auf unterschiedlichen Betriebssystemen zu bauen. Gleichzeitig birgt die erhöhte Komplexität aber auch Risiken. Wir werden uns mit verteilten Builds ausführlich in Abschnitt 8.5 befassen.

Parallelisierung der Builds eines Projekts

Eine weitere Parallelisierung kann zwischen den Builds eines Projektes stattfinden (Abb. 8–8, Mitte). Dadurch wird zwar nicht die Dauer eines einzelnen Builds verkürzt, trotzdem kann das CI-System in bestimmten Fällen schneller Rückmeldungen liefern, weil ein neuer Build eines Projektes gestartet werden kann, ohne das Ende eines bereits laufenden Builds dieses Projekts abwarten zu müssen. Hudson unterstützt diesen Ansatz zurzeit experimentell (in der Jobkonfiguration die Option *Parallele Builds ausführen, wenn notwendig* anwählen).

len). Bei diesem Ansatz ist zu beachten, dass parallel laufende Builds von Hudson jeweils eigenständige Arbeitsverzeichnisse zugeordnet bekommen, um Kollisionen zu vermeiden. Der benötigte Festplattenplatz kann hier unter ungünstigen Bedingungen erheblich zunehmen.

Ein Build besteht in der Regel aus unterschiedlichen Phasen wie etwa Auschecken, Kompilieren, Ausführen von Unit-Tests, Erstellen von Codeanalysen, Ausführen von Integrationstests, Erstellen von Dokumentation, Verteilen der erzeugten Artefakte usw. In vielen Fällen lassen sich einige dieser Schritte parallelisieren: So könnte etwa die Erstellung der API-Dokumentation parallel zu den Integrationstests laufen. Sind zusätzliche Ressourcen vorhanden, so kann in diesem Fall durch die parallele Ausführung der Build-Phasen die Build-Dauer verkürzt werden (Abb. 8–8, rechts). In Hudson wird diese Idee durch Zerteilen eines großen Projektes in mehrere kleinere Projekte realisiert, die untereinander verkettet sind. Der verbleibende Teil dieses Abschnitts 8.3 widmet sich diesem Ansatz.

Parallelisierung der Phasen eines Builds

8.3.1 Abhängigkeiten zwischen Projekten

Angenommen, Ihr Build-Prozess besteht aus drei Phasen: Kompilierung, Erstellung der API-Dokumentation (z.B. mit Javadocs) sowie einer statischen Codeanalyse (z.B. mit FindBugs). Da die letzten beiden Phasen parallel laufen können, wäre eine Aufteilung denkbar wie in Abbildung 8–9 dargestellt. Um dies mit Hudson umzusetzen, müssen Sie zunächst zwei Probleme lösen:

- Wie können Abhängigkeiten zwischen Projekten definiert werden, so dass nach Abschluss eines Builds ganz automatisch nachgelagerte Projekte gestartet werden?
- Wie können Dateien, die während eines Builds erzeugt wurden, an nachgelagerte Builds übergeben werden, etwa kompilierte Java-Klassen an eine FindBugs-Analyse?

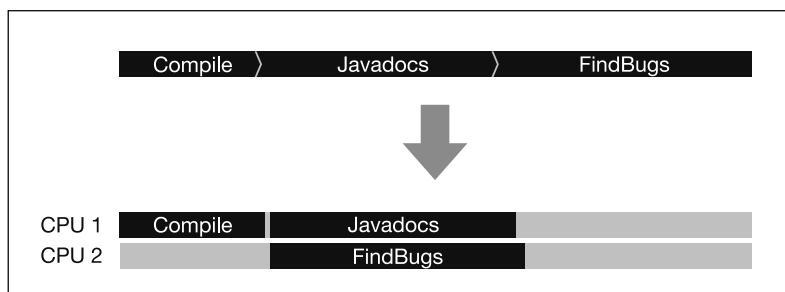


Abb. 8–9
Zerteilen eines Builds

Abhängigkeiten zwischen den Projekten

Abhängigkeiten zwischen den Projekten definieren Sie in der Jobkonfiguration im Abschnitt *Build Auslöser* → *Starte Builds, nachdem andere Projekte gebaut wurden* bzw. unter *Post-Build-Aktionen* → *Weitere Projekte bauen*. Diese Beziehungen sind symmetrisch. Technisch betrachtet ist es also egal, ob Sie die Beziehung vom vorgelagerten zum nachgelagerten Projekt aufbauen oder umgekehrt – Hudson aktualisiert jeweils die andere Seite automatisch.

Tipp: Abhängigkeiten visualisieren

Intern verwaltet Hudson einen Baum mit allen Projektabhängigkeiten, den sogenannten *dependency graph*. Anhand dieser Datenstruktur stößt Hudson nach einem Build die richtigen nachgelagerten Projekte an. Der *dependency graph* wird momentan (V1.360) nicht offiziell in der Benutzeroberfläche visualisiert. Über ein undokumentiertes und experimentelles Funktionsmerkmal können Sie jedoch unter <http://<ihr.hudson.server>/dependencyGraph/graph> eine kleine Grafik abrufen – aber erwarten Sie nicht zu viel.

Plugin

»Downstream View«

Das Plugin »Downstream View« zeigt Ihnen für einen konkreten Build eines Projektes alle dadurch ausgelösten nachgelagerten Builds (Abb. 8–10). Möglich ist dies, weil Hudson zu jedem nachgelagerten Build dessen Ursache aufzeichnet. Beachten Sie, dass Sie in der Visualisierung des Plugins »Downstream View« einen eingefrorenen Schnappschuss der Projektbeziehungen zum Build-Zeitpunkt sehen. Diese Zusammenhänge können sich inzwischen längst verändert haben.

Abb. 8–10

Beziehungen zwischen

Builds mit dem

»Downstream View

Plugin«

Downstream build view

Downstream build view status

- project-b build number 4 (Sat May 29 21:25:38 CEST 2010 - SUCCESS)
- project-c build number 4 (Sat May 29 21:25:38 CEST 2010 - SUCCESS)
 - project-c_1 build number 4 (Sat May 29 21:25:53 CEST 2010 - SUCCESS)
 - project-c_2 build number 1 (Sat May 29 21:25:53 CEST 2010 - SUCCESS)
- project-d build number 1 (Sat May 29 21:25:38 CEST 2010 - SUCCESS)

Übergeben von Dateien an nachgelagerte Builds

Nachgelagerte Builds benötigen meistens Dateien, die während vorgelagerter Builds erzeugt wurden. Im folgenden Beispiel soll der Build #2 des Projektes B (kurz: B#2) Dateien des vorgelagerten Builds #7 des Projektes A (kurz: A#7) verwenden. Wie können also diese Dateien zwischen Builds übergeben werden?

Problematisch: Zugriff auf den Arbeitsbereich

Ein häufig anzutreffender Ansatz greift vom nachgelagerten Build B#12 direkt im Dateisystem auf den Arbeitsbereich des vorgelagerten Projektes A zu. Dabei werden anhand von »Insiderwissen« Annahmen

über den internen Aufbau von Hudsons lokalem Datenverzeichnis getroffen. Von diesem Vorgehen ist abzuraten, da zum einen diese Annahmen bei verteilten Builds nicht immer zutreffen und zum anderen während der Ausführung von B#2 bereits parallel im Arbeitsbereich von A der nächste Build A#8 gebaut werden kann und es somit zu inkonsistenten Lesezugriffen auf den Arbeitsbereich von A kommen kann. Verwenden Sie daher diesen Ansatz nur, wenn Sie die technischen Konsequenzen komplett überschauen.

Alle Probleme des vorangegangenen Ansatzes vermeiden Sie durch Nutzung des Artefakt-Konzeptes, das Hudson bereits von Haus aus mitbringt. Artefakte sind ausgewählte Dateien, die am Ende eines Builds aus dem Arbeitsbereich in einen gesonderten Archivbereich kopiert werden, der *pro Build* angelegt wird. Selbst wenn der Arbeitsbereich zu Beginn eines neuen Builds komplett gelöscht wird, bleiben archivierte Artefakte auch langfristig erhalten, da Projekte zwar nur einen Arbeitsbereich, aber beliebig viele Builds (mit zugehörigen Archiven) besitzen können.

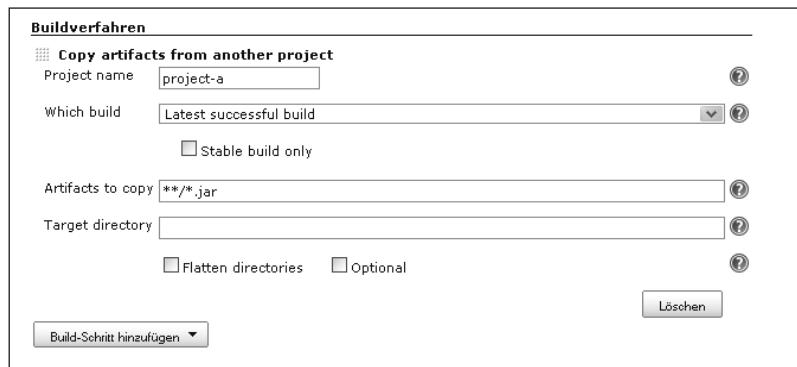
Im ersten Schritt archivieren Sie also alle Dateien, die Sie in nachgelagerten Projekten benötigen, als Artefakte. Dies spezifizieren Sie in der Jobkonfiguration im Abschnitt *Post-Build-Aktionen → Artefakte archivieren*. Geben Sie hier einen oder mehrere Dateinamen an – diese werden immer ausgehend vom Arbeitsverzeichnis verstanden. Sie könnten auch reguläre Ausdrücke verwenden: `readme.txt,license/*,lib/**/*.jar` würde beispielsweise die Datei `readme.txt`, alle Dateien im Verzeichnis `license` sowie alle JAR-Dateien unterhalb von `lib` (egal in welcher Verzeichnistiefe) archivieren.

Im zweiten Schritt müssen Sie auf die archivierten Artefakte zugreifen können. Da alle Artefakte per HTTP abrufbar sind, könnten Sie dies beispielsweise durch das Kommando `wget http://hudson.acme.de/job/a/lastCompletedBuild/artifacts/readme.txt` innerhalb eines Build-Schritts in Projekt B erreichen. Wesentlich eleganter bewerkstelligen Sie dies jedoch mit dem Plugin »Copy Artifact«, mit dem Sie im Rahmen eines Build-Schrittes Artefakte aus vorgelagerten Projekten in den Arbeitsbereich des momentanen Builds kopieren können (Abb. 8–11). Benötigen Sie nur ausgewählte Dateien aus den Artefakten der vorgelagerten Builds, können Sie die zu kopierenden Dateien sogar eingrenzen und so Zeit beim Kopieren einsparen. Da das Plugin »Copy Artifact« Artefakte aus dem Archiv eines Builds kopiert und nicht aus dem Arbeitsbereich eines lokalen Verzeichnisses, funktioniert es problemlos auch im Zusammenspiel mit parallelen und verteilten Builds.

Übergabe als Artefakte

Plugin »Copy Artifact«

Abb. 8-11
Plugin »Copy Artifact«

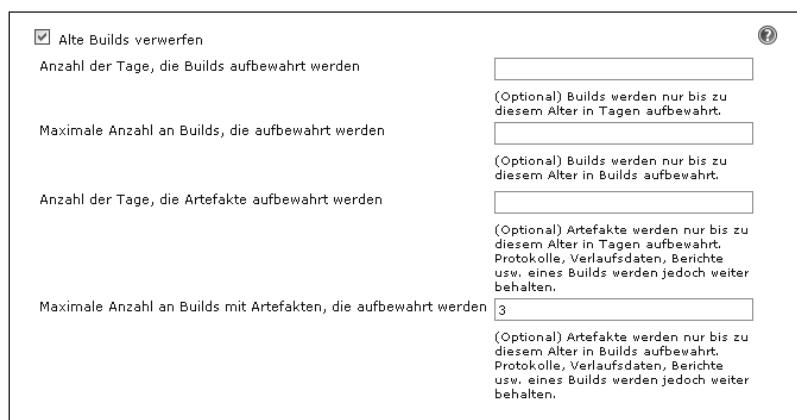


Nachteile des Artefakt-Konzepts

Zwei Nachteile des Artefakt-Konzepts sollen nicht verschwiegen werden:

- Zum einen benötigt das Archivieren der Artefakte sowie das Kopieren zum nachgelagerten Build Zeit und ggf. auch Netzbandbreite. Wirklich praktikabel ist das Verfahren daher nur, wenn der Gesamtumfang der Artefakte den zweistelligen Megabyte-Bereich nicht überschreitet.
- Zum anderen werden die Artefakte dauerhaft mit den jeweiligen Builds archiviert. Im Laufe der Zeit können hier enorme Datenmengen anfallen – wenn Sie hierfür keine Vorehrungen getroffen haben. Glücklicherweise gibt es in der Jobkonfiguration die Option *Alte Builds verwerfen*, in deren erweiterten Einstellungen Sie angeben können, dass Sie zwar alle Builds langfristig aufbewahren wollen (also etwa die Konsolenausgaben oder die Testberichte), die zugehörigen Artefakte hingegen früher entsorgen lassen möchten. Hierzu können Sie ein Verfallsdatum angeben in Form eines maximalen Alters in Tagen oder der maximalen Anzahl an Builds mit aufzubewahrenden Artefakten.

Abb. 8-12
Alte Artefakte verwerfen,
Builds aber behalten



8.3.2 Nachgelagerte Builds zusammenführen

Bisher haben wir nur Build-Prozesse betrachtet, bei denen ein Build einen oder mehrere *nachgelagerte* Builds auslösen sollte. In einigen Fällen möchte man jedoch *vorgelagerte*, parallel ausgeführte Builds wieder zusammenführen, etwa in einem abschließenden Build, der die Ergebnisse mehrerer vorgelagerter Builds zusammenfasst (Abb. 8–13). Wegen der visuellen Ähnlichkeit des Graphen mit dem Karo-Symbol auf Spielkarten (engl. *diamond*) spricht man hier auch von *diamond builds*.

Diamond Builds

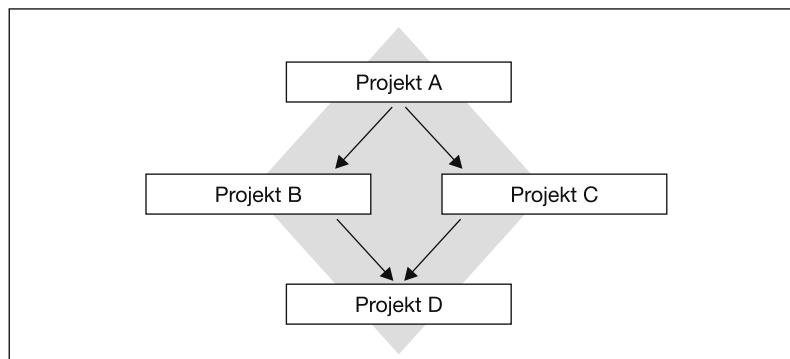


Abb. 8–13

Projektbeziehungen in Form eines diamond builds

Sie können Diamond-Builds mithilfe des Plugins »Join« realisieren: Ein Projekt A (die obere Spitze des Karos) wartet in diesem Fall, bis alle seine direkt nachgelagerten Projekte B und C beendet sind und löst dann erst Projekt D (die untere Spitze des Karos) aus. Beachten Sie, dass das vollständige Wissen um den »Diamantaufbau« in Projekt A steckt. Projekt D kennt seine Beziehung zu Projekt A nicht. Die Builds von Projekt D hingegen enthalten aber den Hinweis *Gestartet durch vorgelagertes Projekt A, Build #X*, so dass diese Beziehung nicht gänzlich unsichtbar bleibt.

Plugin »Join«

Post-Build-Aktionen	
<input checked="" type="checkbox"/> Weitere Projekte bauen	(?)
Zu bauende Projekte	<input type="text" value="project-b,project-c"/>
<input checked="" type="checkbox"/> Join Trigger	(?)
<input type="checkbox"/> Auslösen, selbst wenn der Build instabil ist.	
<input type="checkbox"/> Trigger even if some downstream projects are unstable	
Projects to build once, after all downstream projects have finished	<input type="text" value="project-d"/>
<input type="checkbox"/> Run post-build actions at join	

Abb. 8–14

Diamond-Builds mit dem Plugin »Join«

Option »Build blockieren, solange vorgelagertes Projekt gebaut wird«

Mit dem Plugin »Join« auf den ersten Blick verwandt, aber sehr unterschiedlich in der Wirkung ist die Option *Build blockieren, solange vorgelagertes Projekt gebaut wird*, die Sie in der Jobkonfiguration finden. Hierbei wird ein neuer Build eines Projekts blockiert, solange vorgelagerte Projekte gebaut werden oder in der Build-Warteschlange enthalten sind. Dabei werden auch *transitive* Abhängigkeiten berücksichtigt: Löst Projekt A Projekt B aus und Projekt B wiederum Projekt C, so würde mit dieser Option Projekt C nur gebaut, wenn weder B (C direkt vorgelagert) noch A (C transitiv über B vorgelagert) gerade gebaut oder in der Warteschlange vorgemerkt wären. Die Idee dahinter ist, dass etwa ein aufwendiges Testverfahren in Projekt C nicht durchgeführt werden soll, wenn bereits klar ist, dass durch einen neuen Build von A oder B in der nächsten Zeit Projekt C erneut ausgelöst werden wird. Die Kehrseite ist allerdings, dass es bei hoher Aktivität der Projekte A und B dazu kommen kann, dass das Projekt C »verhungert«, also nie zur Ausführung kommt. Je nach konkretem Anwendungsfall kann diese Wirkung erwünscht oder problematisch sein.

8.3.3 Exklusiver Zugriff auf Ressourcen

In der Praxis kommt immer wieder die Situation vor, dass Builds unterschiedlicher Projekte eine gemeinsame Ressource für die Dauer des Builds exklusiv benötigen, beispielsweise den Zugriff auf die Bildschirmausgabe zum Testen von Desktop-Anwendungen.

Plugin »Locks and Latches«

Mithilfe des Plugins »Locks and Latches« können Sie vermeiden, dass kollidierende Projekte gleichzeitig gebaut werden. Gemeinsam verwendete Ressourcen werden dabei durch ein Schloss (*lock*) repräsentiert, das zu einem Zeitpunkt nur maximal einem Build zugesprochen werden kann. Sie definieren eigene Locks unter *Hudson → Hudson verwalten → System konfigurieren* im Abschnitt »Locks« (Abb. 8–15).

Abb. 8–15

Definition von Locks in der Systemkonfiguration



Benötigte Ressourcen legen Sie anschließend pro Projekt in der Jobkonfiguration im Abschnitt *Buildumgebung → Locks* fest (Abb. 8–16).

Während des Bauens bekommt der erste anfragende Build A#1 die gewünschten Ressourcen zugeteilt und blockiert diese für die Dauer seiner Ausführung. Fordert während dieser Zeit ein anderer Build B#1

**Abb. 8-16**

Nutzung von Locks in der Jobkonfiguration

dieselben Ressourcen an, wird er in eine 60-Sekunden-Pause geschickt, um danach erneut die Verfügbarkeit der Ressourcen zu überprüfen. Wurde A#1 inzwischen beendet und sind die Ressourcen somit frei geworden, kann B#1 fortfahren – andernfalls wird er erneut in eine Zwangspause geschickt.

8.3.4 Parameterübergabe an nachgelagerte Builds

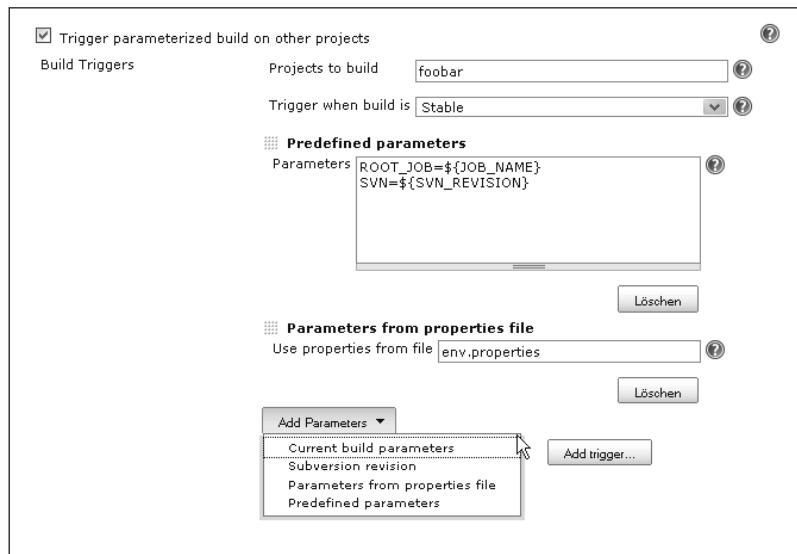
In Abschnitt 8.3.1 haben Sie bereits Anregungen erhalten, wie Sie Dateien aus einem vorgelagerten Build in einen nachgelagerten übergeben können. Oftmals möchte man aber nicht (nur) Dateien, sondern auch Parameter an nachgelagerte Builds übergeben, etwa die Subversion-Revision, die konsistent in allen Builds verwendet werden soll.

Mit Hudsons Bordmitteln ist dies momentan nicht möglich. Die Lücke wird jedoch vom Plugin »Parameterized Trigger« geschlossen (Abb. 8-17). Das Plugin erlaubt auf vier Wegen Parameter zu definieren, die an nachfolgende Builds weitergegeben werden:

Plugin »Parameterized Trigger«

- *Current Build Parameters:*
Alle Parameter, die für den vorgelagerten Build angelegt wurden, werden weitergereicht.
- *Subversion Revision:*
Die verwendete Subversion Revision wird auch beim nachgelagerten Build eingesetzt.
- *Parameters from properties file:*
Eine Textdatei wird eingelesen und deren Inhalt in Form von Parametern weitergegeben. Die Datei kann mehrere Parameter mit dem Aufbau name=wert in jeweils einer eigenen Zeile enthalten. Umgebungsvariablen können verwendet werden (z.B. ROOT_JOB=\${JOB_NAME}). Da diese Textdatei vom Build selbst erzeugt werden kann, stellt dies einen sehr universellen Weg dar, um dynamische Informationen an nachgelagerte Builds weiterzurichten.
- *Predefined Parameters:*
Hier können Sie Parameter und ihre Werte direkt vorgeben. Da Sie hier auch Umgebungsvariablen verwenden können, ist dieser Ansatz wesentlich dynamischer, als es seine Bezeichnung »predefined« zunächst vermuten lässt.

Abb. 8-17
*Weitergabe von
 Parametern an
 nachgelagerte Builds
 mit dem Plugin
 »Parameterized Trigger«*



*Flexiblere Auslösung
 von Builds*

Darüber hinaus erlaubt das Plugin »Parameterized Trigger« – im Gegensatz zu Hudsons eingebauten Möglichkeiten zur Build-Auslösung – auch das Starten von Builds als Reaktion auf *fehlgeschlagene* vorgelagerte Builds. So lassen sich beispielsweise gezielt »Aufräum-jobs« auslösen, um störende Überreste eines Fehlschlags zu entsorgen.

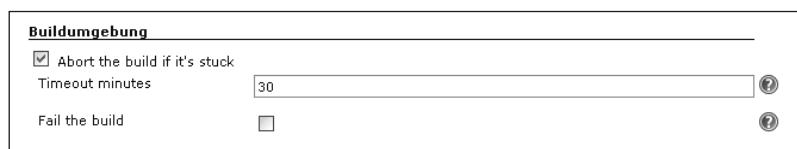
8.3.5 Abbruch hängender Builds

Plugin »Build Timeout«

Eine unspektakuläre, aber sehr nützliche Abrundung des Themas »Parallelisierung von Builds« bietet das Plugin »Build Timeout«, das Builds nach einer maximalen Laufzeit automatisch beendet. Dadurch vermeiden Sie, dass »hängengebliebene« Jobs Ihr Hudson-System unbegrenzt blockieren.

Die maximal erlaubte Laufzeit können Sie pro Projekt in der Jobkonfiguration im Abschnitt *Buildumgebung* → *Abort the build if it's stuck* festlegen (Abb. 8-18).

Abb. 8-18
Plugin »Build Timeout«



8.4 Multikonfigurationsprojekte

Als Build-Manager steht man immer wieder vor der Aufgabe, denselben Build-Prozess in mehreren Konfigurationen durchzuführen.

Beispiel: Sie möchten eine frisch erstellte Software in jeweils drei unterschiedlichen Kundenanpassungen (Alpha, Beta, Gamma) in jeweils fünf Sprachen (en, de, es, it, fr) testen. Kunde Gamma benötigt allerdings keine französische Anpassung. Insgesamt müssten also bereits $3 \times 5 - 1 = 14$ Konfigurationen getestet werden (Abb. 8–19).

		Kunden		
		Alpha	Beta	Gamma
		en	●	●
		de	●	●
		es	●	●
		it	●	●
		fr	●	●

Diese Konfiguration wird nicht gebaut (grauer Ball).

Abb. 8–19
Beispiel einer
Konfigurationsmatrix

Nun hat Ihr Produktmanager bereits weitere Kunden und weitere Sprachanpassungen avisiert, was die Anzahl der zu testenden Konfigurationen kombinatorisch explodieren lässt! Sie benötigen daher dringend Antworten auf die folgenden Fragen:

- Wie kann man in Hudson ähnliche Konfigurationen eines Projekts effizient definieren und verwalten? Anders ausgedrückt: Wie vermeiden Sie es, 14 Projekte für jeweils eine Konfiguration anlegen zu müssen?
- Wie können die Builds der (typischerweise vielen) Konfigurationen parallelisiert ausgeführt werden?
- Wie lassen sich abschließend die Ergebnisse der Builds aller Konfigurationen zu einem Bericht zusammenfassen?

Genau diese Punkte adressiert Hudson mit dem Projekttyp »Multikonfigurationsprojekt«, auf den wir im Folgenden eingehen. Vielleicht ist Ihnen bereits beim Anlegen neuer Projekte neben den bekannten Projekttypen *Free-Style-Projekt* oder *Maven-2-Projekt* die Option *Multikonfigurationsprojekt* aufgefallen.

Projekttyp »Multikonfigurationsprojekt«

Da der Projekttyp die einzige Eigenschaft eines Projektes darstellt, die sich im Nachhinein nicht mehr verändern lässt, bedeutet dies, dass Sie Multikonfigurationsprojekte immer bereits als solche anlegen müssen und bereits bestehende Projekte *nicht* nachträglich umwandeln können.

Konzeptionell können Sie sich ein Multikonfigurationsprojekt als ein übergeordnetes Free-Style-Projekt vorstellen, das dynamisch Unterprojekte erzeugen und ausführen kann. Diese untergeordneten Projekte werden »Matrix-Projekte« genannt, ihre ausgeführten Builds »Matrix-Builds«. Ein Multikonfigurationsprojekt führt also während eines Builds mehrere untergeordnete Matrix-Builds aus. Die Konfigurationen der Matrix-Projekte (SCM, Build-Schritte, Post-Build-Aktionen usw.) werden vor der Ausführung vom übergeordneten Projekt kopiert. Zusätzlich werden jedoch von Matrix-Build zu Matrix-Build unterschiedliche Parameterwerte hineingereicht, auf die ein Matrix-Build dann reagieren kann.

8.4.1 Anlegen eines Multikonfigurationsprojekts

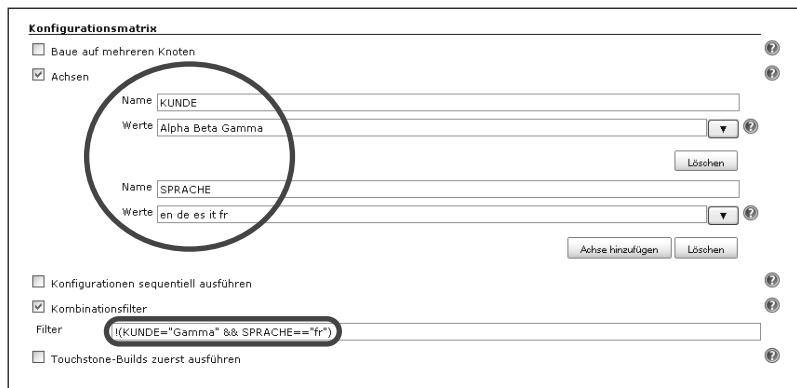
Definition der Achsen

Der Projekttyp »Multikonfigurationsprojekt« arbeitet mit sogenannten Achsen, die den Konfigurationsraum Ihres Projektes aufspannen. Im eingangs erwähnten Beispiel wären das also die Achsen KUNDE und SPRACHE. Die Namen der Achsen können frei gewählt werden und sind später als Umgebungsvariablen während des Build-Ablaufs sichtbar. Daher wird empfohlen, den typischen Konventionen für die Benennung von Umgebungsvariablen zu folgen (keine Leerzeichen und Umlaute verwenden, Namen in Großbuchstaben schreiben, auf Kollisionen mit bestehenden Umgebungsvariablen achten).

Für jede Achse wird eine Liste möglicher Ausprägungen hinterlegt (Abb. 8–20):

Abb. 8–20

*Definition der Achsen
eines Multikonfigurations-
projektes*



Sequenzielle Ausführung

Im Normalfall werden Konfigurationen parallel gebaut. Das liegt nahe, denn in den meisten Fällen stellt eine hohe Anzahl an Konfigurationen und die damit verbundene Notwendigkeit zur Parallelisierung ja gerade die Motivation für ein Multikonfigurationsprojekt dar. Es

gibt aber auch Fälle, in denen Matrix-Builds den exklusiven Zugriff auf gemeinsame Ressourcen benötigen und daher sequenziell gebaut werden müssen. In diesem Fall kommt die Option *Konfigurationen sequenziell* ausführen zur Hilfe. Gleichzeitig wird damit vermieden, dass eine große Zahl an Matrix-Builds andere Builds in der Build-Warteschlange verdrängt.

Hudson baut standardmäßig alle möglichen Kombinationen aller Achsenwerte. Sollen bestimmte Konfigurationen nicht gebaut werden – etwa weil sie aus fachlicher Sicht nicht sinnvoll, technisch unmöglich oder schlichtweg zu zahlreich sind – können einzelne Konfigurationen gezielt während des Builds ausgelassen werden. Dazu geben Sie einen Kombinationsfilter in Form eines booleschen Ausdrucks in der Programmiersprache Groovy an, der von Hudson zur Build-Zeit für jede Konfiguration ausgewertet wird. Je nach Resultat wird die betreffende Konfiguration gebaut (`true`) oder übersprungen (`false`). Die Achsenwerte einer Konfiguration stehen in den Filterausdrücken als gleichnamige Variablen zur Verfügung. Auf diese Weise wird in Abb. 8–20 unten die Konfiguration »Gamma/französisch« ausgeschlossen. Zum Glück müssen Sie kein Groovy-Experte sein, um solche Filterausdrücke zu erstellen. Mit den Beispielen in Abbildung 8–2 aus Abbildung 8–2 kommen Sie bereits sehr weit.

*Kombinationsfilter:
Auswahl von
Konfigurationen*

Ausdruck	Auswirkung
<code>KUNDE=="Alpha" SPRACHE=="en"</code>	Es werden nur Konfigurationen gebaut, die für den Kunden Alpha angepasst oder auf Englisch sind (oder beides).
<code>KUNDE=="Beta" && SPRACHE!="en"</code>	Es werden nur Konfigurationen gebaut, die für den Kunden Beta angepasst und nicht auf Englisch sind.
<code>! (KUNDE=="Gamma" && SPRACHE=="en")</code>	Es werden alle Konfigurationen gebaut, außer die englische Version für den Kunden Gamma.
<code>index%3==0</code>	Es wird nur jede 3. Konfiguration gebaut (meist nur sinnvoll, wenn Sie aus sehr zahlreichen Konfigurationen nur Stichproben bauen können). Die von Hudson vorbelegte Variable <code>index</code> nummeriert jede Konfiguration fortlaufend durch.

Tab. 8–2
*Beispiele für
Kombinationsfilter*

Sie können auch die Reihenfolge beeinflussen, in welcher Hudson die einzelnen Konfigurationen ausführt. Oft lassen sich besonders schnelle oder repräsentative Konfigurationen finden, die zuerst gebaut werden sollten, um möglichst frühzeitig grundsätzliche Probleme zu erkennen (»Kompileiere zuerst auf unserer schnellen Linux-Maschine. Wenn's da

Touchstone Builds

schon nicht läuft, musst du es auf dem alten Mac gar nicht erst versuchen.«). Die Builds dieser speziellen Konfigurationen werden als *Touchstone-Builds*¹ bezeichnet.

Sie können die Ausführung aller verbleibenden Konfigurationen vom erfolgreichen Ausgang der Touchstone-Builds abhängig machen und so gegebenenfalls einer Verschwendung von Ressourcen vorbeugen. Welche Konfigurationen als Touchstone-Builds gebaut werden sollen, definieren Sie ebenfalls mit einem Groovy-Filterausdruck analog zum bereits beschriebenen Kombinationsfilter.

8.4.2 Verteilte Matrix-Builds

Matrix-Builds lassen sich nicht nur parallel bauen. Hudson kann zusätzlich auch noch die Verteilung auf mehrere Knoten für Sie übernehmen.

Option »Baue auf mehreren Knoten«

Im Beispiel in der Abbildung 8–21 ist die Option *Baue auf mehreren Knoten* angewählt, mit der Matrix-Builds verteilt auf mehreren Rechnern ausführt werden. Diese Option wird Ihnen übrigens nur angezeigt, wenn Sie mindestens einen Slave-Knoten angelegt haben.

Im dargestellten Baum aus Knoten und Labels (Gruppen von Knoten) können Sie auswählen, wo Ihre Matrix-Builds gebaut werden sollen. Dabei sind drei Fälle zu unterscheiden:

- Sie haben *weder Rechner noch Labels* angewählt: In diesem Fall überlassen Sie Hudson die Entscheidung, auf welchem Slave-Knoten ein Matrix-Build zur Ausführung kommt.
- Sie haben *genau einen Rechner bzw. ein Label* angewählt. Alle Matrix-Builds werden dann auf diesem Rechner bzw. auf mit diesem Label versehenen Rechnern ausgeführt. Das kann sinnvoll sein, wenn der Build spezielle Systemvoraussetzungen benötigt, etwa ein bestimmtes Betriebssystem.
- Sie haben *mehrere Rechner bzw. Labels* angewählt. In diesem Fall erweitert Hudson die Konfigurationsmatrix um eine Achse *label* und fügt die angewählten Rechner bzw. Labels als Achsenwerte hinzu. Die Konfigurationen werden also nicht auf die ausgewählten Rechner bzw. Labels aufgeteilt, sondern vielmehr auf *jedem* Rechner bzw. Label ausgeführt. Die Anzahl der zu bauenden Konfigurationen erhöht sich dabei also!

1. *Touchstone* bedeutet übersetzt »Prüfstein« und bezieht sich auf eine Methode der Bestimmung des Reinheitsgrades von Edelmetallen durch Reiben auf einem speziellen Prüfstein (<http://de.wikipedia.org/wiki/Prüfstein>).

In unserem Beispiel möchten wir alle Kundenanpassungen in allen Sprachvarianten auf vier Betriebssystemen (Windows, Mac OS X, Linux und Solaris) testen. Passenderweise haben wir bereits Slave-Knoten mit Labels angelegt und können diese nun im Feld »Knoten« anwählen.

Die Anpassung für den Kunden Alpha soll in unserem Beispiel nicht unter Solaris getestet werden, weil der Kunde dieses Betriebssystem nicht einsetzt. Wir erweitern dazu den Kombinationsfilter um eine Bedingung, die den Wert der `label` verwendet. Abschnitt 8–21 zeigt die Umsetzung.

Die Achse »label«

Abb. 8–21

Konfiguration eines
verteilten Matrix-Builds

8.4.3 Zusammenfassung der Build-Ergebnisse

Bisher haben wir gesehen, wie sich Konfigurationen eines Multikonfigurationsprojekts effizient anlegen und verteilt parallelisiert ausführen lassen. Aber wie behält man den Überblick über den Ausgang dieser zahlreichen Matrix-Builds?

Hudson trägt automatisch die einzelnen Ergebnisse der Matrix-Builds auf dem Master wieder zusammen, archiviert diese dort und berechnet ein Gesamtergebnis des Builds des Multikonfigurationsprojekts. Dieser wird vom schlechtesten Resultat aller Matrix-Builds bestimmt. Schlägt also mindestens ein Matrix-Build fehl, gilt auch der Gesamt-Build als fehlgeschlagen.

*Ermittlung eines
Gesamtergebnisses*

Hudson bewahrt detaillierte Build-Ergebnisse für jede Konfiguration getrennt auf. Über die Build-Übersichtsseite des Multikonfigurationsprojekts können Sie durch Klicken auf den jeweiligen Ball einer Konfiguration zu deren Builds navigieren. Auf Ebene des Dateisystems

*Matrix-Builds werden
getrennt aufbewahrt.*

enthalten Multikonfigurationsprojekte in ihrem Jobverzeichnis dazu einen eigenen Unterordner `configurations`, in welchem die Ergebnisse der Matrix-Builds liegen. Sollten Sie am Ende eines Builds Artefakte archivieren, werden diese ebenfalls pro Konfiguration in dieser Verzeichnisstruktur abgelegt.

8.5 Verteilte Builds

Je schneller Continuous Integration in Ihrem Umfeld Akzeptanz findet, desto früher werden Sie an den Punkt gelangen, an dem Sie Ihre Build-Aktivitäten auf mehrere Rechner verteilen möchten. Dafür sprechen viele gute Gründe – gleichzeitig nimmt damit jedoch die Komplexität des Gesamtsystems zu. Wir betrachten daher in den nächsten zwei Abschnitten zunächst die Chancen und Risiken verteilter Builds.

8.5.1 Warum verteilen?

Verteilte Builds leisten gute Dienste in folgenden, typischen Anwendungsfällen:

- Builds sollen *parallelisiert* ausgeführt werden, um lange Warteschlangen zu vermeiden.
- Builds sollen auf *heterogenen Infrastrukturen* ausgeführt werden, z.B. um Tests unter unterschiedlichen Betriebssystemen auszuführen oder architekturnspezifische Compiler auszunutzen.
- Das CI-System soll *robuster* gegen Ausfälle durch den Einsatz redundanter Build-Knoten werden.
- Die Anzahl der Slave-Knoten soll *dynamisch skalierbar* sein – je nach momentanem Bedarf. Somit muss die Hardware eines Hudson-Servers nicht für seltene Spitzenlasten ausgelegt sein, die vielleicht nur wenige Male im Jahr erreicht werden.
- Ausgewählte Projekte mit hoher Priorität sollen *exklusive* Build-Ressourcen benutzen können.

8.5.2 Warum nicht verteilen?

Bei allen Vorzügen verteilter Builds soll nicht verschwiegen werden, dass damit die Gesamtkomplexität des CI-Systems deutlich zunimmt. Dies bedeutet unter anderem:

- Build-Manager müssen sich besser in Hudson auskennen (Schulungsaufwand).

- Das Einrichten der Slave-Knoten erfordert zusätzlichen Konfigurationsaufwand.
- Administrative Aufgaben nehmen mit der Anzahl der eingesetzten Knoten zu, z.B. Verfügbarkeit überwachen, Updates einspielen, Build-Umgebung einrichten, Systemsicherheit gewährleisten, Sicherungen erstellen usw.
- Der Netzauslastung nimmt ebenfalls mit der Anzahl der eingesetzten Knoten zu, da mehr Daten zwischen beteiligten Rechnern ausgetauscht werden müssen.

Der Vollständigkeit halber soll also festgehalten werden, dass für sehr kleine Installationen eine Konzentration auf einen Server durchaus sinnvoll sein kann. In der Praxis entwickeln sich jedoch die meisten Hudson-Instanzen im Laufe der Zeit zu einer verteilten Installation.

8.5.3 Slave-Knoten einrichten

Die Verteilung von Builds erfolgt bei Hudson nach dem Master-Slave-Prinzip, d.h., der Hudson-Server fungiert als zentraler Master-Knoten, der Build-Aufträge an einen oder mehrere Slave-Knoten delegiert. Auf dem Slave-Knoten wird dazu keine weitere, vollständige Hudson-Instanz installiert, sondern lediglich ein Slave-Agent von geringer Größe (`slave.jar`, ca. 300 KB). Über die bidirektionale Verbindung zum Slave-Agent kann der Hudson-Master Daten und Befehle an den Slave-Knoten übertragen, umgekehrt aber auch über Ereignisse auf dem Slave-Knoten informiert werden.

Zusätzliche Slave-Knoten richten Sie unter *Hudson → Hudson verwalten → Knoten verwalten → Neuer Knoten* ein. Sind diese Knoten angelegt, arbeitet Hudson automatisch im Master-Slave-Modus. Sie müssen also keine weiteren Schritte unternehmen, um verteilte Builds zu erstellen. Ohne weitere Einstellung beginnt Hudson, alle geplanten Projekte auf alle verfügbaren Knoten zu verteilen. Wenn Sie hingegen ein Projekt einem bestimmten Knoten zuordnen möchten, können Sie dies in der Jobkonfiguration mittels der Option *Binde dieses Projekt an einen bestimmten Knoten* angeben.

Die Konfiguration eines Slave-Knotens erfordert nur wenige Parameter (siehe Abb. 8–22):

Neue Knoten einrichten

Konfiguration eines Slave-Knoten

Abb. 8-22
Konfiguration eines
Slave-Knotens

Name Der Name des Knoten muss innerhalb einer Hudson-Instanz eindeutig gewählt werden. In der Praxis hat es sich bewährt, hier den Namen des Rechners zu verwenden.

Beschreibung Die kurze Beschreibung des Knotens kann dessen charakteristische Eigenschaften enthalten, z.B. »Windows 7, Oracle 11i, Firefox«. Die Beschreibung wird in der Projektkonfigurationsseite angezeigt und erleichtert somit die Zuordnung eines Projekts zu speziellen Knoten.

Anzahl der Build-Prozessoren Die Anzahl der Build-Prozessoren gibt an, wie viele Builds gleichzeitig auf einem Rechner gebaut werden können. Es handelt sich dabei also *nicht* um die Anzahl der physikalisch vorhandenen Prozessoren (obwohl dies einen guten Ausgangspunkt darstellt). Tipp: Möchten Sie Builds ausschließlich auf Slave-Knoten ausführen, setzen Sie diesen Wert für den Master-Knoten auf 0.

Stammverzeichnis im entfernten Dateisystem Hudson legt auf einem Slave-Knoten alle Dateien unterhalb eines Stammverzeichnisses ab. Dies umfasst nicht nur die Arbeitsverzeichnisse der einzelnen Jobs, sondern auch automatisch installierte Ant- oder Maven-Versionen bzw. JDKs. Da der Master-Knoten nach einem verteilten Build wichtige Dateien von den Slave-Knoten einsammelt und auf dem Master zentral und dauerhaft speichert, kann dieses Stammverzeichnis auf dem Slave-Knoten problemlos gelöscht werden – während des nächsten Builds wird das Verzeichnis automatisch neu initialisiert.

Labels Knoten können über Labels in logische Gruppen eingeteilt werden. Ein Knoten kann auch mehrere Labels tragen, in diesem Fall gehört er dann gleichzeitig zu mehreren Gruppen. Ein Eintrag von »windows oracle firefox« würde einen Knoten also drei Gruppen zuweisen. Da Sie Projekte nicht nur einzelnen Rechnern, sondern auch Labels zuweisen können, geben Sie Hudson mehr Freiheiten bei der Ablaufplanung

der Builds. Beispiel: Sie haben fünf Knoten das Label »windows« zugewiesen und ein Projekt an dieses Label gebunden. Hudson wird einen neuen Build des Projektes automatisch auf dem nächsten freien Rechner dieser Fünfergruppe ausführen.

Mit der Einstellung *Auslastung* geben Sie an, wie Hudson den Knoten mit Build-Aufträgen beschicken kann. Momentan gibt es zwei Strategien: *Diesen Rechner so viel wie möglich verwenden* bedeutet, dass alle anstehenden Builds auf diesem Knoten gebaut werden können. Das Gegenteil dazu ist *Diesen Rechner exklusiv für gebundene Jobs reservieren*, welches einen Knoten für ausgewählte Projekte freihält.

Auslastung

Die Startmethode entscheidet darüber, wie sich Master- und Slave-Knoten im Netz finden. Damit ist meistens auch die Frage verbunden, ob der Slave-Agent vom Hudson-Master gestartet werden soll oder ob er extern verwaltet wird. Momentan können Sie unter vier Optionen wählen:

Startmethode

■ *Java Network Launch Protocol (JNLP):*

Bei dieser Option startet der Anwender den Slave-Agenten selbst, typischerweise durch Anklicken der Java-Web-Start-Schaltfläche *Launch* auf der Übersichtsseite des Knotens.

■ *Ausführung eines Kommandos auf dem Master:*

Sie führen auf dem Master ein beliebiges Kommando aus, das im Ergebnis den Slave-Agenten auf dem Slave-Knoten startet. Da das Kommando auf dem Master-Knoten und nicht auf dem Slave-Knoten ausgeführt wird, muss letzterer zum Zeitpunkt der Ausführung noch nicht zwingend laufen. Diese Option ermöglicht es daher beispielsweise, vom Master-Knoten aus den Slave-Knoten zunächst einzuschalten – sei es ein in Hardware existierender Rechner oder aber eine virtuelle Maschine innerhalb einer Virtualisierungslösung (siehe auch Abschnitt 8.5.6).

■ *Windows-Slave als Windows-Dienst:*

Der Master versucht über das Windows Management Interface (WMI) den Slave-Agenten auf dem Salve-Knoten zu starten und als Windows-Dienst zu betreiben.

■ *Agenten per SSH starten:*

Diese Option eignet sich hervorragend für Slave-Knoten, die unter Unix-ähnlichen Betriebssystemen betrieben werden. Der Master-Knoten meldet sich hierbei per SSH am Slave-Knoten an, kopiert zunächst den Slave-Agenten auf den Slave-Knoten und startet diesen abschließend dort. Es müssen daher keine lokalen Installationsarbeiten auf dem Slave-Knoten vorausgegangen sein: Hudson bringt hier alles mit, was auf dem Slave-Knoten benötigt wird.

Verfügbarkeit Wie lange soll der Slave-Agent auf einem Slave-Knoten laufen? Unter *Verfügbarkeit* können Sie entweder keine Einschränkung vornehmen (*Slave immer angeschaltet lassen*) oder aber eine Zu- und Abschaltung nach Bedarf bzw. Zeitplan konfigurieren. Hudson startet dann automatisch neue Agenten, wenn ein Job bereits eine gewisse Zeit in der Build-Warteschlange warten musste. Umgekehrt können Agenten bei anhaltender Untätigkeit auch wieder abgeschaltet werden.

Eigenschaften des Knotens Im Abschnitt *Eigenschaften des Knotens* können Sie knotenspezifische Anpassungen vornehmen, z.B. indem Sie den Wert von Umgebungsvariablen pro Knoten individuell anpassen. Auf diese Weise können Sie einheitliche Build-Skripte erstellen, die unverändert auf unterschiedlichen Rechnern (ggf. sogar mit unterschiedlichen Betriebssystemen) ausgeführt werden können.

8.5.4 Verteilen der Build-Umgebung

Das Konzept verteilter Builds klingt sehr einleuchtend: Auszuführende Arbeit wird auf einen Slave-Knoten transportiert, dort erledigt und die Ergebnisse werden auf den Master-Knoten zurückgeschickt. Gerne wird dabei übersehen, dass für diese Vorgehensweise auch auf jedem Slave-Knoten eine ausreichende Build-Umgebung installiert sein muss: also mindestens ein JRE, um den Slave-Agenten zu starten, sowie weitere Werkzeuge wie Compiler, Testwerkzeuge, Archivierungsprogramme, Signaturdateien und so weiter – je nach Anforderungen der zu bauenden Projekte.

Automatische Installation von Ant, Maven und JDKs

Ant, Maven und JDKs bilden die Grundlage der meisten Java-Projekte. Deshalb bietet Hudson in seiner Systemkonfiguration unter *Hudson → Hudson verwalten → System konfigurieren* in den jeweiligen Abschnitten *Ant, Maven* und *JDK* an, diese Software automatisch auf Knoten zu installieren (unabhängig davon, ob es ein Master- oder Slave-Knoten ist). Die eigentliche Installation kann dabei auf unterschiedliche Weisen vollzogen werden. Die »eingebauten« drei Varianten werden im Folgenden näher gezeigt (weitere können über Plugins hinzugefügt werden; siehe dazu Erweiterungspunkt *Tool Installer* in Kapitel 9).

**Abb. 8-23**

Automatische Installation aus dem Internet (hier: Maven)

Wir betrachten dabei die drei Installationsweisen für Maven, die analog auch für Ant und JDKs zur Verfügung stehen:

■ *Automatische Installation aus dem Internet:*

Am wenigsten Vorbereitung erfordert eine Installation direkt aus dem Internet (Abschnitt 8.5.6). Der Hudson-Update-Server stellt dazu eine Liste aller verfügbaren Maven-Versionen und deren Download-Links bereit. Existiert die gewählte Maven-Version noch nicht lokal auf dem ausführenden Knoten, lädt Hudson das Installationsarchiv zunächst herunter und installiert es dann unter `<HUDSON_HOME>/tools`. Es liegt in der Natur der Sache, dass dieses Verfahren einen Internetzugang voraussetzt. Zudem wird dabei automatisch heruntergeladene Software zu einem essenziellen Teil der Build-Werkzeugkette, was insbesondere bei sicherheitsbewussten Administratoren Unwohlsein auslösen könnte.

■ *Ausführung eines Kommandos:*

Glücklicherweise bietet Hudson für dieses Problem bereits eine Lösung an: die Installation über ein lokal ausgeführtes Kommando bzw. Skript. Dadurch lassen sich Installationsprozeduren in Hudson einbinden, welche auch weitergehenden Sicherheitsanforderungen genügen, z.B. die ausschließliche Verwendung signierter Installationsarchive aus firmeninternen Softwarekatalogen.

■ *Entpacke *.zip/*.tar.gz-Archiv:*

Besteht der Installationsvorgang nur aus dem Kopieren einer Verzeichnisstruktur an einen bestimmten Platz im lokalen Dateisystem, kann das Installationsverfahren »Entpacke *.zip/*.tar.gz-Archiv« zum Einsatz kommen. Es erfordert als Konfiguration lediglich eine URL, von der das Installationsarchiv heruntergeladen werden soll. Diese URL muss übrigens nur vom Master-Knoten aus erreichbar sein, nicht aber von Slave-Knoten, da der Master-Knoten das Herunterladen und Verteilen übernimmt. Hudson vergleicht den serverseitigen Zeitstempel der URL mit dem der lokal installierten Version und lädt das Installationsarchiv nur

erneut herunter, wenn auf dem Server Aktualisierungen vorgenommen wurden.

*Wahl des
Installationsverfahrens
per Label*

Setzen Sie plattformspezifische Werkzeuge ein, kann sich das erforderliche Installationsverfahren von Knoten zu Knoten unterscheiden. Für die beiden letztgenannten Installationsverfahren (Ausführung eines Kommandos, Entpacken eines Archivs) können Sie daher zusätzlich Labels spezifizieren. Anhand der Labels kann Hudson dann die benötigten Werkzeuge im passenden Verfahren durchführen.

*Weitere Ansätze zur
Synchronisation der
Build-Umgebungen*

Um die Build-Umgebung auf vielen Rechnern identisch und aktuell zu halten, hat es sich oftmals auch als praktikabel erwiesen, auf einem Netzlaufwerk ein Verzeichnis mit allen benötigten Werkzeugen einzurichten und dieses für alle Slave-Knoten freizugeben. Eine etwas abgeschwächte Variante basiert ebenfalls auf einem zentralen Verzeichnis mit allen benötigten Werkzeugen, das aber – im Gegensatz zu einer Netzfrezigabe – auf jeden Slave-Knoten *kopiert* wird. Dadurch werden Slave-Knoten von Änderungen im zentralen Werkzeugverzeichnis entkoppelt, was erwünscht sein kann (Reduktion störender Einflüsse auf einen Knoten), aber einen Mehraufwand für den Administrator bedeutet (nach jeder Änderung müssen erneut Kopiervorgänge angestoßen werden).

8.5.5 Überwachen des CI-Clusters

Jeder Systemadministrator kann ein Lied davon singen: Die Überwachung eines Rechnerparks ist nicht trivial. Das gilt selbstverständlich auch für ein CI-Cluster, also ein CI-System mit zahlreichen Slave-Knoten: Sind alle Knoten übers Netz erreichbar? Ist ausreichend freier Platz auf allen Festplatten vorhanden? Laufen alle Systemuhren synchron? Diese und weitere technische Größen wollen rund um die Uhr überwacht sein.

Über den Link *Build-Prozessor Status* am linken Seitenrand der Übersichtsseite von Hudson gelangen Sie deshalb zu einer Zusammenstellung aller angelegter Knoten sowie deren überwachten Betriebsgrößen (Abb. 8–24). Für diese Größen sind konfigurierbare Grenzwerte hinterlegt, bei deren Nichteinhaltung Hudson den betreffenden Knoten »offline« schaltet, ihm also keine weiteren neuen Builds zuordnet. In Hudson wird diese Funktion als »präventive Überwachung« bezeichnet, weil sie vermeidet, dass Builds in einer Umgebung gestartet werden, die keinen Erfolg erwarten lässt.

Über Plugins lassen sich weitere zu überwachende Betriebsgrößen hinzufügen – denkbar wären Temperatur, momentane CPU-Auslastung, Anzahl angemeldeter Benutzer usw.

S	Name	I	Antwortzeit	Freier Plattenplatz	Freier TEMP-Platz	Architektur	Zeitdifferenz	
	master		1ms	5GB	16GB	Windows XP (x86)	synchron	
	macbook0001		129ms	75GB	75GB	Mac OS X (x86_64)	3 Minuten 39 Sekunden vorgehend	
	windows0001		33ms	16GB	16GB	Windows XP (x86)	synchron	

[Status aktualisieren](#)

Abb. 8-24

Darstellung angelegter Knoten mit kritischen Betriebsgrößen

8.5.6 Virtualisierung

Unter Virtualisierung versteht man die Loslösung eines Rechners von einer real existierenden Hardware. Alle nicht flüchtigen Informationen wie etwa Festplatteninhalt, BIOS-Einstellungen, ggf. Inhalt des Hauptspeichers usw. werden in einer Datei gespeichert, die als Abbild oder *image* bezeichnet wird. Der Rechner existiert somit nicht mehr als Kombination aus Hardware und Daten, sondern lediglich als große Abbild-Datei auf einer Festplatte. Dieses Abbild lässt sich dann auf einer virtuellen Hardware ausführen. Dadurch kann auf einem Server mit Betriebssystem A ein zweiter, virtueller Rechner mit gänzlich unterschiedlichem Betriebssystem B ausgeführt werden. Ist der Server ausreichend leistungsstark, können sogar mehrere virtuelle Rechner gleichzeitig betrieben werden. Der Server wird im Umfeld der Virtualisierung als Wirt (*host*) bezeichnet, der virtuelle Rechner hingegen als Gast (*guest*) oder virtuelle Maschine (*virtual machine*).

Zu den bekanntesten Herstellern von Virtualisierungssoftware dürfen momentan VMware und Parallels gehören. Auch Oracle bzw. Microsoft bieten mit VM VirtualBox bzw. Virtual PC/Hyper-V Virtualisierungsprodukte an.

Das Prinzip der Virtualisierung erschließt im Zusammenhang mit Continuous Integration gleich eine ganze Reihe von Vorteilen:

Vorteile der
Virtualisierung

- Da ein virtueller Rechner im ausgeschalteten Zustand vollständig als Abbild in Dateiform vorliegt, lässt sich durch Kopieren dieses Abbilds auf einfachste Weise ein Schnappschuss seiner momentanen Konfiguration anlegen. Durch Zurückkopieren des Schnappschusses kann immer wieder zu diesem definierten Ausgangszustand zurückgekehrt werden. Auf diese Weise kann beispielsweise die Installation einer Software auf einem Rechner mit frisch installiertem Betriebssystem getestet werden. Nach dem Test wird der virtualisierte Rechner wieder zurückgesetzt und steht – täglich grüßt das Murmeltier – zum nächsten Test im bekannten Ausgangszustand bereit.

- Werden viele unterschiedliche Rechnerkonfigurationen benötigt und müssen diese nicht parallel zur Verfügung stehen, so können die Konfigurationen sequenziell als virtuelle Rechner auf derselben Hardware eines Wirtsrechners ausgeführt werden. So lässt sich etwa eine Anwendung nacheinander in zehn virtualisierten Windows-Varianten auf einem Wirtssystem testen, ohne dass dazu zehn real existierende Rechner benötigt werden. Das senkt nicht nur Anschaffungskosten, sondern verbilligt auch den Betrieb: Im schlimmsten Fall würden die zehn real existierenden Rechner vielleicht ja die meiste Zeit untätig, aber stromfressend vor sich hindümpeln.
- Benötigen Sie zusätzliche virtuelle Rechner im parallelen Betrieb, reicht es aus, den Wirtsrechner aufzurüsten und mehr Guestsysteme zu starten. An den virtuellen Rechnern selbst hingegen müssen keine Konfigurationsänderungen vorgenommen werden.

VMware Server

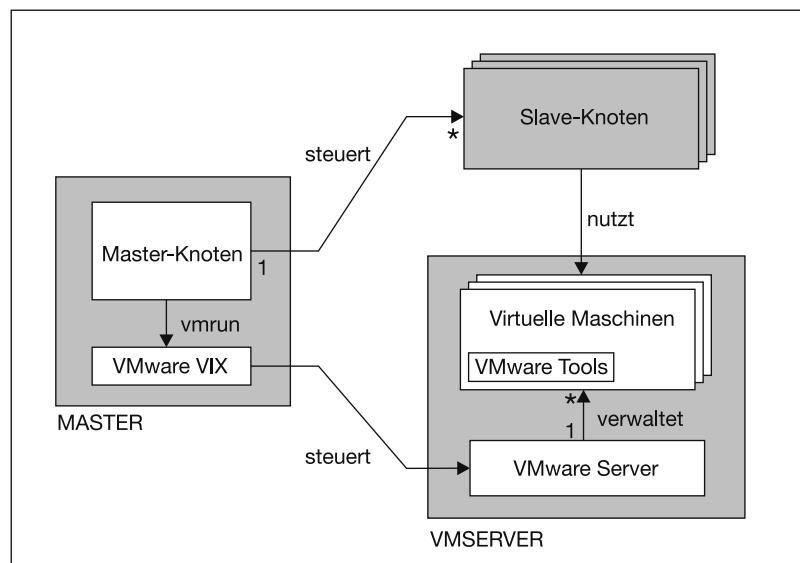
Im folgenden Beispiel betrachten wir exemplarisch das Zusammenspiel von Hudson und VMware Server. Wir verwenden VMware-Servert, weil es sich um eine sehr verbreitete Virtualisierungslösung handelt, die in Produktionsumgebungen zuverlässig eingesetzt werden kann und zudem kostenfrei vom Hersteller bereitgestellt wird. Mit Produkten anderer Hersteller lassen sich vergleichbare Ergebnisse erzielen, lediglich aus Platzgründen beschränken wir uns in diesem Buch auf einen prominenten Vertreter.

VMware-Konzepte

Dazu lernen wir zunächst die benötigten VMware-Komponenten kennen, deren Zusammenspiel mit Hudson in Abbildung 8-25 dargestellt ist:

Abb. 8-25

Zusammenspiel von
Hudson, Slave-Knoten
und virtualisierten
Rechnern



- *VMware Server* ist das Herzstück der Virtualisierungslösung. Diese Software betreibt eine oder mehrere virtuelle Maschinen und stellt Managementfunktionen nach außen bereit. So können Sie beispielsweise über eine Weboberfläche virtuelle Maschinen einrichten, starten, überwachen oder stoppen. Detaillierte Informationen zu Installation und Betrieb des Servers sowie zur Einrichtung von virtuellen Maschinen entnehmen Sie der Dokumentation des VMware Servers.
- *VMware Tools* werden optional auf den Gastsystemen installiert und verbessern zum einen die Geschwindigkeit der virtuellen Systeme. Sie bieten aber auch zusätzlich erweiterte Zugriffsmöglichkeiten von außen, etwa den Austausch von Dateien zwischen Gast und Wirt, die Abfrage laufender Prozesse, das Starten von Programmen innerhalb des Gastes usw. Obwohl die Installation der VMware Tools nicht zwingend notwendig ist, empfiehlt sie sich aufgrund der genannten zusätzlichen Funktionen. Die VMware Tools sind im VMware Server enthalten.
- *VMware VIX* ist eine kostenfreie Sammlung von Werkzeugen, um VMware-Produkte fernzusteuern, und enthält unter anderem die Kommandozeilenanwendung `vmrun`, mit der sich sehr einfach virtuelle Maschinen in einem VMware Server starten und stoppen lassen – auch wenn VMware VIX und VMware Server auf unterschiedlichen Rechnern installiert sind. Sie müssen dabei allerdings beachten, dass nicht jede Version von VMware VIX mit jeder Version von VMware Server zusammenarbeitet: Die unterstützten Kombinationen entnehmen Sie der Dokumentation zu VMware VIX, die Sie gemeinsam mit der Software selbst kostenfrei von der VMWare-Website herunterladen können. Die Beispiele in Abbildung 8–1 sollen lediglich einen Eindruck davon vermitteln, wie einfach es ist, aus der Kommandozeile heraus virtuelle Maschinen zu verwalten. Darüber hinaus erlaubt es VMware VIX auch, Dateien zwischen Wirt und Gast auszutauschen, die laufenden Prozesse auf dem Gast abzufragen, einen Bildschirmschnappschuss des Gastes abzuspeichern und noch vieles mehr. Eine komplette Auflistung aller Funktionen und deren Parameter finden Sie im VMware-VIX-Benutzerhandbuch.

Starten einer virtuellen Maschine

```
vmrun -T server -h https://vmserver:8333/sdk -u root -p GeHeiM start  
"[standard] ubuntu-9.04/ubuntu-9.04.vmx"
```

Listing 8–1

Beispiele für die
Verwendung von `vmrun`

Stoppen einer virtuellen Maschine

```
vmrun -T server -h https://vmserver:8333/sdk -u root -p GeHeiM stop  
"[standard] ubuntu-9.04/ubuntu-9.04.vmx" hard
```

```
# Zurücksetzen einer virtuellen Maschine
vmlinuz -T server -h https://vmserver:8333/sdk -u root -p GeHeiM \
revertToSnapshot "[standard] ubuntu-9.04/ubuntu-9.04.vmx"
```

*Anwendungsbeispiel:
Testen eines
Installationsprogramms*

In unserem fiktiven Beispiel möchten wir testen, ob sich ein von uns entwickeltes Softwareprojekt fehlerfrei auf einem frisch aufgesetzten Linux-System installieren lässt. Pro Build möchten wir also ein virtuelles Linux-System starten und dort das Installationsprogramm unserer Software ausführen. Danach soll das Linux-System für den nächsten Test wieder in den Ausgangszustand zurückversetzt werden. Wie lässt sich also dieses Szenario mithilfe von virtuellen Testrechnern umsetzen?

Umsetzung

Um die Anforderungen unseres Beispiels zu erfüllen, gehen wir in folgenden Schritten vor (die Rechnernamen in Großbuchstaben beziehen sich auf eine Infrastruktur wie in Abb. 8–25 dargestellt):

1. Vor Beginn der Testinstallation startet der Hudson-Master-Knoten auf MASTER über VIX eine virtuelle Maschine auf VMSERVER.
2. Hudson wartet, bis diese virtuelle Maschine hochgefahren ist.
3. Hudson installiert per SSH-Zugang unsere Software auf der virtuellen Maschine und merkt sich den Ausgang des Installationsvorgangs.
4. Hudson beendet über VIX die virtuelle Maschine auf VMSERVER und setzt sie wieder auf einen definierten Ausgangszustand zurück.
5. Je nach Ausgang des Installationsvorgangs wird in Hudson der Status des Builds als erfolgreich oder fehlgeschlagen gesetzt.

Listing 8–2 zeigt die Umsetzung dieser Schritte als Shell-Skript, wie Sie es innerhalb eines Build-Schrittes im Rahmen eines Free-Style-Projektes verwenden könnten:

Listing 8–2

*Shell-Skript zur
Ausführung einer*

*Probeinstallation in einer
virtuellen Maschine*

```
# Starten einer virtuellen Maschine
vmlinuz -T server -h https://vmserver:8333/sdk -u root -p GeHeiM \
start "[standard] ubuntu-9.04/ubuntu-9.04.vmx"

# 120 Sekunden warten, bis virtuelle Maschine hochgefahren ist
sleep 120

# Kopiere Installationsprogramm auf virtuelle Maschine
# (installer.tag.gz wurde im vorherigen Build-Schritt erzeugt)
scp installer.tar.gz root@ubuntu:/tmp

# Entpacke Installationsprogramm auf virtueller Maschine
ssh root@ubuntu tar xzf /tmp/installer.tar.gz

# Installationsprogramm ausführen und Ergebnis merken
ssh root@ubuntu /tmp/installer/install.sh
RESULT_CODE!=#
```

```
# Stoppen und zurücksetzen einer virtuellen Maschine  
vmrun -T server -h https://vmserver:8333/sdk -u root -p GeHeiM \  
revertToSnapshot "[standard] ubuntu-9.04/ubuntu-9.04.vmx"  
  
# Ergebnis zurückgeben  
return $RESULT_CODE
```

Betrachten wir die einzelnen Schritte in Abbildung 8–2 genauer: Zunächst erfolgt das Starten der virtuellen Maschine über einen Aufruf des Kommandozeilenprogramms `vmrun`. Im gezeigten Beispiel starten wir die virtuelle Maschine `ubuntu-9.04` auf `VMSERVER`.

Im Anschluss wird gewartet, bis die virtuelle Maschine hochgefahren ist. Im gezeigten Beispiel ist das sehr rudimentär durch eine Pause festgelegter Länge gelöst. Wenn das Gastbetriebssystem einen Netzwerkzugang anbietet, könnten Sie an dieser Stelle auch periodisch den Zustand des Systems von außen abfragen (z.B. mit `ssh`) und so lange warten, bis etwa bestimmte Prozesse gestartet wurden. Weitere Möglichkeiten eröffnen sich, wenn im Guestsystem die VMware Tools installiert wurden. In diesem Fall können Sie direkt über VMware VIX den Zustand der virtuellen Maschine von außen beobachten – etwa ob ein benötigter Prozess bereits läuft oder eine bestimmte Datei angelegt wurde, die den Abschluss eines erfolgreichen Systemstarts anzeigen. Die entsprechenden Kommandos entnehmen Sie dem VMware-VIX-Benutzerhandbuch.

Anschließend wird das Installationsprogramm auf das Testsystem kopiert und ausgeführt. Da wir im Beispiel auf einem Linux-System arbeiten, können wir hier `scp` und `ssh` verwenden. Eine fehlerfreie Installation soll in unserem Beispiel daran erkannt werden können, dass das Installationsprogramm einen Rückgabewert von 0 zurückgibt. Wir speichern den Rückgabewert in der Variable `INSTALLATION_RESULT`.

Danach kann die virtuelle Maschine gestoppt und auf den Zustand `snapshot` zurückgesetzt werden. Dies geschieht wieder mit dem Kommando `vmrun`. Das Zurücksetzen auf einen bestimmten Zustand (*snapshot*) schließt das Stoppen der Maschine mit ein.

Abschließend wird der Inhalt der Variable `INSTALLATION_RESULT` als Ergebnis an Hudson zurückgegeben. Ein Wert von 0 wird von Hudson als Erfolg verstanden, jeder andere als Fehlschlag interpretiert.

Es existiert bereits ein Plugin, das die bisher beschriebenen Schritte kapselt und mit einer benutzerfreundlichen Oberfläche versieht. Leider hat das Plugin seit April 2008 keine Aktualisierung mehr erhalten und arbeitet daher in seiner momentanen Fassung nur mit VMware Server Version 1.0 zusammen. Kein Grund zur Besorgnis: Sie können den Funktionsumfang des Plugins durch Aufrufe des Kommandos `vmrun` wie oben beschrieben nachbilden.

Plugin »VMware«

Hudsons Slave-Knoten auf virtualisierten Rechnern

In den obigen Abschnitten haben wir virtualisierte Rechner lediglich betrachtet, um eine geeignete Umgebung für die Ausführung von Tests zu schaffen. Könnte man nicht auch die Hudson-Slaves selbst auf virtualisierten Rechnern betreiben? Selbstverständlich: Laufen die virtualisierten Rechner im Dauerbetrieb, so legen Sie dazu wie gewohnt einfach weitere Slave-Knoten an. Für Hudson macht es keinen Unterschied, ob der Knoten auf realer oder virtualisierter Hardware betrieben wird.

Soll Hudson hingegen einen Slave-Knoten auf virtualisierter Hardware automatisiert starten können, so lässt sich dazu in der Konfiguration des Slave-Knotens die Startmethode *Starte Slave durch Ausführung eines Kommandos auf dem Master einsetzen*. Als auszuführendes Kommando geben Sie einen `vmrun`-Startbefehl nach oben bereits gezeigtem Schema an. Im einfachsten Fall wurde Hudsons Slave-Agent `slave.jar` bereits zuvor im Abbild des Gastsystems installiert und kann so beim Hochfahren des Gastsystems automatisch gestartet werden. Dadurch wird die Verbindung zum Hudson-Master hergestellt, und der »virtuelle Slave-Knoten« ist einsatzbereit.

8.5.7 Bauen in der Wolke

Eine besondere Spielart der Virtualisierung stellt *cloud computing* dar, bei der virtuelle Maschinen auf gemieteter Hardware ausgeführt werden. Statt also Server im eigenen Rechenzentrum vorzuhalten, mietet man sich bei einem Cloud-Computing-Anbieter Rechenzeit, die in der Regel stundengenau abgerechnet wird. Üblich sind hier Stundenpreise von 0,08–3,00 Euro pro virtualisiertem Rechner (Stand Mitte 2010), je nach Ausstattung der virtuellen Maschine (Rechenleistung, Hauptspeicher, vorinstalliertes Betriebssystem). Zusätzlich werden Gebühren für den Datenverkehr von und zum Rechenzentrum des Anbieters fällig.

Beispiel: Amazon Web Services (AWS)

Einer der wichtigsten Anbieter in diesem Bereich dürfe die Firma Amazon sein, die neben der bekannten Handelsplattform für Verbraucher unter dem Begriff »Amazon Web Services (AWS)« (<http://aws.amazon.com>) auch rund 20 unterschiedliche IT-Dienstleistungen für Unternehmen anbietet. Wir betrachten am Beispiel der AWS, wie sich Cloud Computing mit Hudson nutzen lässt. Dabei spielen drei AWS-Angebote eine besondere Rolle:

Elastic Compute Cloud (EC2)

Elastic Compute Cloud (EC2) erlaubt die Ausführung virtueller Maschinen (»EC2-Instanzen«) und ist somit das wichtigste der betrachteten Angebote. Amazon bietet dazu über 5.000 öffentliche Amazon Machine Images (AMI) an, auf denen unterschiedliche Linux- und Windows-Betriebssysteme vorinstalliert sind. Als EC2-Anwender

können Sie ein solches Image über eine webbasierte Verwaltungskonsole starten. Nach etwa einer Minute steht Ihnen dann ein virtueller Rechner auf Basis dieses Images zur Verfügung, der sich über das Internet erreichen lässt. Im Umgang unterscheidet sich dieser Rechner grundsätzlich nicht von einem physikalisch existierenden Server. Das bedeutet, Sie können sich wie gewohnt an diesem System anmelden, Software installieren, Dateien hoch- und herunterladen usw. Im Gegensatz zu einem physikalisch existierenden System sind jedoch alle Änderungen flüchtig. Nach dem Abschalten der virtuellen Maschine gehen also Informationen verloren, die während des Betriebs erzeugt wurden. Der EC2-Anwender muss also beim Starten einer virtuellen Maschine selbst dafür sorgen, dass zunächst alle erforderliche Software und benötigte Daten auf diese virtuelle Maschine übertragen werden. Umgekehrt muss er vor dem Herunterfahren alle wichtigen Daten über das Netz auf andere Rechner sichern. Bei gelegentlicher Nutzung und einfachen Szenarien ist dieses Vorgehen praktikabel. Für größere Installationen bietet sich hingegen die Kombination mit weiteren Diensten an.

Simple Storage Service (S3) stellt Speicherkapazität in Amazons Rechenzentren bereit. Vereinfacht ausgedrückt, mieten Sie sich dabei ein Stück Festplattenplatz. S3 wird im Zusammenhang mit EC2 interessant, weil Sie diesen gemieteten Speicherplatz dazu nutzen können, um eigene Rechner-Images abzulegen und diese als virtuelle Maschinen zu starten. Dadurch wird es möglich, sich nicht nur öffentlich zugängliche Images, sondern ein System nach spezifischen Wünschen einzurichten (z.B. mit speziellen Datenbank- und Applikationsservern), davon ein Image zu erstellen und fortan direkt neue virtuelle Maschinen mit diesem Stand zu starten.

Dies löst allerdings nicht das Problem der flüchtigen Daten beim Herunterfahren einer virtuellen Maschine: Hierbei kommt der Elastic Block Storage zu Hilfe.

Elastic Block Storage (EBS) bietet virtuelle Datenträger für die Speicherung auf Blockebene zur Verwendung mit EC2-Instanzen. EBS stellt also Speicherplatz bereit, der sich in Ihren virtuellen Maschinen wie eine Festplatte verwenden lässt. Dieser Speicherplatz bleibt auch nach dem Abschalten der virtuellen Maschine erhalten. Die Notwendigkeit, wichtige Daten vor dem Herunterfahren einer virtuellen Maschine auf andere Rechner zu sichern, entfällt somit bei der Nutzung von EBS. Die Preisgestaltung richtet sich wie bei S3 nach der Menge des angemieteten Speicherplatzes.

Simple Storage Service (S3)

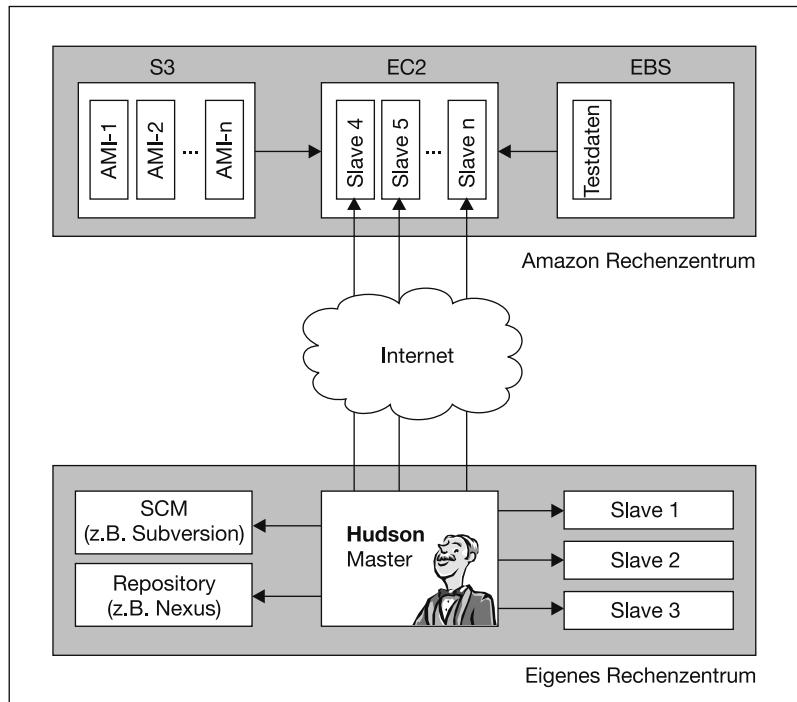
Elastic Block Storage (EBS)

CI mit AWS

Abbildung 8–26 zeigt ein typisches Zusammenspiel der vorgestellten AWS-Dienste. Im dargestellten Szenario sollen Tests auf Slave-Knoten ausgeführt werden, die auf EC2-Instanzen laufen.

Abb. 8–26

Zusammenspiel der AWS-Dienste mit Hudson-Slave-Knoten in der Wolke



Was gehört in die Wolke?

Man kann an der Darstellung in Abbildung 8–26 außerdem gut sehen, dass eine CI-Infrastruktur stets einen *Verbund* mehrerer Rechnersysteme darstellt und es gut überlegt sein will, welche Teile man lokal am Standort hält und welche man in die Wolke auslagert. Dienste wie etwa Versionskontrollsysteme für Quelltexte sollten in der Regel rund um die Uhr erreichbar sein und gleichzeitig maximal gegen den Zugriff Unbefugter geschützt sein. Aus Kosten- und Sicherheitsüberlegungen scheidet hier daher oftmals ein Betrieb in der Wolke aus. Auf der anderen Seite benötigen Slave-Knoten, die Quelltexte komplizieren sollen, eine Verbindung zum Versionskontrollsysteem und gegebenenfalls noch zu weiteren Repositories (z.B. bei Maven-Projekten). Sollen die Slave-Knoten also in der Wolke laufen, müssen die Server am eigenen Standort vom Internet aus »sichtbar« sein. Mit dieser Konstellation fühlen sich nicht nur paranoide Administratoren unwohl, sie bedeutet auch zusätzlichen Netzverkehr und – je nach Datenmenge – deutlich langsamere Builds. In Cloud-Computing-Szenarien ist der Build-Manager also ganz besonders gefordert, in Abstimmung mit Entwicklern, Admi-

nistratoren und Management die richtige Balance aus Vor- und Nachteilen zu finden.

Da Cloud Computing im Grundsatz lediglich eine technisch und betriebswirtschaftlich besonders realisierte Bereitstellung von Rechenzeit darstellt, gelten für die Nutzung von Hudson weiterhin dieselben Bedingungen und Möglichkeiten wie für Rechner im Server-Rack Ihres eigenen Rechenzentrums. Lediglich das Starten und Stoppen der Rechner im Rahmen eines Builds bedarf besonderer Schritte. Diese werden Ihnen durch das Plugin »Hudson Amazon EC2« glücklicherweise komplett abgenommen. Das Plugin fügt der systemweiten Konfiguration unter *Hudson verwalten* → *System konfigurieren* einen neuen Abschnitt *Cloud* hinzu. Abbildung 8–27 zeigt eine beispielhafte Konfiguration.

Plugin

»Hudson Amazon EC2«

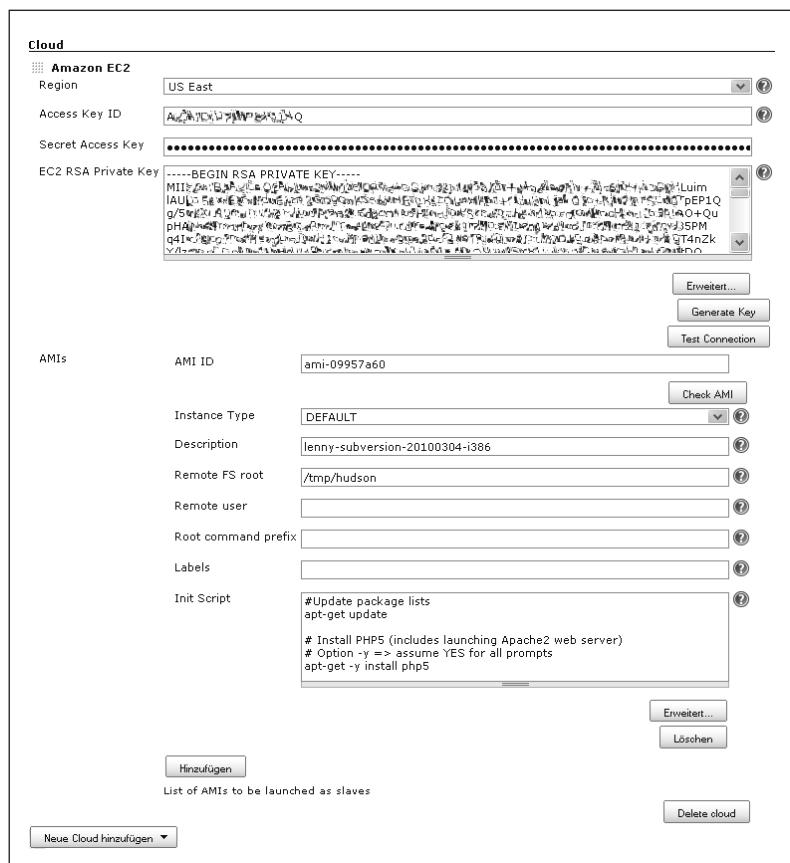


Abb. 8–27

Konfiguration des Plugin
»Hudson Amazon EC2«

Unter *Region* wählen Sie das Amazon-Rechenzentrum, in dem Ihre Maschinen gestartet werden sollen. Amazon bietet hier geografisch

verteilt mehrere Standorte an. Über die *Access Key ID* und den *Secret Access Key* legen Sie das Amazon-Kundenkonto fest, dem letztendlich auch die genutzte Rechenzeit in Rechnung gestellt wird. Der *EC2 RSA Private Key* wird benötigt, um auf den gestarteten Instanzen Kommandos auszuführen. Diese Angaben können Sie im AWS-Kundenbereich entnehmen bzw. dort selbst festlegen.

Unterhalb der allgemeinen Einstellungen fügen Sie alle Amazon Machine Images (AMI) hinzu, die Sie vom Hudson-Master-Knoten als Slave-Knoten starten lassen möchten. Eine geeignete AMI finden Sie über die Suchfunktionen der AWS-Management-Konsole. Im abgebildeten Beispiel wird ein Debian-AMI mit vorinstalliertem Subversion-Client verwendet. Benötigen Sie weitere Softwarepakete, die nicht im Standardumfang eines AMIs enthalten sind, können Sie diese mittels eines *Init Scripts* beim Hochfahren der EC2-Instanz automatisch installieren lassen. Im gezeigten Beispiel wird so ein Apache2-Webserver mit PHP5-Unterstützung installiert. Da manche AMIs kein vorinstalliertes Java mitbringen, kann das Initialisierungsskript praktischerweise auch zur Einrichtung eines Java Runtime Environments bzw. Java Development Kits dienen. Wie bereits erwähnt, starten neue Instanzen stets auf einer Kopie eines AMIs. Nach dem Herunterfahren der Instanz werden alle Änderungen auf dieser Kopie verworfen. Daher muss das Initialisierungsskript bei jedem Hochfahren einer Instanz die notwendigen Nachinstallationen wiederholen. Da Amazons Rechenzentren über eine sehr gute Internetanbindung verfügen, benötigen Installationen aus gängigen Internetquellen jedoch nur wenige Sekunden oder Minuten. Benötigen Sie jedoch exotischere Anpassungen oder eine umfangreiche Anzahl an zusätzlichen Paketen, sollten Sie über ein privates, maßgeschneidertes AMI nachdenken, das Sie auf S3 hinterlegen.

Verwenden angelegter AMIs

Die hier angelegten AMIs dienen als Schablone für Slave-Knoten, die Sie aus Hudson heraus starten können. Dieser Vorgang wird Provisionierung genannt und erfolgt aus der Knoten-Übersichtsseite von Hudson: Ein Ausklappmenü am Ende der Liste der angelegten Knoten erlaubt das automatische Hinzufügen und Starten eines neuen Slave-Knotens in der Amazon EC2-Cloud (Abb. 8–28, oben). Hat Hudson für diesen neu angelegten Knoten Arbeit (z.B. weil alle anderen Knoten belegt sind oder ein Job fest diesem Slave-Knoten zugeordnet wurde), startet Hudson eine EC2-Instanz im Amazon-Rechenzentrum, und der Gebührenzähler beginnt zu laufen ... Nach dem Startkommando dauert es ca. 1–2 Minuten, dann steht der neue Knoten für Builds zur Verfügung (Abb. 8–28, unten). Wird der Slave-Knoten für länger als 30 Minuten nicht mehr benötigt oder manuell gelöscht, beendet Hudson die Instanz im Amazon-Rechenzentrum, und der Gebührenzähler wird wieder gestoppt.

Abb. 8-28

Provisionierung neuer
Slave-Knoten in der
EC2-Cloud

Das Thema Cloud Computing hat in den letzten Monaten äußerst starkes Interesse erfahren. Aus der persönlichen Erfahrung des Autors lohnt sich dessen Einsatz aber längst nicht in allen Anwendungsfällen. Anhand folgender Vor- und Nachteile können Sie ermitteln, ob die Verbindung von Continuous Integration und Cloud Computing in Ihrem Falle sinnvoll ist. Die wichtigsten Argumente für »CI in der Wolke«:

*Ci in der Wolke:
Hit oder Hype?*

■ *Schnelle Skalierbarkeit:*

Zusätzliche Rechner können in Minutenschnelle bereitgestellt sein. Stehen kurzfristig viele Builds an, sind zusätzliche Ressourcen ohne langen Beschaffungsvorlauf verfügbar.

■ *Einfache Bereitstellung identischer Hudson-Knoten:*

Von einem Image können mehrere Knoten gestartet werden. Diese sind (bis auf IP-Adresse und Rechnernamen) völlig identisch konfiguriert.

■ *Rechenleistung nach Bedarf:*

Sie müssen nur für tatsächlich benötigte Rechenzeit bezahlen und keinen Rechnerpark auf dem Niveau einer möglichen Spitzenlast vorhalten.

Dem gegenüber stehen jedoch die Nachteile von »CI in der Wolke«:

■ *Ihr Build-Prozess muss »elastisch« sein:*

Cloud Computing erlaubt zwar das Zuschalten von weiteren Rechnern in Minutenschnelle – wenn sich Ihr Build-Prozess aber nicht automatisch auf mehrere Rechner verteilen lässt, werden Sie keinen Geschwindigkeitsvorteil erzielen.

■ *Höhere Kosten im Dauerbetrieb:*

Obwohl die Stundenpreise pro Rechner sehr preiswert erscheinen (weit unter einem Euro), können Cloud-Computing-Instanzen bei durchgängiger Nutzung deutlich teurer sein als dauerhaft angemietete oder sogar gekaufte Server.

■ *Erhöhter Netzverkehr:*

Sofern Sie nicht Ihre komplette Entwicklungsinfrastruktur (Versionsmanagement, CI-Server, Test-Server, Repositories usw.) in die Wolke auslagern, kann ein nicht unbeträchtlicher Datenverkehr zwischen Ihrem Standort und dem Rechenzentrum des Cloud-Computing-Anbieters entstehen. Dadurch entstehen Ihnen nicht nur weitere Kosten. Auch die Build-Dauer wird – je nach verfügbarer Bandbreite – negativ beeinflusst.

■ *Sicherheitsfragen:*

Als EC2-Anwender sind Sie für die Absicherung Ihrer Instanzen gegen missbräuchliche Nutzung und Ausspähung verantwortlich. Für viele Firmen beendet die Vorstellung, die eigenen Quelltexte – also die »Kronjuwelen« des Unternehmens – übers Internet in ein fremdes Rechenzentrum übertragen zu müssen, die weiteren Bemühungen hinsichtlich »CI in der Wolke«. Auch Amazon hat dies inzwischen erkannt und bietet mit dem Angebot »Virtual Private Cloud (VPC)« gegen Aufpreis die Möglichkeit, EC2-Instanzen isoliert über ein Virtual Private Network (VPN) anzubinden.

Abschließend sei empfohlen, genau zwischen den Vorteilen zu unterscheiden, die sich

- einerseits aus dem *technologischen* Konzept der Virtualisierung,
- andererseits aus dem *betriebswirtschaftlichen* Konzept der bedarfsabhängigen Anmietung von Rechenzeit ergeben.

Viele Teams finden ihre »goldene Mitte« in der Verwendung virtueller Maschinen, betreiben diese aber auf Servern des eigenen, unternehmensweiten Rechenzentrums.

8.6 Hudson absichern

Je größer ein Team oder Unternehmen wird, desto mehr gewinnt die Absicherung der IT-Systeme gegen fahrlässigen, mutwilligen oder gar kriminellen Missbrauch an Bedeutung. Bei einem CI-System ist dies nicht anders.

So könnte Hudson im Intranet einer kleinen Webagentur vielleicht noch ohne weitere Schutzmaßnahmen betrieben werden – hier darf

jeder alles, und wenn etwas schiefläuft, lässt man sich vom Kollegen auf der anderen Seite des Schreibtisches aus der Patsche helfen. Ganz anders liegt der Fall in einem großen Konzern: Aufgrund der hohen Anzahl der Mitarbeiter und des steten Austauschs in der Belegschaft sind persönlich zugeordnete Zugriffsrechte unabdingbar. Zudem können organisatorische oder juristische Rahmenbedingungen eine Nachvollziehbarkeit aller Aktionen auf und Änderungen an einem IT-System vorschreiben.

Wenn wir also vom »Absichern eines Hudson-Systems« sprechen, müssen wir uns um drei Dinge kümmern:

*Authentifizierung,
Autorisierung, Auditierung*

■ ***Authentifizierung:***

Wer ist der Benutzer? Kollegin Müller oder Kollege Mayer? Oder ein Drittsystem vom Rechenzentrum nebenan, das automatisiert Daten abfragen möchte?

■ ***Autorisierung:***

Welche Funktionen darf Kollegin Müller im Hudson-System ausführen? Darf Frau Müller das Hudson-Projekt A nur einsehen oder aber auch neue Builds starten?

■ ***Auditierung:***

Wer hat welche Aktionen wann vorgenommen? Wer hat zuletzt Projekt F neu konfiguriert? Wer das Deployment per Hudson am vergangenen Freitag gestartet? Vereinfacht gesprochen, setzt Auditierung also einen »Fahrtenschreiber« Ihrer Hudson-Instanz voraus.

Die Unterteilung in die genannten drei Aspekte spiegelt sich in Unternehmen typischerweise auch organisatorisch wider:

- Die zentrale Unternehmens-IT stellt einen *Authentifizierungsdienst* bereit, etwa als LDAP-Verzeichnis oder Active-Directory-Dienst.
- Rechte und Rollen (*Autorisierung*) hingegen werden auf Fachabteilungs- bzw. Projektebene durch Linienmanager bzw. Projektleiter festgelegt.
- Die *Auditierung* schließlich wird technisch von einem Systemadministrator durchgeführt. Verwendet werden die Daten hingegen von Führungskräften, wenn Ungereimtheiten zu klären sind ...

8.6.1 Authentifizierung

Hudson authentifiziert seine Benutzer auf vier unterschiedliche Weisen (weitere lassen sich über Plugins nachrüsten):

- keine Authentifizierung (anonymer Betrieb)
- über Hudsons eingebaute Benutzerverwaltung

- über die Benutzerverwaltung des Containers
- über externe Benutzerverzeichnisse

Anonymer Betrieb

Im einfachsten Fall werden Benutzer gar nicht unterschieden. Gegenüber Hudson treten sie alle als ein alternder Benutzer in Erscheinung. Dies kann völlig ausreichend sein für kleine Installationen (bis ca. 5 Benutzer), die grundlegend gegen öffentlichen Zugriff abgeschirmt sind – etwa im Intranet eines kleinen Unternehmens.

Hudsons eingebaute Benutzerverwaltung

Wird der Benutzerkreis größer und verlangt eine organisatorische Unterteilung (etwa in Entwickler, Administratoren, Manager usw.) kann Hudsons eingebaute Benutzerverwaltung gute Dienste leisten. Wenn ein Hudson-System im Internet zugänglich sein soll, ist dieser Ansatz die schnellste Möglichkeit, sensible Daten und Funktionen zu schützen. Neue Benutzer werden dabei über *Hudson verwalten* → *Benutzer verwalten* → *Neuen Benutzer anlegen* hinzugefügt (Abb. 8-29). Hinweis: Der Link *Benutzer verwalten* ist nur sichtbar, wenn Sie vorher unter *Hudson verwalten* → *System konfigurieren* die Option *Hudson absichern* aktiviert haben. Verwechseln Sie die Benutzerverwaltung auch nicht mit der ähnlich aussehenden Benutzerübersicht, die Sie von Hudsons Startseite über Link *Benutzer* in der linken Seitenleiste erreichen.

Abb. 8-29
Hudsons interne Benutzerverwaltung

The screenshot shows the 'User Database' section of the Hudson interface. At the top, there's a search bar, a help icon, and a sign-in link. Below that, a navigation menu includes 'Zurück zur Übersicht' and 'Neuen Benutzer anlegen'. The main area is titled 'Users' and contains a list of three users: 'Hubert J. Farnsworth', 'Max Mustermann', and 'Simon Wiest'. Each user entry includes a small profile picture and a delete icon (a crossed-out circle).

Name	
Hubert J. Farnsworth	
Max Mustermann	
Simon Wiest	

At the bottom of the page, there's a footer with the text 'Erzeugung dieser Seite: 24.07.2010 11:32:38' and 'Hudson ver. 1.363'.

Auf der jeweiligen Profilseite des Benutzers (erreichbar durch Klicken auf den Namen) können dann weitere Einstellungen wie etwa Klartextname, Beschreibung, E-Mail-Adresse, Standardansicht usw. vorgenommen werden. Dort kann ein Benutzer auch wieder gelöscht werden. Für 20–30 Personen ist diese Art der Benutzerverwaltung durchaus praktikabel. Ein Gruppenkonzept fehlt hier allerdings – es können nur individuelle Benutzer angelegt werden.

Wird Hudson innerhalb eines Servlet-Containers wie beispielsweise Tomcat betrieben, können Authentifizierungsanfragen an den Container delegiert werden. Dies ist insbesondere dann praktisch, wenn dort bereits ein Benutzerverzeichnis eingerichtet wurde und Prozesse zu dessen Pflege etabliert sind.

Benutzerverwaltung
des Containers

Zur Authentifizierung in großem Stil kommen dedizierte Systeme zum Einsatz. Am verbreitetsten dürften dazu in der Unix-Welt LDAP-Server bzw. in der Windows-Welt Active-Directory-Server sein. Beide lassen sich an Hudson anbinden, wobei die LDAP-Integration bereits Bestandteil der Hudson-Distribution ist. Active Directory muss hingegen per Plugin nachgerüstet werden. Im Gegensatz zu Hudsons eingebauter Benutzerverwaltung unterstützen die externen Benutzerverzeichnisse auch Benutzergruppen, was in einem späteren Schritt die Pflege von Rechten immens vereinfachen kann.

Externe
Benutzerverzeichnisse

Lightweight Directory Access Protocol (LDAP)

Das Lightweight Directory Access Protocol ist nicht nur in der Unix-Welt eine sehr verbreitete Methode, um Benutzerverzeichnisse aufzubauen. Aufgrund seiner Popularität wird es oftmals auch in Windows-Umgebungen verwendet, um Benutzerinformationen einheitlich und zentral zu verwalten. Hudson unterstützt LDAP daher bereits in der Basisdistribution, also ohne dazu zusätzliche Plugins zu benötigen.

Konfiguration der
LDAP-Anbindung

Sie aktivieren die Authentifizierung über LDAP, in dem Sie zunächst unter *Hudson verwalten* → *System konfigurieren* die Option *Hudson absichern* anwählen. Anschließend wählen Sie im Abschnitt *Zugriffskontrolle* → *Benutzerverzeichnis (Realm)* die Option *LDAP* an. In den meisten Fällen reichen lediglich drei weitere Angaben aus, um die Konfiguration zu vervollständigen:

- Unter *Server* geben Sie die URL Ihres LDAP-Servers an, inklusive des Protokolls, z.B. `ldaps:// mein.ldap.server`.
- Unter *Manager-DN* geben Sie den Distinguished Name (DN) eines Benutzerkontos an, das auf das LDAP-Verzeichnis lesenden Zugriff hat, z.B. `uid=swiest,ou=HudsonAdmin,dc=swiest,dc=de`.
- Unter *Manager-Password* geben Sie das zugehörige Kennwort zur Manager-DN an.

Anhand dieser drei Angaben weiß Hudson nun, welchen LDAP-Server er zur Authentifizierung befragen kann und unter welchem Benutzerkonto diese Anfrage durchgeführt werden soll. Unterstützt der LDAP-Server »anonymes Binding« (also Abfragen ohne vorherige Authentifizierung des Fragenden), sind sogar die Angaben von Manager-DN und Manager-Passwort optional.

Die Angaben für *Stamm-DN*, *Basis zur Benutzerabfrage*, *Filter zur Gruppenabfrage* und *Basis zur Gruppenabfrage* können Sie in den meisten Fällen leer lassen. Hudson verwendet dann Vorgabewerte, die gängigen Konventionen beim Aufbau von LDAP-Verzeichnissen entsprechen. Gelingt die Authentifizierung über den LDAP-Server mit den Vorgabewerten hingegen nicht, sollten Sie sich Hilfe vom Verwalter Ihres LDAP-Systems einholen. Hier haben sich die weitergehenden Texte der eingebauten Online-Hilfe als hilfreich erwiesen, um dem LDAP-Verwalter zu erläutern, welche Funktion die einzelnen Felder besitzen.

Gruppen in LDAP

Im Gegensatz zu Hudsons eingebautem Benutzerverzeichnis erlaubt LDAP auch die Definition von Gruppen, wie z.B. Hudson-Administratoren, Abteilungen, Projektteams. Dies erleichtert die Zuordnung von Rechten ungemein, da Sie nun innerhalb von Hudson den Zugriff auf die Systemkonfiguration auf die Gruppe »Hudson-Administratoren« beschränken können und dann den genauen Personenkreis dieser Gruppe im LDAP-Server vornehmen. Kommt eine neue Person hinzu, müssen Sie diese lediglich im LDAP-Server dieser Gruppe zuordnen – innerhalb von Hudson sind keine weiteren Konfigurationsschritte notwendig.

Plugin »LDAP Email Plugin«

Abschließend noch ein Hinweis aus der Praxis: In Hudsons öffentlichem Update-Center befindet sich auch das Plugin »LDAP Email«. Dieses Plugin stellt *nicht* Hudsons oben beschriebene LDAP-Anbindung bereit. Es hilft vielmehr in Ausnahmefällen bei der Ermittlung von E-Mail-Adressen für gegebene Benutzernamen, wenn diese im LDAP-Verzeichnis an unkonventionellen Stellen abgelegt sind. Verwendet Ihr LDAP-Verzeichnis hingegen dazu das übliche Attribut *mail*, ist die LDAP-Unterstützung der Basisdistribution völlig ausreichend.

Microsoft Active Directory

Plugin »Active Directory«

Microsofts Active Directory wird insbesondere in Windows-Umgebungen als Benutzerverzeichnis eingesetzt. Zur Authentifizierung gegenüber einem Active Directory benötigt Hudson das Plugin »Active Directory«.

Die Konfiguration erfolgt analog zu LDAP im vorausgegangenen Abschnitt. Sie müssen hier allerdings nur eine Angabe tätigen: die Windows-Domäne des Active Directory. Voraussetzung ist, dass Hudson auf einem Windows-Rechner läuft, der zu dieser Domäne gehört. Um Hudson auf einem Unix-System laufen zu lassen, das gegen ein Active Directory authentifiziert, wird LDAP empfohlen (ein Active-Directory-Server kann seine Informationen auch über LDAP anbieten).

Ganz ähnlich der LDAP-Anbindung können Benutzer in einem Active Directory auch Gruppen zugeordnet werden. Auch hier erleichtert dies die Pflege von Rechten innerhalb Hudsons enorm.

Gruppen im Active Directory

Andere Benutzerverzeichnisse

Obwohl mit LDAP und Active Directory der Löwenanteil aller installierten Benutzerverzeichnisse abgedeckt sein dürfte, bietet Hudson über Plugins auch weitere Alternativen etwa für Atlassian Crowd oder Jasig Central Authentication Service (CAS).

Um gänzlich exotische Benutzerverzeichnisse anzubinden, können Sie das Plugin »Script Security Realm« einsetzen. Dieses Plugin leitet die Eingabe von Benutzernamen bzw. Passwort als Umgebungsvariablen `U` bzw. `P` an ein Kommandozeilenprogramm weiter, das anhand dieser zwei Informationen beliebige Algorithmen zur Authentifizierung implementieren kann – etwa in einer Textdatei nachschlagen, eine Datenbank befragen, einen Web-Service konsultieren usw. Liefert die Kommandozeilenanwendung einen Rückgabewert (*return code*) von 0, gilt die Authentifizierung als erfolgreich. In allen anderen Fällen wird die Ausgabe des Kommandozeilenprogramms als Fehlermeldung angezeigt, und die Authentifizierung gilt als fehlgeschlagen.

Plugin »Script Security Realm«

8.6.2 Autorisierung

Nach erfolgreicher Authentifizierung legt die Autorisierung fest, welche konkreten Operationen ein angemeldeter Benutzer ausführen darf. Hudson kennt hier vier eingebaute Verfahren (weitere lassen sich über Plugins nachrüsten):

■ *Jeder darf alle Aktionen ausführen:*

Dies ist der Vorgabewert und schränkt die Rechte eines Benutzers in keiner Weise an.

■ *Angemeldete Benutzer dürfen alle Aktionen ausführen:*

In diesem Fall muss eine erfolgreiche Authentifizierung vorausgegangen sein, um dann ohne weitere Einschränkungen arbeiten zu können. Diese Einstellung stellt einen Minimalschutz bereit, da sie anonyme Benutzer ausschließt.

■ *Matrix-basierte Sicherheit:*

Anhand einer Rechtetabelle (Matrix) können Benutzern und Gruppen beliebige Kombinationen von momentan 16 unterschiedlichen, vorgegebenen Rechten zugeordnet werden. Solche Rechte sind beispielsweise *Anlegen von Projekten, Starten von Builds, Ändern der Systemkonfiguration, Zugriff auf den Arbeitsbereich von Projek-*

ten usw. Erteilte Rechte gelten dabei immer für alle Projekte. Hat ein Benutzer also das Recht *Projekt löschen*, so darf er alle Projekte ohne Ausnahme löschen.

■ *Projektbasierte Matrix-Zugriffssteuerung:*

Dieses Vorgehen stellt insofern eine Erweiterung der Matrix-basierten Sicherheit dar, da sich hier einem Benutzer oder einer Gruppe nicht nur global Rechte erteilen lassen (in der Systemkonfiguration), sondern auch auf Projektebene (in der Projektkonfiguration). Dadurch sind Fälle abbildbar wie »Kollege Maier soll in Projekt A alle Rechte haben, in Projekt B nur neue Builds starten können und auf Projekt C keinen Zugriff haben«. Globale und projektspezifische Rechte addieren sich dabei. Haben Sie also einem Benutzer oder einer Gruppe ein Recht auf globaler Ebene erteilt, können Sie dieses Recht nicht auf Projektebene entziehen. Fälle wie »Kollege Schulze hat auf alle Projekte Zugriff *außer* auf Projekt D« lassen sich also nicht auf einfache Weise konfigurieren. Stattdessen müssen Schulze explizit auf Projektebene für A, B, C, E, F usw. Rechte eingeräumt, Projekt D hingegen ausgespart werden.

Tipp: Falls Sie sich mal aussperren sollten...

Gerade bei den ersten Experimenten zur Absicherung eines Hudson-Servers kommt es durchaus vor, dass man sich selbst unabsichtlich aus dem System aussperrt. Wie gelangen Sie aber wieder ins System, um den Konfigurationsfehler zu beheben?

Dazu deaktivieren Sie übergangsweise Hudsons Zugriffsschutz, indem Sie zunächst Hudson herunterfahren, dann in der Datei <HUDSON_HOME>/config.xml das Element <authorizationStrategy> entfernen und schließlich Hudson neu starten.

Hudson ist dann wieder für jedermann zugänglich, so dass Sie möglichst rasch Ihre gewünschte Absicherung in der Systemkonfiguration vornehmen sollten.

8.6.3 Auditierung

Hudson bietet unter *Hudson verwalten* → *Systemprotokolle* Einblicke in sein internes Logging-System an. Dort werden Ereignisse wie beispielsweise das Laden von Plugins oder die Kommunikation mit Slave-Knoten und Versionsmanagementsystemen protokolliert. Hierbei handelt es sich aber um Aufzeichnungen sehr technischer Natur, die hauptsächlich für Systemadministratoren und Hudson-Plugin-Entwickler gedacht sind.

In diesem Abschnitt verstehen wir unter »Auditierung« hingegen vielmehr die Protokollierung von *fachlichen* Ereignissen eines CI-Sys-

tems aus der Sicht eines Build- und Releasemanagers. Für ihn sind Fragen relevant wie »Wann wurde Projekt A zuletzt gebaut?«, »Wurde die Hudson-Konfiguration in der letzten Woche verändert?« oder »Wer hat wann zuletzt ein Deployment auf die Produktionsmaschinen angestoßen?«.

Hierzu kann das Plugin »Audit Trail« zum Einsatz kommen. Es protokolliert chronologisch Anfragen an Hudson in eine Textdatei. Einen Ausschnitt einer solchen Datei sehen Sie in Listing 8–3.

```
Jul 24, 2010 10:40:57 job/A/ #20 Gestartet durch Benutzer
anonymous
Jul 24, 2010 10:42:50 /job/B/configSubmit by swiest
Jul 24, 2010 10:44:26 /configSubmit by swiest
[...]
```

Plugin »Audit Trail«

Listing 8–3

*Ausschnitt aus Logdatei
des Plugins »Audit Trail«*

Wie ist diese Datei zu lesen? Im Kern protokolliert das Plugin aufgerufene URLs – zusammen mit Zeitpunkt und Namen des Benutzers. In Zeile 1 wurde also Job A durch einen nicht angemeldeten Benutzer (anonymous) gestartet. In Zeile 2 wurde die Konfiguration des Projekts B von Benutzer swiest geändert. In Zeile 3 wurde die Hudson-Serverkonfiguration durch swiest geändert. Beachten Sie, dass Sie dem Protokoll zwar entnehmen können, dass eine neue Konfiguration an den Server geschickt wurde, aber nicht deren Inhalt. Die Frage »Was hat swiest denn genau geändert?« kann also im Nachhinein nicht mehr beantwortet werden.

Audit Trail	
Log Location	<input type="text" value="d:\hudson\audit-trail.log"/>
Log File Size MB	<input type="text" value="5"/>
Log File Count	<input type="text" value="1"/>
URL Patterns to Log	<input type="text" value=".*/(?:configSubmit doDelete postBuildResult cancelQueue stop toggleLogKeep do)"/>
Log how each build is triggered	<input checked="" type="checkbox"/>

Abb. 8–30

*Konfiguration des Plugins
»Audit Trail«*

Das Plugin wird unter *Hudson verwalten → Systemeinstellungen* konfiguriert (Abb. 8–30). Dabei lässt sich zum einen angeben, in welche Datei protokolliert werden soll – auf Wunsch auch mit einem Größenlimit und Angaben zur Log-Rotation. Zum anderen können die zu protokollierenden Ereignisse in Form eines Filters angegeben werden (Feld *URL Patterns to Log*). Dieser Filter wird als regulärer Ausdruck auf alle eingehenden URL-Anfragen angewendet. Nur wenn sich eine Übereinstimmung ergibt, wird die Anfrage im Audit-Trail-Protokoll vermerkt.

8.7 Weitere nützliche Plugins

In diesem Abschnitt betrachten wir eine Reihe von Plugins, die sich zwar keinem der bisher behandelten Themen eindeutig zuweisen lassen, sich aber nichtsdestotrotz in der Praxis als hilfreich erwiesen haben. Die Auswahl aus den über 200 inzwischen verfügbaren Plugins stellt eine sehr subjektive dar, basiert aber auf Erfahrungen aus dem Aufbau zahlreicher Hudson-Installationen.

8.7.1 Claim

Wer ist der Kümmerer?

Plugin »Claim«

Stellen Sie sich vor, der Build eines Projektes schlägt von Zeit zu Zeit fehl. Wer kümmert sich dann um die Angelegenheit? In kleinen Abteilungen kann man dies auf Zuruf regeln. Sind die Kollegen jedoch räumlich getrennt, wird dies schon schwieriger...

Hier hilft das Plugin »Claim«, das es Ihnen erlaubt, einen fehlgeschlagenen Build mit einem »Ich kümmere mich darum«-Symbol zu markieren. Dadurch wird für alle Beteiligten sichtbar, dass bereits Hilfe auf dem Weg ist und wer genau sich der Angelegenheit angenommen hat.

Sie aktivieren die Claim-Funktion per Projekt in den Post-Build-Aktionen durch Anwählen der Option *Allow broken build claiming*. Schlägt einer der folgenden Builds fehl, können Sie den ganzen Build für sich »in Beschlag nehmen« (*claim*). Sie müssen dazu sinnvollerweise unter Ihrem Benutzerkonto im Hudson-System angemeldet sein. Tipp: Sie können sogar ganz gezielt einzelne fehlgeschlagene Tests übernehmen, wenn Sie vor dem Build unter *Veröffentlichte JUnit-Testergebnisse* die Unteroption *Allow claiming of failed tests* angewählt haben.

Wer gerade an welchen Problemstellen arbeitet, lässt sich zum einen dem Claim-Report entnehmen, den Sie über das Symbol *Claim-Report* an der linken Seitenleiste der Hudson-Startseite erreichen (Abb. 8–31). Zusätzlich bietet das Plugin auch eine neue Spalte »Claim« an, die Sie benutzerdefinierten Projektlistenansichten hinzufügen können.

Abb. 8–31

Claim-Report: Wer arbeitet gerade an was?

Build	Date	Failure Duration	Status	Description
greetr #10	8 Minuten 8 Sekunden	8 Minuten 8 Sekunden	claimed by swiest	because: Muß Datenbankserver neu starten.
svninfo #6	2 Minuten 10 Sekunden	3 Minuten 24 Sekunden	claimed by swiest	because: Subversion war offline.

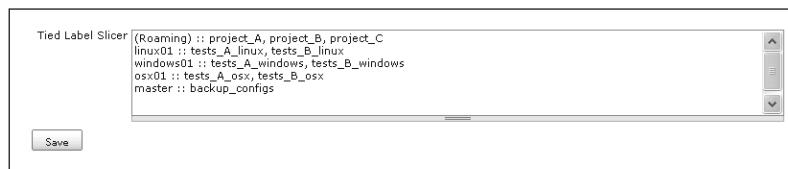
Symbol: S M L

Legende: Alle Builds Nur Fehlschläge Nur jeweils letzte Builds

8.7.2 Configuration Slicing

Die Verwaltung von Dutzenden von Projekten kann in Hudson mitunter arbeitsintensiv sein: Angenommen Sie benötigen einen Überblick, welche Ihrer Projekte welchen Knoten zur Ausführung fest zugeordnet sind. Sie könnten nun natürlich Projektkonfiguration für Projektkonfiguration durchklicken und sich auf einem Blatt Papier Notizen machen.

Oder Sie verwenden das Plugin »Configuration Slicing«. Es erlaubt Ihnen, im »Querschnitt« (*slicing*) durch alle Projektkonfigurationen die Einstellungen für eine bestimmte Option anzuzeigen – und diese sogar zu verändern. Momentan werden ca. 10 Optionen unterstützt, z.B. die Aktivierung zeitbasierter Build-Auslöser oder eben die oben angesprochene Zuordnung zu Knoten. Sie erreichen die Slicing-Funktionen über *Hudson verwalten* → *Configuration Slicing*. Dort wählen Sie die gewünschte Konfigurationsoption aus. Abbildung 8–32 zeigt den *Tied Label Slicer*. Im Textfeld wird Ihnen die Zuordnung nach Slave-Knoten angezeigt. Sie können dieses Textfeld editieren und mit *Save* wieder in die jeweiligen Projektkonfigurationen zurückschreiben lassen – ganz ohne sich mühsam von Projekt zu Projekt klicken zu müssen.



Projekte neu konfigurieren
– Sieben auf einen Streich!

Plugin
»Configuration Slicing«

Abb. 8–32
Schnelle Zuordnung von
Projekten zu Knoten mit
dem Plugin
»Configuration Slicing«

Leider ist das Plugin »Configuration Slicing« momentan auf nur ca. 10 Optionen begrenzt. Wer hingegen auch ein bisschen die Scripting-Sprache Groovy spricht, kann sich mit kleinen Codeschnipseln ganz universell Übersichten über alle Projekte erstellen lassen – und bei Bedarf auch Konfigurationsoptionen in allen Projekten auf einen Rutsch aktualisieren. Sie können auf dem Hudson-Master über die Funktion *Hudson verwalten* → *Skript-Konsole* eigene Groovy-Skripte ausführen. Diese haben vollen Zugriff auf Hudsons internes Datenmodell, also insbesondere auch auf alle Projekte mit ihren Builds, auf angelegte Slave-Knoten, installierte Plugins, die Benutzerverwaltung usw.

Als Anregung finden Sie in Listing 8–4 ein schönes Beispiel, das von Ant-Committer Jan Matérne für das Hudson-System der Apache Foundation entwickelt wurde: Das Skript untersucht alle aktiven Projekte darauf, ob das Build-Timeout-Plugin für das jeweilige Projekt bereits konfiguriert wurde, und aktiviert es bei Bedarf. Danach wird

Alternative:
Groovy-Scripting

jeweils eine Timeout-Zeit von 30 Minuten gesetzt. Zum Abschluss wird eine tabellarische Übersicht über den kompletten Vorgang ausgegeben. Das Plugin »Build Timeout« muss dazu natürlich vorher bereits über das Update-Center installiert worden sein.

Listing 8-4

Einheitliches Setzen einer Build-Timeout-Zeit für alle aktiven Projekte mit Groovy

```

hudsonInstance = hudson.model.Hudson.instance
allItems = hudsonInstance.items
activeJobs = allItems.findAll{job -> job.isBuildable()}
failTheBuild = true
timeoutMinutes = 30

println "Name | Old | New | Mode"
activeJobs.each { job ->
    // Get the timeout plug in
    wrapper = job.getBuildWrappersList().findAll{it instanceof
        hudson.plugins.build_timeout.BuildTimeoutWrapper }[0]

    // Get the current timeout, if any
    currentTimeout = (wrapper != null) ? wrapper.timeoutMinutes : ""

    // Update the timeout, maybe requires instantiation of wrapper
    action = (wrapper != null) ? "updated" : "added timeout"
    if (wrapper == null) {
        plugin = new hudson.plugins.build_timeout.BuildTimeoutWrapper
            (timeoutMinutes, failTheBuild)
        job.getBuildWrappersList().add(plugin)
    } else {
        wrapper.timeoutMinutes = timeoutMinutes
    }

    // String preparation for table output
    String jobname = job.name.padRight(30)
    String oldTimeoutStr = currentTimeout.toString().padLeft(5)
    String newTimeoutStr = timeoutMinutes.toString().padLeft(5)

    // Tabular output
    println "$jobname | $oldTimeoutStr | $newTimeoutStr | $action "
}
"OK"

```

Die Ausgabe des Skripts bei der Ausführung in der Groovy-Skript-Konsole ist in Listing 8-5 gezeigt.

Listing 8-5

Ausgabe des Groovy-Skripts

Name	Old	New	Mode
project-A	30	30	updated
project-B		30	added timeout
C	10	30	updated
project-D	120	30	updated
Result:	OK		

Plugin »Scriptler«

Ein vielversprechender Neuzugang ist in diesem Zusammenhang das Plugin »Scriptler«, das zum einen die Ausführung von Groovy-Skript-

ten auf allen verfügbaren Knoten erlaubt – also nicht nur auf dem Master-Knoten. Zum anderen ermöglicht es sehr einfach den Austausch von erprobten Skripten unter Hudson-Anwendern.

8.7.3 Build-Promotion

In einem gut eingerichteten CI-Server ist es keine Seltenheit, dass alle paar Minuten neue Builds eines Projektes fertiggestellt werden. Was hier den Softwareentwickler mit schneller Rückmeldung erfreut, kann in der Testabteilung Kopfschmerzen verursachen: Sollen wirklich alle Builds anschließend ein aufwendiges manuelles Testverfahren durchlaufen? Ist es noch sinnvoll, mehrere Stunden einen bestimmten Build #100 zu testen, wenn inzwischen bereits Build #101, #102, #103 und #104 desselben Projektes abgeschlossen wurden?

Um nachgelagerte, langsamere Stufen Ihres Softwareentwicklungsprozesses von der »hohen Drehzahl« des CI-Systems zu entkoppeln, können Sie mit dem Plugin »Promoted Builds (Simple)« manuell einzelne Builds *promoten*, also besonders hervorheben. Diese Builds werden zum einen visuell besonders auffällig dargestellt, zum anderen werden sie unbefristet aufbewahrt. Die nachgelagerte Testabteilung (*quality assurance*, kurz: QA) kann sich also wieder entspannen: Sie muss nun nicht mehr jedem neuen Build nachjagen, sondern nur noch ausgewählte, mit *QA-Build* gekennzeichnete Builds prüfen. Ist diese Prüfung erfolgreich, kann ein Tester den Build weiter befördern, z.B. als *QA-Approved* (»Build ist qualitätsgesichert«). Auf diese Weise lassen sich einfache Freigabeprozesse einführen, in denen ein Build Stufe um Stufe seinen Weg bis zur Freigabe durchläuft.

Sie legen mögliche Promotionsstufen unter *Hudson verwalten* → *System konfigurieren* fest. Für jede Stufe lassen sich eine Bezeichnung und ein kleines Symbol (16×16 Pixel) angeben (Abbildung 8–33).

Plugin »Promoted Builds (Simple)«

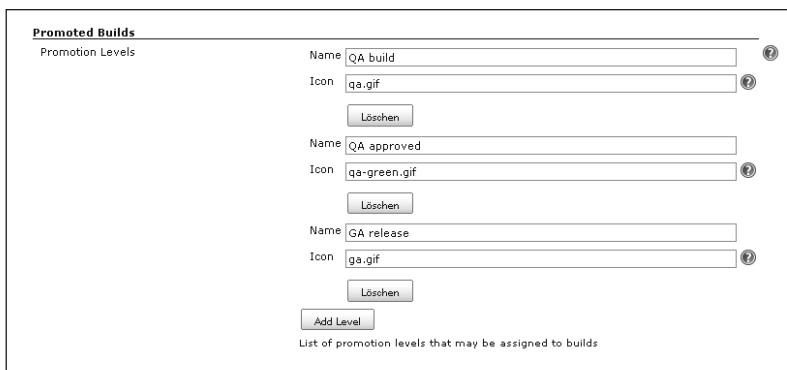
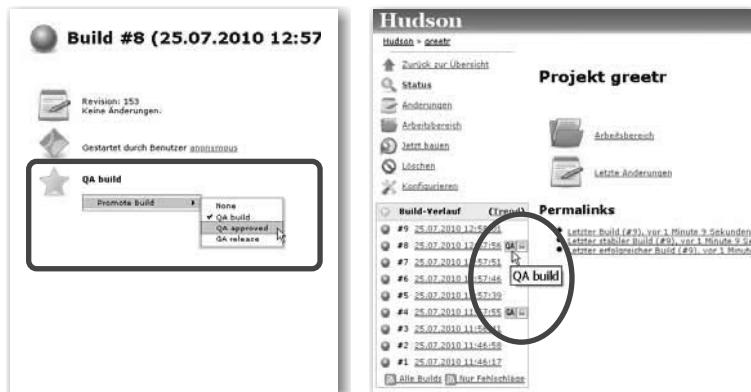


Abb. 8–33
Definition der
Promotionsstufen in der
systemweiten
Konfiguration

Auf der Übersichtsseite eines Builds können Sie nach Abschluss des Builds über das Auswahlfeld *Promote Build* die gewünschte Stufe zuordnen. Auf der Übersichtsseite eines Projektes werden die zugewiesenen Stufen visuell dargestellt (Abb. 8–34).

Abb. 8–34

Befördern eines Builds
(links) und dessen
Darstellung in der
Projektübersicht (rechts)

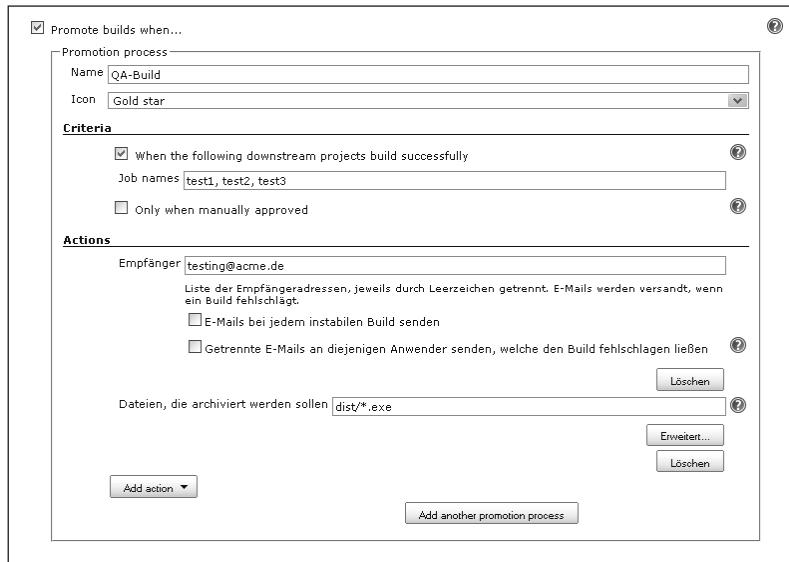


Das Plugin »Promoted Builds (Simple)« verlangt stets die *manuelle* Festlegung der Promotionsstufe. Es handelt sich hier also nicht um eine automatisch ausgeführte Aktivität, sondern vielmehr um ein Kommunikationsmittel zwischen den Benutzern einer Hudson-Installation. Der Promotionsstatus eines Builds ist über das Remote-API von außen abfragbar, so dass die Integration mit Drittsystemen denkbar wäre (z.B. automatisches Anlegen von Testaufträgen oder Abholen neuer beförderter Programmversionen).

Plugin »Hudson Promoted Builds«

Einen Schritt weiter geht »der große Bruder« von »Promote Builds (Simple)« – das Plugin »Hudson Promoted Builds« (Abb. 8–35). Es erlaubt, die Promotion an Bedingungen zu knüpfen, die sich von Hudson selbst überprüfen lassen, und macht somit den Weg frei für vollautomatisierte Promotionen. So lassen sich etwa Builds, bei denen alle Tests in nachgelagerten Projekten erfolgreich waren, befördern. Manuelle Promotionen sind aber auch weiterhin möglich.

Des Weiteren kann das Plugin bei erfolgreicher Promotion zusätzliche Aktionen auslösen, etwa Benachrichtigungen verschicken oder Build-Artefakte für die Testabteilung archivieren.

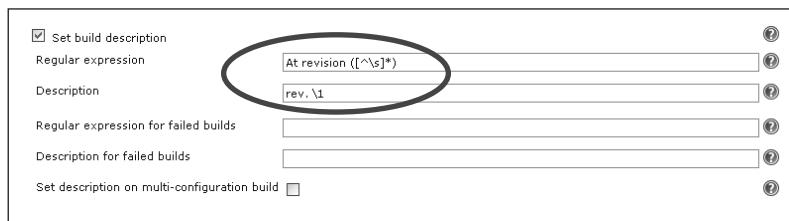
**Abb. 8–35**

Plugin »Hudson Promoted Builds«: Automatisierte Promotion mit verknüpften Aktivitäten

8.7.4 Description Setter

Angenommen, Sie möchten für jeden Build eine charakteristische Eigenschaft direkt auf der Übersichtsseite eines Projektes anzeigen lassen. Das könnte zum Beispiel die verwendete Subversion-Revision sein, die erstellte Maven-Version, der Rechner, auf dem der Build gebaut wurde oder der Benutzer, der den Build ausgelöst hat, usw.

Das Plugin »Description Setter« macht hierfür intelligenten Gebrauch von Hudsons Möglichkeit, jeden Build mit einer Beschreibung zu versehen. Zunächst aktivieren Sie das Plugin pro Projekt durch Anwählen der Option *Set build description* in den Post-Build-Aktionen der Projektkonfiguration. Dann definieren Sie einen regulären Suchausdruck und eine Schablone für die anzuzeigende Beschreibung (Abb. 8–36)

**Abb. 8–36**

Konfiguration des Plugins »Description Setters« pro Projekt

Dadurch durchsucht das Plugin nach jedem Build die Konsolenausgabe mit dem regulären Ausdruck nach bestimmten Fundstellen, z.B. nach einer Zeile mit dem Aufbau »At revision 1234«, wie sie am Ende eines Subversion-Checkouts ausgegeben wird. Einen Teil der ersten Fundstelle – in unserem Beispiel die Revisionsnummer 1234 – setzt das Plugin dann als Beschreibung des Builds. Da Beschreibungen von Hudson prominent in der Weboberfläche platziert werden, kann auf diese Weise jegliche Information eingefügt werden (Abb. 8–37) – immer vorausgesetzt, sie ist in der Konsolenausgabe vorhanden und kann über einen regulären Ausdruck erkannt werden.

Abb. 8–37

Anzeige von

Revisionsnummern pro

Build durch das Plugin

»Description Setter« (links:

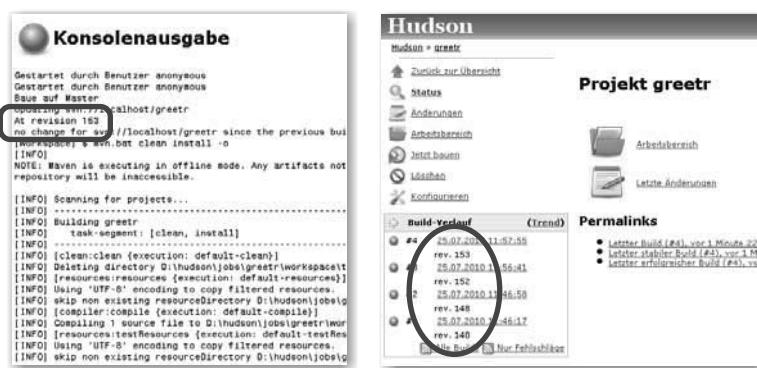
Fundstelle in der

Konsolenausgabe, rechts:

Darstellung der

Revisionsnummer in der

Weboberfläche)



Als zusätzliches Bonbon können Sie übrigens noch einen weiteren regulären Ausdruck angeben, mit dem fehlgeschlagene Builds erkannt werden können, etwa wenn irgendwo in der Konsolenausgabe das Signalwort »[ERROR]« vorkommt.

8.7.5 Green Balls

Hudson verwendet zur Visualisierung der Build-Ergebnisse Kugeln in Blau, Gelb und Rot. Für viele Benutzer ist die Ampeldarstellung (Grün, Gelb, Rot) jedoch eingängiger. Das Plugin »Green Balls« löst dieses Problem, indem alle blauen Kugeln in der Weboberfläche in Grün umgefärbt werden. Nebenbei: Obwohl dieses Plugin keine spektakuläre Erweiterung Hudsons darstellt, gehört es zu den am meisten installierten – ungefähr in Augenhöhe mit dem inzwischen legendären Plugin »Chuck Norris«, das Build-Ergebnisse mit haarsträubenden Superlativen würzt (»There is no ESC key on Chuck Norris' keyboard, because no one escapes Chuck Norris«) ...

8.7.6 Locale

Hudson wurde inzwischen in über 20 Sprachen – mehr oder weniger vollständig – lokalisiert. In welcher Landessprache sich Hudson präsentiert, ist von den Einstellungen des Webbrowsers abhängig. In den meisten Fällen zeigt ein »deutscher« Browser also die deutsche Hudson-Lokalisierung, ein »englischer« Browser die englische usw. In internationalen Unternehmen kann es gewünscht sein, unabhängig vom Browser in allen Fällen dieselbe Lokalisierung – oft die englische – anzuseigen. Das Plugin »Locale« erlaubt es, einen Hudson-Server unabhängig von den Einstellungen der verwendeten Browser auf eine bestimmte Lokalisierung festzulegen.

8.7.7 Continuous Integration Game

Einen besonders launigen Weg, Continuous Integration in einer Arbeitsgruppe einzufügen, stellt das Plugin »Continuous Integration Game« dar: Für jeden Build werden Punkte an alle Benutzer verteilt, die zu diesem Bild mit Änderungen im Versionskontrollsystem beigetragen haben (Abb. 8–38). Für erfolgreiche Builds gibt es jeweils einen Pluspunkt, für fehlgeschlagene hingegen jeweils 10 Minuspunkte. Des Weiteren lassen sich Punkte für neu erstellte oder reparierte Unit-Tests erzielen – aber auch wieder verlieren, wenn in einem Build neue fehlgeschlagende Tests hinzukommen. Ebenso können die Ergebnisse aus statischen Codeanalysen (PMD, Checkstyle, FindBugs usw.) Einfluss finden, wenn die entsprechenden Plugins für diese Analysewerkzeuge ebenfalls installiert wurden.

Plugin »Continuous Integration Game«

Score card	
Rule	Points
1 failing tests were fixed	1.0
1 new failing tests were added	-1.0
4 new checkstyle warningss were found	-4.0
Participating players	
Simon Wiest	

Abb. 8–38

Punkteauswertung eines Builds

Pro Projekt kann entschieden werden, ob Builds dieses Projekts in der Wertung teilnehmen sollen. Die aktuelle »Gesamtbestenliste« lässt sich über das Symbol »Leader board« direkt von Hudsons Startseite aus erreichen. Je nach Klima der Arbeitsgruppe und Betriebsgröße kann das Continuous Integration Game so zum Thema im Biergarten werden ... oder aber beim Betriebsrat.

8.8 Zusammenfassung

In diesem Kapitel haben wir Aufgabenstellungen betrachtet, auf die fortgeschrittene Hudson-Anwender und -Administratoren in ihrem Alltag stoßen – insbesondere bei der Verwaltung umfangreicher Hudson-Installationen. In vielen Fällen trugen intelligent entworfene Plugins entscheidend zur Lösung bei.

Umgekehrt lässt sich also ableiten, dass sich ein regelmäßiger Blick in die Liste der verfügbaren Plugins lohnt. Paradoxe Weise kennt man dadurch nicht selten bereits Lösungen für Anforderungen, die erst später in der eigenen Arbeitsgruppe gestellt werden!

Was aber, wenn sich kein passendes Plugin finden lässt? Wie schwer ist es eigentlich, eine scheinbar »unlösbar« Aufgabe mit einem selbst entwickelten, maßgeschneiderten Plugin zu lösen? Wir widmen uns daher im kommenden Kapitel der Entwicklung eigener Hudson-Plugins.

9 Hudson erweitern

Mit über 200 frei verfügbaren Plugins lässt sich Hudson bereits schon heute an die meisten Umgebungen und Werkzeugketten anpassen. Manchmal sind die eigenen Anforderungen jedoch so speziell (oder die Ideen so originell), dass sich die Entwicklung eines eigenen Plugins lohnt.

Wie Sie feststellen werden, gestaltet sich dies in vielen Fällen viel einfacher als erwartet: Zum einen liegen die eingangs erwähnten 200 Plugins alle im Quelltext vor. Sie können sich also aus einer Fülle an Vorlagen bedienen. Zum anderen stellt das Hudson-Projekt Maven-basierte Werkzeuge bereit, die einem Plugin-Entwickler das Leben bedeutend erleichtern. Wo fangen wir also an?

Zunächst werden wir in einem Schnellstart innerhalb von 60 Sekunden ein vollständiges Hudson-Plugin anlegen, probeinstallieren und zur Distribution verpacken. Nach diesem Sprint werden wir eine effiziente Plugin-Entwicklungsumgebung auf Basis von Eclipse einrichten. Danach verschaffen wir uns einen Überblick über das Hudson-Plugin-Framework, die verwendeten Technologien und die verfügbaren Erweiterungspunkte (*extension points*).

Im letzten Drittel des Kapitels entwickeln wir entlang eines durchgängigen Beispiels »Artifact Size« ein Plugin, das viele typische Plugin-Facetten aufzeigt: Erweiterungen der Benutzeroberfläche, Auswertung von Build-Ergebnissen, Fernsteuerung per Kommandozeile (CLI), XML-Datenaustausch, Internationalisierung und vieles mehr. Das Beispiel kann somit als solider Ausgangspunkt für weitere Entdeckungsreisen in Eigenregie dienen.

Noch eine letzte Anmerkung vorneweg: Dieses Kapitel verwendet auffallend viele englische Begriffe. Dies liegt weniger an der Übersetzungsfaulheit des Autors, sondern soll die Entsprechung zwischen Konzepten und den zugehörigen Java-Klassen nachvollziehbarer machen. Dies erleichtert zudem auch die Suche in Mailinglisten und anderen Informationsquellen des Internets.

Aufbau dieses Kapitels

Denglisch-Alarm?

9.1 Erste Schritte

9.1.1 Schnellstart: Ihr erstes Plugin in 60 Sekunden

Entwicklung mit Maven

Hudson-Plugins entwickeln Sie am besten mit dem Build-Werkzeug Maven. Das Hudson-Projekt stellt hierzu praktische Kommandos bereit, die Ihnen viele Arbeitsschritte beim Erstellen, Verpacken und Testen von Plugins abnehmen. Auch wenn Sie bisher noch wenig Kontakt mit Maven hatten: Keine Bange – es reicht aus, wenn Sie lediglich eine Handvoll Maven-Kommandos beherrschen, die Sie alle im folgenden Abschnitt kennenlernen werden.

Voraussetzungen

JDK, Maven, Internetzugang

Voraussetzung für den Schnellstart ist lediglich ein installiertes Java-Development-Kit 6¹ und eine aktuelle Maven-Version. Das folgende Beispiel wurde mit Maven 2.2.1 erstellt. Zusätzlich sollten Sie Zugang zum Internet haben, um benötigte Bibliotheken während der Ausführung von Maven nachzuladen. Ob alles einsatzbereit ist, können Sie kurz überprüfen mit:

```
$ mvn -version
Apache Maven 2.2.1 (r801777; 2009-08-06 21:16:01+0200)
Java version: 1.6.0_16
Java home: c:\Programme\Java\jdk1.6.0_16\jre
Default locale: de_DE, platform encoding: Cp1252
OS name: "windows xp" version: "5.1" arch: "x86" Family: "windows"
```

Kurzschriftweise für HPI-Tools einrichten

Bevor es richtig losgehen kann, nehmen wir noch eine kleine Maven-Konfiguration vor, die uns später Tipparbeit ersparen wird: Dazu ergänzen wir in der Datei `settings.xml` im Stammverzeichnis Ihres Benutzerkontos einen zusätzlichen `<pluginGroup>`-Eintrag (Windows: `%USERPROFILE%\.m2\settings.xml`, Linux: `~/.m2/settings.xml`):

Listing 9–1

settings.xml – benutzerspezifische Maven-Konfigurationen

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
          http://maven.apache.org/xsd/settings-1.0.0.xsd">
  <pluginGroups>
    <pluginGroup>org.jvnet.hudson.tools</pluginGroup>
  </pluginGroups>
</settings>
```

1. Java 6 ist nur zum Bauen von Plugins erforderlich. Die Plugins selbst dürfen hingegen keine Abhängigkeiten zu Java 6 benötigen, da Hudsons offizielle Laufzeitumgebung immer noch Java 5 ist.

Wenn die Datei `settings.xml` noch nicht existiert, legen Sie sie mit obenstehendem Inhalt neu an. Der zusätzliche Eintrag ermöglicht den Aufruf von Kommandos der Hudson-Plugin-Unterstützung (HPI) in verkürzter Form, statt `mvn org.jvnet.hudson.tools:maven-hpi-plugin:create` müssen Sie also lediglich noch `mvn hpi:create` eingeben.

Das HelloWorld-Plugin

Im Schnellstart werden wir nun ein HelloWorld-Plugin-Projekt anlegen. Das Plugin wird Hudson um einen neuen Builder erweitern. Builder kennen Sie bereits von der Konfigurationsseite eines Free-Style-Projekts: In der Liste *Build-Schritt hinzufügen...* werden verfügbare Builder angezeigt, z.B. *Ant aufrufen* oder *Windows Batch Datei ausführen* (Abb. 9–1 links).

Funktionsumfang
des Plugins

Der neue Builder *Say hello world* ist sehr einfach gestrickt und soll bei seiner Ausführung lediglich einen Gruß in die Konsolenausgabe des Builds schreiben (Abb. 9–1 rechts). Der Name des Gegrüßten soll auf der Jobkonfigurationsseite einstellbar sein (Abb. 9–1, Mitte). Zusätzlich soll in den Systemeinstellungen unter *Hudson verwalten → System konfigurieren* per Checkbox auswählbar sein, ob Grüße statt in Englisch auf Französisch ausgegeben werden sollen (nicht in der Abbildung dargestellt).



Abb. 9–1
Funktionsweise des
HelloWorld-Plugins

Anlegen des HelloWorld-Plugins

Wie der Zufall so spielt, erzeugt das Maven-Kommando `hpi:create` für Sie vollautomatisch die vollständigen Quelltexte eines lauffähigen Hudson-Plugins mit dem oben beschriebenen Funktionsumfang. Sie werden dabei lediglich nach der `groupId` und der `artifactID` des neuen Plugins gefragt. Die `groupId` entspricht dem späteren Java-Package, in dem Ihr Plugin untergebracht sein wird. Planen Sie, Ihr Plugin später im Rahmen des Hudson-Projekts zu veröffentlichen, können Sie als Konvention `hudson.plugins` angeben. Die `artifactID` ist der interne Name des Plugins – in unserem Beispiel `helloworld`. Los geht's:

```
$ mvn hpi:create
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'hpi'.
[INFO] -----
[INFO] Building Maven Default Project
[INFO]   task-segment: [hpi:create] (aggregator-style)
[INFO] -----
[...]
[INFO] Velocitymacro : initialization complete.
[INFO] Velocity successfully started.
[INFO] [hpi:create {execution: default-cli}]
Enter the groupId of your plugin: hudson.plugins
[INFO] Defaulting package to group ID: hudson.plugins
Enter the artifactId of your plugin: helloworld
[...]
[INFO] Archetype created in dir: d:\dev\helloworld
[INFO] -----
[INFO] BUILD SUCCESSFUL
[INFO] -----
[INFO] Total time: 12 seconds
[INFO] Finished at: Mon Apr 12 13:28:50 CEST 2010
[INFO] Final Memory: 11M/19M
[INFO] -----
```

Als Ergebnis haben Sie nun ein neues Verzeichnis **helloworld** auf der Festplatte. Darin finden Sie wiederum eine Maven-POM-Datei sowie ein Verzeichnis **src** mit allen Quelltexten des Plugins.

Testen des HelloWorld-Plugins

Das nächste Kommando erledigt gleich mehrere Aufgaben auf einen Streich: Zunächst wird unser Plugin kompiliert, dann testweise in eine Hudson-Instanz ausgebracht, und schließlich wird diese Instanz zum interaktiven Testen gestartet:

```
$ cd helloworld
$ mvn hpi:run
[INFO] Scanning for projects...
[INFO] -----
[INFO] Building Unnamed hudson.plugins:helloworld:hpi:1.0-SNAPSHOT
[INFO]   task-segment: [hpi:run]
[INFO] -----
[...]
12.04.2010 15:37:47 hudson.TcpSlaveAgentListener <init>
INFO: JNLP slave agent listener started on TCP port 3050
```

Unter <http://localhost:8080> erreichen Sie nun Ihre Hudson-Testinstanz und können sich dort im Plugin-Manager vergewissern, dass Ihr neues Plugin auch installiert wurde (wie in Abb. 9–2). Sie beenden den Testbetrieb mit STRG + C.



Abb. 9-2
Installiertes
HelloWorld-Plugin in
Hudson-Testinstanz

Die erste Ausführung von `hpi:run` kann ein paar Minuten dauern, da Maven eine speziell vorbereitete Hudson-Distribution herunterlädt, die erweiterte Funktionen für Entwickler bereithält. Alle weiteren Starts benötigen dann nur noch wenige Sekunden. Das Arbeitsverzeichnis (`HUDSON_HOME`) der Hudson-Testinstanz finden Sie übrigens unter `helloworld/work`.

Verpacken des HelloWorld-Plugins

Angenommen, Sie haben Ihr neues Plugin ausgiebig ausgebaut und getestet. Um das neue Plugin als installationsfähige HPI-Datei zu verpacken, rufen Sie lediglich `mvn package` auf:

```
$ cd helloworld
$ mvn package
[INFO] Scanning for projects...
[INFO] -----
[INFO] Building Unnamed hudson.plugins:helloworld:hpi:1.0-SNAPSHOT
[INFO]   task-segment: [package]
[INFO] -----
[...]
[INFO] Building jar: d:\dev\helloworld\target\helloworld.hpi
[INFO] Building jar: d:\dev\helloworld\target\helloworld.jar
[INFO] -----
[INFO] BUILD SUCCESSFUL
[INFO] -----
[INFO] Total time: 30 seconds
[INFO] Finished at: Mon Apr 12 15:26:39 CEST 2010
[INFO] Final Memory: 39M/70M
[INFO] -----
$ ls -l target
drwxrwxrwx+ 4 swiest Kein    0 Apr 12 15:26 classes
drwxrwxrwx+ 3 swiest Kein    0 Apr 12 15:26 generated-sources
drwxrwxrwx+ 4 swiest Kein    0 Apr 12 15:26 helloworld
-rwxrwxrwx  1 swiest Kein 8689 Apr 12 15:26 helloworld.hpi
```

```
-rwxrwxrwx 1 swiest Kein 7678 Apr 12 15:26 helloworld.jar  
drwxrwxrwx+ 2 swiest Kein 0 Apr 12 15:26 test-classes
```

Im Unterverzeichnis target befindet sich das installationsfähige Plugin helloworld.hpi im HPI-Format. Sie können diese Datei nun manuell über den Plugin-Manager in ein laufendes Hudson-System hochladen oder an Sie dafür bewundernde Kollegen verteilen.

Gratulation – Sie haben Ihr erstes Hudson-Plugin erstellt!

Säubern des Projektverzeichnisses

Gelegentlich kann es notwendig sein, das Projektverzeichnis von Überresten aus vorherigen Testläufen zu löschen. Dies erreichen Sie mit folgendem Maven-Kommando:

```
$ mvn clean
```

9.1.2 Anatomie eines Plugins

Vermutlich haben Sie sich bereits während des Schnellstarts in Abschnitt 9.1.1 gefragt, welche Quelltextdateien durch hpi:create angelegt wurden. Schauen wir uns dazu ein bisschen im Projektverzeichnis helloworld um:

Das komplette Plugin besteht aus sieben Dateien, deren Verzeichnislayout in Abbildung 9–3 dargestellt ist. Unter diesen Dateien befindet sich lediglich eine einzige Java-Datei, in der also vermutlich die komplette Funktionalität des Plugins enthalten sein wird. Fünf weitere Dateien (.html und .jelly) werden für die Darstellung von Formularfeldern auf den Konfigurationsseiten sowie für die Beschreibung des Plugins im Plugin-Manager benötigt. Die siebte Datei, pom.xml, legt den Maven-Projekttyp und die Abhängigkeiten des Plugins zu anderen Bibliotheken fest.

An dieser Stelle möchten wir uns zunächst einen groben Überblick über die Anatomie eines Plugins verschaffen. Wir betrachten daher im Folgenden nur die Datei `HelloWorldBuilder.java` näher. Das Zusammenspiel mit den anderen Dateien (insbesondere mit den Jelly-Schablonen) lernen wir im Rahmen des ausführlichen Beispiels »Artifact Size« in Abschnitt 9.3 besser kennen.

Die Aufgaben der Quelltextdateien

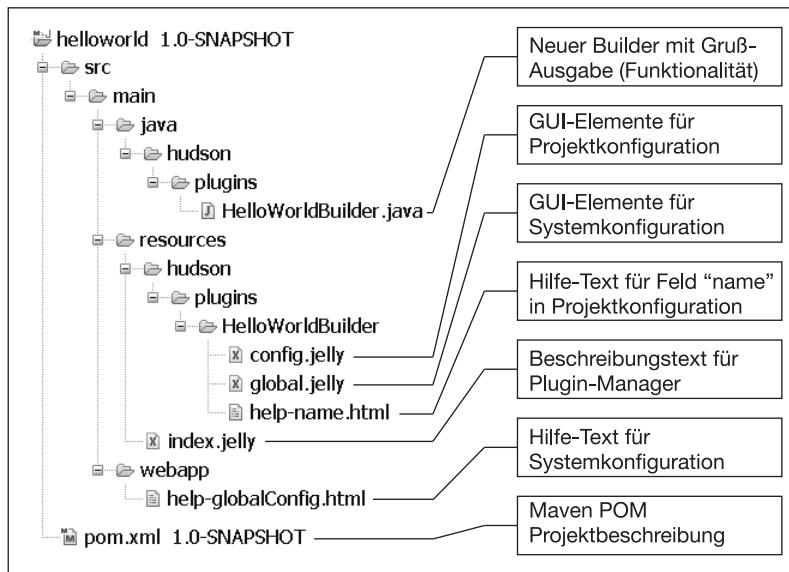


Abb. 9–3
Dateilayout des
HelloWorld-Plugins

Listing 9–2 zeigt die komplette Datei `HelloWorldBuilder.java`. Zur kompakteren Darstellung in diesem Buch wurden lediglich Package-Deklaration, Import-Statements und einige Kommentare gekürzt.

Der HelloWorld-Builder

```

01 public class HelloWorldBuilder extends Builder {
02
03     private final String name;
04
05     @DataBoundConstructor
06     public HelloWorldBuilder(String name) {
07         this.name = name;
08     }
09
10    public String getName() {
11        return name;
12    }
13
14    @Override
15    public boolean perform(AbstractBuild build,
16                           Launcher launcher, BuildListener listener) {
17        if(getDescriptor().useFrench()) {
18            listener.getLogger().println("Bonjour, " + name + "!");
19        } else {
20            listener.getLogger().println("Hello, " + name + "!");
21        }
22        return true;
23    }
24

```

Listing 9–2
HelloWorldBuilder.java

```

25  @Override
26  public DescriptorImpl getDescriptor() {
27      return (DescriptorImpl)super.getDescriptor();
28  }
29
30  @Extension
31  public static final class DescriptorImpl extends
32      BuildStepDescriptor<Builder> {
33
34      private boolean useFrench;
35
36      public FormValidation doCheckName(@QueryParameter
37          String value) throws IOException, ServletException {
38          if(value.length()==0)
39              return FormValidation.error("Please set a name");
40          if(value.length()<4)
41              return FormValidation.warning("Name too short?");
42          return FormValidation.ok();
43      }
44
45      public boolean isApplicable(
46          Class<? extends AbstractProject> aClass) {
47          return true;
48      }
49
50      public String getDisplayName() {
51          return "Say hello world";
52      }
53
54      @Override
55      public boolean configure(StaplerRequest req, JSONObject o)
56          throws FormException {
57          useFrench = o.getBoolean("useFrench");
58          save();
59          return super.configure(req,o);
60      }
61
62      public boolean useFrench() {
63          return useFrench;
64      }
65  }
66 }
```

*Struktur von
HelloWorldBuilder.java*

Verschaffen wir uns zunächst einen Überblick über die Struktur `HelloWorldBuilder.java`, bevor wir den Quelltext Zeile für Zeile betrachten: Zunächst fällt auf, dass die Klasse von `hudson.model.Builder` ableitet (die ihrerseits wiederum von `hudson.model.BuildStep` ableitet). Diese Klasse stellt also einen der Build-Schritte dar, die Sie von der Jobkonfigurationsseite eines Free-Style-Projektes kennen – aber sicherlich haben

Sie bereits vermutet, dass Plugin-Entwicklung etwas mit dem Ableiten bereits vorhandener Basisklassen zu tun hat. Interessanter scheint daher die innere Klasse `DescriptorImpl`, die von `BuildStepDescription<Builder>` ableitet und zudem noch die Annotation `@Extension` trägt.

Wie spielen `HelloWorldBuilder` und `DescriptorImpl` zusammen? Und wie erkennt Hudson dieses Gespann als ein Plugin? Hudson findet Erweiterungen, indem es nach Klassen sucht,

- die mit der Annotation `@Extension` markiert sind
- und ganz bestimmte Basisklassen erweitern – die sogenannten Erweiterungspunkte (*extension points*).

Im vorliegenden Fall trifft dies also *nicht* auf die Klasse `HelloWorldBuilder`, sondern auf `DescriptorImpl` zu.

Deskriptoren dienen der Registrierung eines Plugins in Hudson-Server. Sie erzeugen und konfigurieren außerdem Instanzen des Plugins zur Laufzeit. Sie kennen diese Art der Beziehung vermutlich bereits aus dem Entwurfsmuster »Factory«. Von `HelloWorldBuilder` kann es also viele unterschiedliche Instanzen geben, die jeweils unterschiedliche Personen grüßen und vielleicht sogar in unterschiedlichen Jobs angelegt wurden.

Einen Deskriptor hingegen gibt es nur einmal (Entwurfsmuster »Singleton«). Er beinhaltet nicht nur beschreibende Daten, etwa mit welchem Namen der Builder im Auswahlmenü der Build-Schritte angezeigt werden soll. Er speichert auch globale Parameter, die für alle Instanzen von `HelloWorldBuilder` identisch sind (hier: die Einstellung, ob auf Französisch begrüßt werden soll).

Es verbleibt die Frage, woher Hudson denn weiß, dass `DescriptorImpl` einen `HelloWorldBuilder` beschreibt? Schließlich enthält `DescriptorImpl` keinerlei Hinweis auf `HelloWorldBuilder`, z.B. in einer Methodensignatur oder Ähnlichem. Die Antwort liegt in der Konvention, Deskriptoren als innere Klasse in die zu beschreibende Klasse einzubetten. Die Basisklasse `hudson.model.Descriptor` enthält die notwendige Funktionalität, um eine umgebende Klasse als die zu beschreibende Klasse zu entdecken.

Betrachten wir nun `HelloWorldBuilder.java` Zeile für Zeile. Auch wenn Sie sich am liebsten gleich hinter Ihre IDE schwingen würden, sei Ihnen nachdrücklich angeraten, die folgenden zwei Seiten trotzdem am Stück zu lesen. Keine Sorge: Die investierten Minuten sparen Sie später durch ein besseres Verständnis problemlos wieder ein.

- **Zeile 1:** `HelloWorldBuilder` leitet von `hudson.model.Builder` ab. Dadurch wird das grundlegende Verhalten eines Builders geerbt

Wie erkennt Hudson Plugins?

Deskriptoren sind Factories für Plugins.

Deskriptoren sind Singletons.

Verbindung von Deskriptor und beschriebener Klasse

HelloWorldBuilder.java: Zeile für Zeile

und es müssen nur noch spezifische Felder und Methoden ergänzt werden.

- **Zeile 3:** Der Name des Gegrüßten wird im Feld `name` gespeichert. Beachten Sie, dass das Feld als `final` deklariert ist. `HelloWorldBuilder` wird über seinen Konstruktor vollständig initialisiert und ist danach nicht mehr veränderbar.
- **Zeile 5–8:** Der Konstruktor erzeugt eine vollständig initialisierte Instanz von `HelloWorldBuilder`. Die Annotation `@DataBoundConstructor` ist Teil des Webframeworks Stapler (mehr zu Stapler erfahren Sie in Kapitel 9). Sie sorgt dafür, dass aus den übermittelten Formulardaten der Benutzeroberfläche der Wert des Parameters `name` in den gleichnamigen Konstruktorparameter `name` geschrieben wird. Das Formular ist in der Datei `config.jelly` definiert. Diese Konvention erübriggt eine explizite Abbildung (*mapping*) von Formulardaten und Konstruktorparametern.
- **Zeile 10–12:** Der öffentliche »Getter« `getName` für das Feld `name` wird beim Aufbau der Jobkonfigurationsseite vom Framework Jelly aufgerufen (mehr zu Jelly erfahren Sie ebenfalls in Kapitel 9). Dadurch kann das Eingabefeld für den Namen des Gegrüßten mit dem aktuell eingestellten Wert vorbelegt werden.
- **Zeile 14–23:** Die Methode `perform` enthält die eigentliche Funktionalität des Plugins, also das Grüßen in der Konsolenausgabe. Die Methodenparameter (`build`, `launcher`, `listener`) erlauben Zugriff auf das Objektmodell Hudsons. In unserem Beispiel wird dies genutzt, um über `listener.getLogger().println()` in die Konsolenausgabe zu schreiben. Außerdem können Sie in Zeile 17 sehen, wie innerhalb eines Builders auf globale Konfigurationsdaten des Deskriptors zugegriffen werden kann. Die Methode `perform` liefert einen booleschen Wert zurück: `true` bedeutet, dass der Build weiter fortgesetzt werden kann; `false` hingegen würde zum Abbruch eines Builds führen (etwa wenn ein Fehler während der Ausführung von `perform` aufgetreten wäre).
- **Zeile 25–28:** Die Methode `getDescriptor` liefert eine Instanz eines Deskriptors zurück.
- **Zeile 30–32:** Die statische innere Klasse `DescriptorImpl` leitet von `BuildStepDescriptor` ab. Dadurch kann die Basisfunktionalität eines Deskriptors speziell für Build-Schritte geerbt werden. Es müssen nur noch zwei Methoden – `getDisplayName` und `isApplicable` – implementiert und ggf. eigene Felder und Methoden ergänzt werden. Da alle Instanzen von `HelloWorldBuilder` durch den gleichen Deskriptor beschrieben werden, ist die Klasse `static`. Wie bereits

erwähnt, wird durch die Einbettung als innere Klasse der Bezug zwischen `HelloWorldBuilder` und seinem Deskriptor `DescriptorImpl` hergestellt.

- **Zeile 34:** Im Feld `useFrench` wird gespeichert, ob die Begrüßung in Französisch statt Englisch erfolgen soll.
- **Zeile 36–43:** Die Methode `doCheckName` wird von der Weboberfläche zur Validierung der Benutzereingabe verwendet. Validierungen laufen AJAX-ähnlich asynchron ab. Dadurch können Werte in Formularen der Benutzeroberfläche bereits bei der Eingabe überprüft werden und nicht nur bei der Übermittlung einer ganzen Konfigurationsseite. Beachten Sie, dass auch hier wieder eine Namenskonvention am Werk ist: Die Methode `doCheckXyz` muss exakt zum Formularparameter `xyz` der Weboberfläche passen. Das Ergebnis der Validierung ist nicht etwa nur binärer Natur, sondern erlaubt über die Klasse `FormValidation` differenziertere Rückmeldungen. Diese werden in der Oberfläche entsprechend unterschiedlich visualisiert: Fehler in Rot, Warnungen in Gelb. Tipp: `FormValidation` ist mehr als nur ein passiver Datencontainer. Die Klasse beinhaltet grundlegende Validierungsfunktionen, die Ihnen typische Überprüfungen bereits abnehmen können, z.B. mit `FormValidation.validatePositiveInteger`.
- **Zeile 45–48:** Die Methode `isApplicable` lässt Sie einschränken, für welche Art von Projekten Ihr Builder geeignet ist. So können Sie etwa Free-Style- und Maven-Projekte zulassen, aber Multikonfigurationsprojekte ausschließen. In unserem Beispiel findet keine Einschränkung statt.
- **Zeile 50–52:** Die Methode `getDisplayName` liefert einen Text zurück, der zur Anzeige des Builders in der Benutzeroberfläche verwendet wird. Im Falle eines Builders wird dieser Text im Auswahlmenü *Build-Schritt hinzufügen...* auftauchen. Beachten Sie, dass in unserem Beispiel keine Lokalisierung in Fremdsprachen vorgesehen ist. Wir werden aber in Abschnitt 9.3.7 darauf eingehen, wie Sie mehrsprachige Plugins realisieren können.
- **Zeile 54–60:** Die Methode `configure` wird aufgerufen, wenn eine neue Systemkonfiguration übermittelt wird – also immer dann, wenn Sie auf der systemweiten Konfigurationsseite auf *Übernehmen* klicken. Das Plugin trägt zu dieser Seite den in der Datei `global.jelly` definierten Formularabschnitt bei. Sie müssen an dieser Stelle selbst Sorge dafür tragen, dass die Konfigurationseinstellungen, die Sie im Objekt `o` zurückerhalten, auch dauerhaft auf Festplatte gespeichert werden. Dazu werten Sie zunächst die erhaltenen

Daten aus und übernehmen die benötigten Werte in Ihren Deskriptor (Zeile 57). Hier wären natürlich auch weitere Validierungen und Datentransformationen denkbar. Abschließend rufen Sie `save` auf, um die Datenfelder Ihres Deskriptors dauerhaft zu speichern. Interessanterweise ist dazu keine weitere Konfiguration notwendig. Der Deskriptor geht einfach davon aus, dass alle öffentlichen Felder bzw. Felder mit öffentlichen »Gettern« gespeichert werden sollen. Markieren Sie ein Feld als `transient`, wird es von diesem Mechanismus ausgenommen und wird nicht gespeichert. Abschließend wird mit `super.configure` der Elternklasse Gelegenheit gegeben, auf die Übermittlung von Konfigurationsdaten zu reagieren.

- **Zeile 62–64:** Der »Getter« für das Feld `useFrench` wird beim Aufbau der systemweiten Konfigurationsseite vom Framework Jelly aufgerufen. Dadurch kann die Checkbox mit dem aktuell eingesetzten Wert vorbelegt werden.

Wie geht es weiter?

Dieser zeilenweise Durchlauf von `HelloWorldBuilder.java` beschließt unsere anatomische Betrachtung des `HelloWorld`-Plugins. Wir wissen nun im Groben, wie ein Plugin aufgebaut ist und welche Komponenten dabei zusammenspielen müssen. Im folgenden Abschnitt kümmern wir uns um die Einrichtung einer effizienten Plugin-Entwicklungsumgebung.

9.1.3 Einrichten einer Entwicklungsumgebung

Der Schnellstart in Abschnitt 9.1.1 hatte sein Versprechen gehalten: Über `hpi:create` hatten wir uns ein `HelloWorld`-Plugin erzeugt, mit `hpi:test` getestet und schließlich mit `package` eine installationsfähige HPI-Datei erzeugt. Um aber auch umfangreichere Plugins wirklich effizient zu entwickeln, lassen Sie sich am besten von einer IDE unterstützen. Im Folgenden lernen Sie, wie Ihnen Eclipse bei der Entwicklung von Hudson-Plugins hilft.

settings.xml

Falls noch nicht geschehen, sollten Sie – wie im Schnellstart in Listing 9–1 beschrieben – die Datei `settings.xml` um einen `pluginGroup`-Eintrag erweitern.

Eclipse

Wenn Sie diesen Abschnitt lesen, setzen Sie in den meisten Fällen Eclipse bereits ein. Falls nicht, installieren Sie nun eine aktuelle Version. Für dieses Buch wurde Version 3.5 (Galileo) eingesetzt.

m2eclipse

Da Eclipse von Haus aus keine Maven-Unterstützung mitbringt, aber Hudsons Plugin-Tools Maven voraussetzen, installieren Sie das Eclipse-Plugin `m2eclipse` (<http://m2eclipse.sonatype.org>). Durch `m2eclipse` lassen sich unter anderem Maven-Builds direkt aus Eclipse

starten, was zur Fehlersuche sehr hilfreich ist – etwa weil Sie dann über Breakpoints die Programmausführung an wichtigen Stellen anhalten und inspizieren können.

Glücklicherweise müssen Sie mit Ihrem eigenen Plugin nicht »bei null« beginnen. Sie haben sogar gleich zwei Möglichkeiten:

Erzeugen eines
Plugin-Skeletts

- Möglichkeit 1: Sie erzeugen sich – wie im Schnellstart gezeigt – ein neues Plugin mittels `hpi:create`. Tipp: In der POM-Datei des so erzeugten Plugins wird unterhalb von `<parent>` im Element `<version>` angegeben, mit welcher Hudson-Version das Plugin entwickelt wurde. Ersetzen Sie diese Angabe durch die gerade aktuelle Versionsnummer. Dadurch bauen und testen Sie Ihr Plugin mit `hpi:run` automatisch gegen diese aktuelle Version.
- Möglichkeit 2: Sie kopieren sich den Quelltext eines der über 200 existierenden Plugins. Am einfachsten checken Sie sich dazu einmal die kompletten Quelltexte aller Plugins lokal auf Ihre Festplatte aus. Dieser Schritt ist technisch gesehen nicht zwingend erforderlich. In der Praxis hat es sich aber als sehr nützlich erwiesen, diese Quellen als Nachschlagewerk und zur Inspiration zur Hand zu haben.

Die Quelltexte können Sie mit einem Subversion-Client aus dem öffentlichen Hudson-Repository herunterladen. In der Kommandozeile sieht das so aus:

```
$ svn checkout https://hudson.dev.java.net/svn/hudson/trunk  
/hudson/plugins --username guest --password ""
```

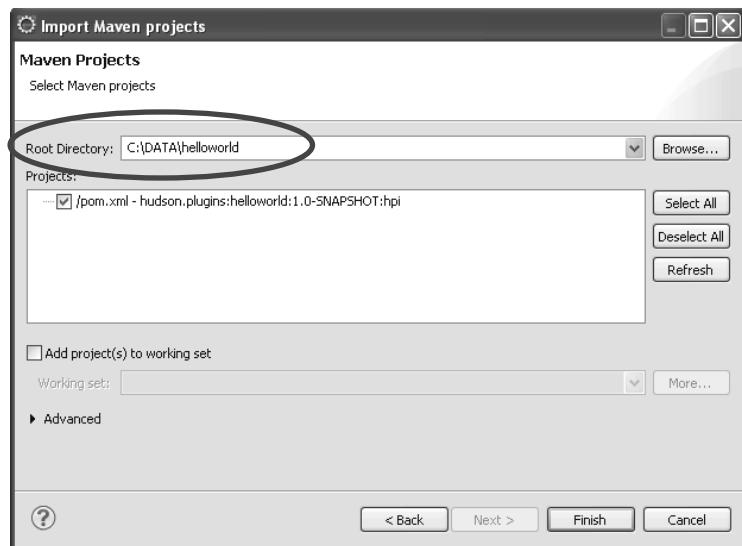
Die Quelltexte der Plugins benötigen ca. 500 MB auf Ihrer Festplatte. Je nach Internetanbindung dauert das erste Auschecken also ein paar Minuten. Leider ist der öffentliche Subversion-Server nicht immer so schnell, wie man sich das wünschen würde. Falls die Übertragung beim Auschecken abbricht, wiederholen Sie das Kommando `checkout` einfach nochmals. Sie halten die ausgecheckten Verzeichnisse im Lauf der Tage und Wochen aktuell, indem Sie in das Verzeichnis `plugins` wechseln und das Subversion-Kommando `update` ausführen:

```
$ cd plugins  
$ svn update --username guest --password ""
```

Nach diesen Vorarbeiten kehren wir in Eclipse zurück und importieren dort Ihr neues Plugin-Projekt. Verwenden Sie dazu aus dem Menü den Befehl *File → Import...* und wählen dann im sich öffnenden Dialog die Import-Quelle *Maven → Existing Maven Projects* aus. Im Dialog *Import Maven Projects* geben Sie unter *Root Directory* das Verzeichnis Ihres Plugin-Projektes an (siehe Abb. 9–4).

Importieren des Plugin-
Projekts in Eclipse

Abb. 9-4
Import des Plugin-Projekts
mit m2eclipse



Eclipse beginnt nun mit dem Import Ihres Projektes und wird typischerweise noch weitere Abhängigkeiten aus Maven-Repositorien nachladen. Es kann also je nach Umfang noch einige Minuten dauern, bis das Plugin-Projekt vollständig in Eclipse startklar ist. Sie starten Ihr Plugin in einer Hudson-Testinstanz, indem Sie Ihr Plugin-Projekt im *Project Explorer* mit rechts anklicken und aus dem Kontextmenü den Befehl *Run As → Maven build..* auswählen. Im Dialog *Edit Configuration* tragen Sie im Feld *Goals* den Wert `clean hpi:run` ein und wählen gegebenenfalls unter *Maven Runtime* die zu verwendende Maven-Version aus (siehe Abb. 9-5).

Mit *Run* wird das Projekt gesäubert, dann neu gebaut und abschließend eine Hudson-Testinstanz mit installiertem Plugin gestartet. Sie können nun in einem Browser unter der URL `http://localhost:8080` die Früchte Ihrer Arbeit bewundern.

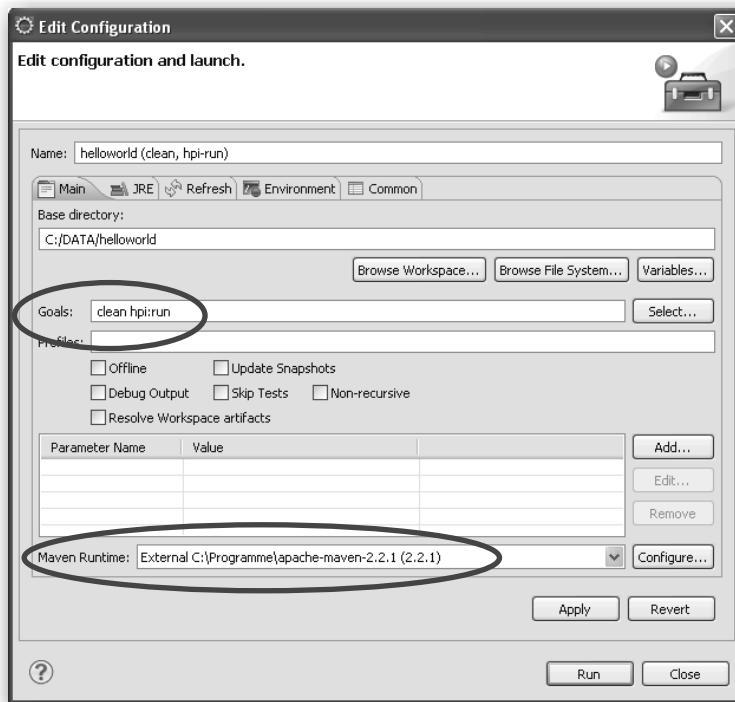


Abb. 9–5
Konfiguration zum
Starten des Plugin-
Projekts mit m2eclipse

Als letzten Handgriff laden wir noch die Quelltexte für das Modul *hudson-core* aus dem Maven-Repository nach. Klicken Sie dazu im *Project-Explorer* den Knoten *Maven-Dependencies* → *hudson-core-1.xxx.jar* mit rechts an und wählen aus dem Kontextmenü den Befehl *Maven* → *Download sources* (siehe Abb. 9–6).

Javadocs des
Core-Moduls

Dadurch werden alle Referenzen auf Klassen aus dem Hudson-Kern automatisch mit Javadoc-Tooltips versehen (siehe Abb. 9–7). Darüber hinaus erlaubt Eclipse nun auch Sprünge direkt in die Quelltexte dieser Hudson-Klassen. Gerade für Einsteiger in die Plugin-Entwicklung ist dies eine äußerst praktische Möglichkeit, Hudsons innere Vorgänge besser zu verstehen.

Abb. 9–6
Nachladen der Quelltexte
für das Modul
hudson-core

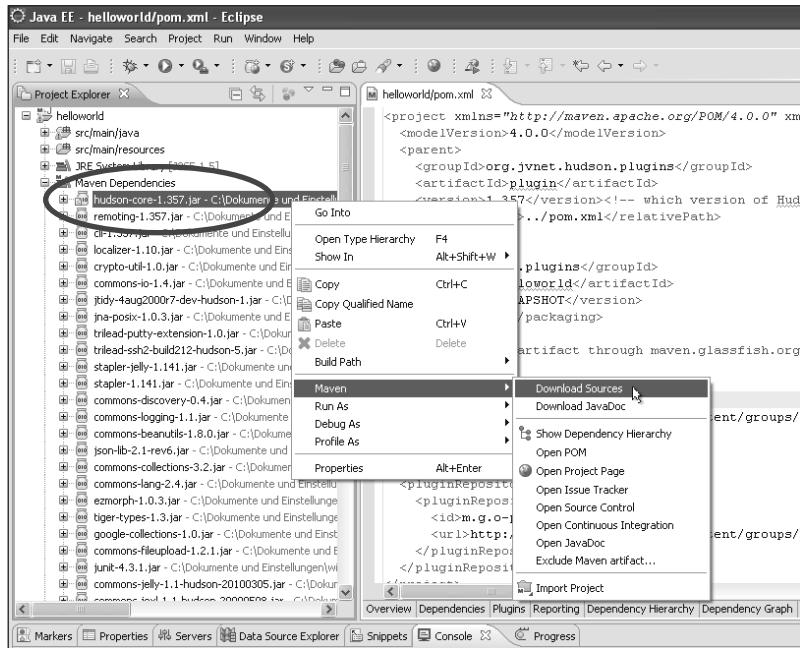
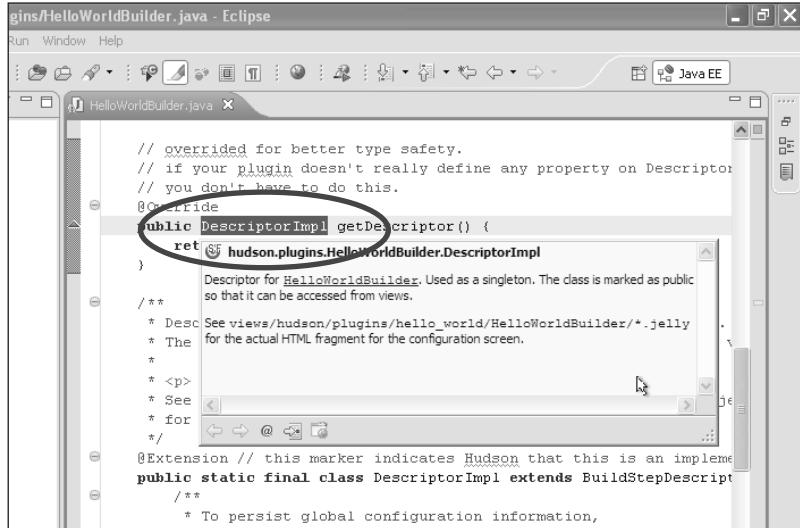


Abb. 9–7
Javadoc-Tooltips für
Klassen des Moduls
hudson-core



Debuggen mit
Breakpoints

Statt Maven von der Kommandozeile aus zu starten, verwenden Sie nun Eclipse. Dies hat den Vorteil, dass Sie in Ihren Java-Quelltexten Haltepunkte (*breakpoints*) setzen können, um die Programmausführung an diesen Stellen anzuhalten. Dies ist nicht nur praktisch bei der Fehlersuche, sondern hilft auch beim Erlernen der inneren Vorgänge

von Hudson (»Mal sehen, wann diese Methode configure überhaupt aufgerufen wird. Ich setze mal einen Breakpoint...«).

9.1.4 Zusammenfassung

Ihre Hudson-Plugin-Entwicklungsumgebung ist nun startklar: Sie können wie gewohnt in Eclipse Ihren Plugin-Java-Code entwickeln und über m2eclipse in einer Hudson-Testinstanz starten. Durch das Nachladen der Quelltexte für das Modul hudson-core erhalten Sie die Java-docs zu Hudsons Modellklassen in Eclipse angezeigt und können bei Bedarf direkt in die Hudson-Quelltexte springen. Darüber hinaus lassen sich Haltepunkte setzen, um den Programmablauf an kritischen Stellen zu pausieren.

9.2 Hudsons Plugin-Konzept

In diesem Abschnitt versorgen wir uns mit dem konzeptionellen Rüstzeug zur Plugin-Entwicklung für Hudson, bevor wir im nachfolgenden Abschnitt 9.3 Schritt für Schritt ein Beispiel-Plugin entwickeln werden.

9.2.1 Hudsons Technologie-Stapel

Hudson verwendet gleich eine ganze Reihe von bekannten und weniger bekannten Frameworks. Glücklicherweise benötigen Sie – je nach Art Ihres Plugins – keine tiefergehenden Kenntnisse in *allen* dieser Frameworks. Durch das Studium der Quelltexte existierender Plugins kommen Sie bereits sehr weit.

Die drei Frameworks Stapler, Jelly und XStream spielen hingegen eine so zentrale Rolle, dass auf sie an dieser Stelle explizit eingegangen werden soll. Abbildung 9–8 zeigt deren Zusammenhang im eingesetzten Technologie-Stapel.

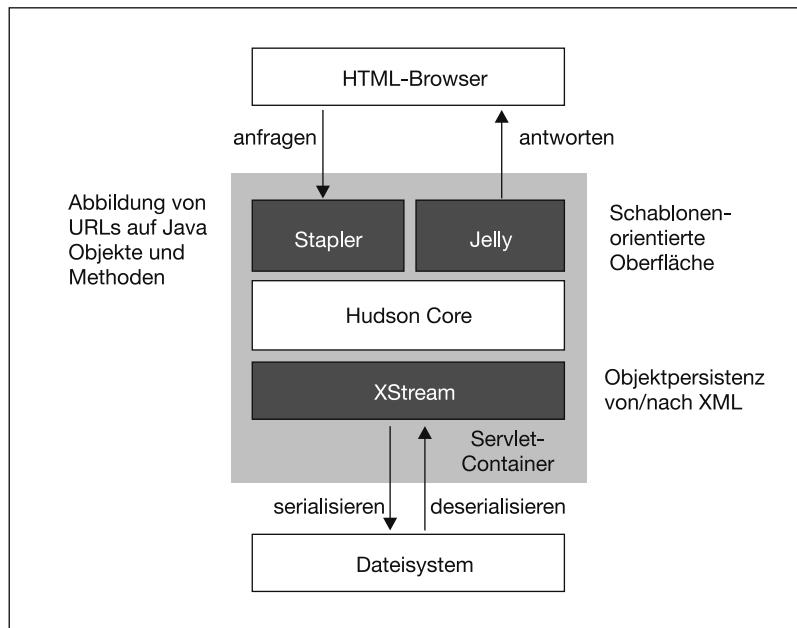
Die drei zentralen Frameworks

Hudson verwendet kein verbreitetes Webframework wie etwa Apache Struts, Apache Wicket oder Spring MVC. Stattdessen kommt das relativ unbekannte Framework Stapler (<https://stapler.dev.java.net>) zum Einsatz. Diese Technologieentscheidung dürfte einen einfachen Grund haben: Stapler ist ein weiteres Projekt des Hudson-Gründers Kohsuke Kawaguchi. Staplers Leitgedanke besteht darin, die URLs eingehender HTTP-Anfragen zu interpretieren und per Namenskonvention automatisch an Methoden serverseitiger Java-Objekte weiterzuleiten. Stapler (deutsch: Heftgerät, Klammeraffe, Tacker) heftet also URLs und zugehörige Java-Objekte automatisch zusammen.

Stapler

Abb. 9–8

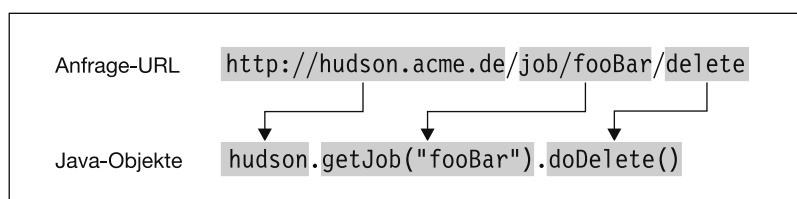
Die zentralen Frameworks
in Hudson



Die URL wird damit zu einer »sprechenden« Adresse innerhalb der serverseitigen Objekthierarchie – ein Konzept, das Ihnen vermutlich aus REST-Architekturen bekannt vorkommt. Die Stamm-URL / der Webanwendung ist dabei an die Singleton-Instanz der Klasse `Hudson` gebunden, die serverseitig nur genau einmal existiert. Da die Klasse `Hudson` eine Methode `getJob(String name)` besitzt, kann `Stapler` die URL `/job/fooBar` automatisch an dasjenige Java-Objekt weiterleiten, welches `getJob("fooBar")` zurückliefert. Die URL kann auch eine Aktion beinhalten, die auf einem Java-Objekt ausgeführt werden soll: Die URL `/job/fooBar/delete` wird an die Methode `doDelete()` des Jobs `fooBar` weitergeleitet (siehe Abb. 9–9).

Abb. 9–9

Abbildung einer URL auf
einen Java-
Methodenaufruf durch
Stapler (Beispiel)



Als Plugin-Entwickler sollten Sie mitnehmen, dass URLs und Ihre Java-Objekte automatisch verbunden werden – wenn Sie sich an die `Stapler`-Namenskonventionen halten. Die Details zur Abbildung von URLs auf Java-Objekte und Methoden können Sie der `Stapler`-Dokumentation (<https://stapler.dev.java.net>) entnehmen.

Zur Erzeugung der HTML-Oberfläche verwendet Hudson das Projekt Apache Jelly (<http://jakarta.apache.org/commons/jelly>). Mit Jelly lassen sich XML-Schablonen in ausführbaren Code umwandeln, der dann wiederum HTML anzeigen kann. Sie können sich das sehr ähnlich zu Java-Server Pages (JSP) im Zusammenspiel mit der Java-Server Pages Standard Tag Library (JSTL) vorstellen. Hudson definiert eine Reihe von eigenen Jelly-Tags, um das Erstellen der Schablonen für die Benutzeroberfläche zu vereinfachen. In Hudsons Quelltexten erkennen Sie die Jelly-Schablonen an der Dateiendung .jelly.

Jelly

HTML-Dokumente werden im Webbrowser mit AJAX-fähigen Komponenten aus der YUILibrary (<http://developer.yahoo.com/yui/>) angereichert.

Hudson verwendet zur dauerhaften Datenspeicherung keine relationale Datenbank oder vergleichbare Technologien. Stattdessen werden alle Daten im Dateisystem abgelegt – in der überwiegenden Mehrheit handelt es sich dabei um XML-Dateien.

XStream

Diese werden mit der Bibliothek XStream (<http://xstream.codehaus.org>) erzeugt, die Java-Objekte in XML-Dateien sowohl serialisieren als auch wieder rückwärts deserialisieren kann. Im einfachsten Fall erzeugt XStream dazu für alle Datenfelder eines Java-Objektes gleichnamige XML-Elemente in der Ausgabedatei: Das Feld `String name="Max"` wird in der XML-Datei also zum Element `<name>Max</name>`. Umgekehrt werden beim Einlesen für alle XML-Elemente gleichnamige Felder initialisiert: Für das XML-Element `<name>Joe</name>` würde also das Feld `name` auf den Wert »Joe« gesetzt werden usw. Natürlich lässt sich die Abbildung von Objekten auf das XML-Format mit XStream auch individuell anpassen. Details dazu entnehmen Sie der Dokumentation auf der XStream-Projektseite (Tipp: Starten Sie mit dem »Two Minute Tutorial« unter <http://xstream.codehaus.org/tutorial.html>).

Als Plugin-Entwickler ist die direkte Abbildung von Objekten auf das XML-Dateiformat insofern von Bedeutung, dass ein Umbenennen der Felder in Ihrer Java-Klasse dazu führen kann, dass bereits bestehende serialisierte XML-Dateien nicht mehr vollständig eingelesen werden. Wählen Sie Ihre Namen also mit Bedacht (oder konfigurieren Sie XStream so, dass auch bestehende Dateien korrekt eingelesen werden). Bedenken Sie auch, dass leichtfertiges Umbenennen von Klassen und Eigenschaften dazu führen kann, dass neue Versionen Ihres Plugins inkompatibel zu bereits veröffentlichten Vorgängerversionen werden – zweifellos eine der sichersten Methoden, sich den Zorn bisher zufriedener Anwender zuzuziehen ...

Insgesamt kann zusammengefasst werden, dass in den Hudson-Quelltexten sehr stark mit Namenskonventionen gearbeitet wird. Dies

Einsatz von
Namenskonventionen

vermeidet umständliche und langwierige Konfigurationsdateien, setzt aber gleichzeitig ein behutsames Vorgehen etwa beim Umbenennen von Methoden, Feldern und Dateinamen voraus. Wir werden auf das Zusammenspiel der Namenskonventionen bei der Entwicklung unseres Beispiel-Plugins näher eingehen.

9.2.2 Was muss ein Plugin-Entwickler wissen?

Egal, ob Sie Plugins für Hudson, Eclipse, Photoshop oder eine andere Plugin-unterstützende Anwendung entwickeln: Es gibt eine Reihe von Fragen, die dabei jedes Mal beantwortet werden muss. Zu den wichtigsten gehören:

- Wie erkennt die Wirtsanwendung, welche Plugins überhaupt installiert sind (*plugin discovery*)?
- Welche Aspekte der Wirtsanwendung lassen sich mit Plugins erweitern (*extension points*)?
- Wie können Aktionen, passend zum Lebenszyklus (*life cycle*) eines Plugins, ausgelöst werden, etwa eine umfangreiche Initialisierung beim Starten oder »Aufräumarbeiten« beim Herunterfahren des Plugins?

Plugin Discovery

HPI-Dateien im Plugin-Verzeichnis Hudson durchsucht bei Serverstart das Plugin-Verzeichnis <HUDSON_HOME>/plugins nach *.hpi-Dateien. Diese werden in ein gleichnamiges Verzeichnis ohne hpi-Endung entpackt. Anschließend sucht Hudson in diesen Verzeichnissen in der Manifest-Datei META-INF/MANIFEST.MF nach einer gültigen Plugin-Beschreibung. Das Manifest enthält alle weiteren Informationen, die benötigt werden, um das Plugin zu instanzieren und zu starten. Dazu gehören beispielsweise auch Abhängigkeiten zu anderen Plugins oder die Hudson-Version, für die das Plugin gedacht ist. Vielleicht wundern Sie sich, wie das Beispiel im vorausgegangenen Schnelleinstieg funktioniert hat, ohne dass Sie jemals eine solche Manifest-Datei zu Gesicht bekommen haben. Die Antwort: Beim Aufruf von hpi:run bzw. hpi:package wird das Manifest automatisch erzeugt und an die richtige Stelle gepackt. Die Datei war also durchaus da – Sie haben es nur nicht bemerkt.

Durch diesen einfachen, dateibasierten Mechanismus können Plugins einfach installiert bzw. deinstalliert werden, indem die HPI-Datei ins Plugin-Verzeichnis abgelegt wird bzw. dort gelöscht wird. Wenn Sie über Hudsons Benutzeroberfläche Plugins installieren, geschieht technisch betrachtet übrigens nichts anderes.

Beachten Sie, dass Hudson sein Plugin-Verzeichnis nur beim Neustart durchsucht und danach keine Veränderungen in diesem Verzeichnis erwartet. Frisch installierte Plugins werden also erst nach einem Neustart in der Anwendung sichtbar. Aus demselben Grund sollten Sie Plugins auch nur löschen, wenn Hudson zuvor heruntergefahren wurde.

Eine besondere Spielart stellen *.hpl-Dateien im Plugin-Verzeichnis dar. Hierbei handelt es sich nicht um vollständige Plugins, sondern lediglich um Verweise auf ein entpacktes Plugin, das an einer anderen Stelle im lokalen Dateisystem liegt (der letzte Buchstabe »l« der Dateiendung steht für *link*). Dieses Konstrukt ist für Plugin-Entwickler gedacht, die sich so das ständige Neupaketieren eines Plugins als *.hpi-Datei sparen können und Hudson direkt auf das Arbeitsverzeichnis ihrer IDE verweisen können.

HPL-Dateien

Extension Points

Hudson verfügt momentan über ca. 70 Erweiterungspunkte (*extension points*), an denen Sie Ihre eigene Funktionalität andocken können. Den Erweiterungspunkt *Builder* haben Sie bereits im Schnelleinstieg kennengelernt. In Abschnitt 9.2.3 werden wir alle momentan verfügbaren Erweiterungspunkte kurz vorstellen.

*Über 70
Erweiterungspunkte
stehen zur Verfügung*

Pro Erweiterungspunkt können auch mehrere Erweiterungen registriert werden. Ein prominentes Beispiel ist der Erweiterungspunkt *SCM*, an dem gleichzeitig Erweiterungen für unterschiedliche Versionskontrollsysteme andocken können (siehe Abb. 9–10, oben). Ein Plugin kann auch mehrere Erweiterungspunkte implementieren, die erst im Zusammenspiel die gewünschte zusätzliche Funktionalität bereitstellen.

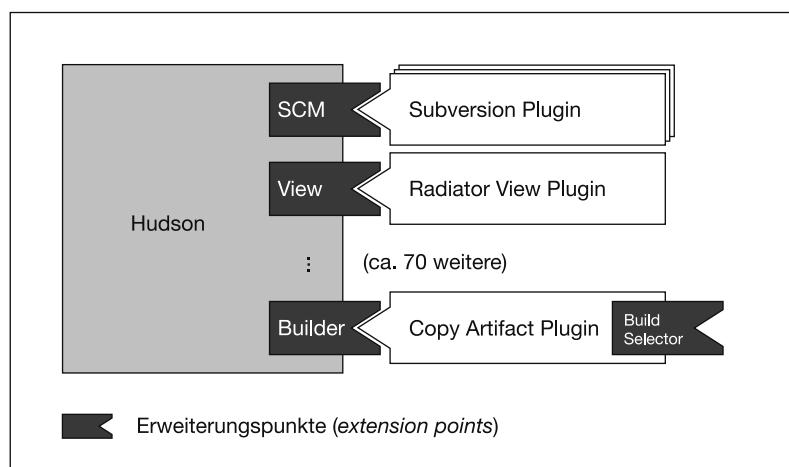


Abb. 9–10
Hudson lässt Plugins an Erweiterungspunkten andocken.

*Java-Annotation
@Extension*

Um eine Erweiterung in Hudson registrieren zu lassen, markiert der Entwickler sie im Java-Quelltext mit der Java-Annotation `@Extension`. Dadurch wird diese automatisch von Hudson gefunden und an der passenden Stelle eingefügt, z.B. als Eintrag in eine Auswahlliste. Auf welche Weise eine Erweiterung in der Benutzeroberfläche dargestellt wird (wenn überhaupt), hängt vom implementierten Erweiterungspunkt ab.

Eigene

*Erweiterungspunkte:
Plugins für Plugins*

Übrigens können Plugins auch eigene Erweiterungspunkte definieren. So lassen sich dann »Plugins für Plugins« entwickeln. Das Plugin »Copy Artifact« beispielsweise erlaubt es, Artefakte aus dem Build eines Projektes in ein anderes Projekt zu kopieren. Wie genau wird aber dieser Build festgelegt: Per Nummer? Immer der letzte Build? Immer der letzte erfolgreiche Build? Das Plugin definiert dazu einen eigenen Erweiterungspunkt `BuildSelector` (siehe Abb. 9–10, unten), mit dem sich zusätzliche Strategien nachrüsten lassen, um einen ganz bestimmten Build auszuwählen – etwa nach Erstellungsdatum, Größe der Artefakte, Betriebssystem usw.

Life Cycle

Enthält ein Plugin-Projekt eine Klasse, die von der abstrakten Klasse `java.hudson.Plugin` abgeleitet ist, werden deren Lebenszyklus-Methoden (`load`, `save`, `start`, `stop` usw.) von Hudson zum entsprechenden Zeitpunkt aufgerufen. In den Implementierungen der abstrakten Methoden können dann z.B. Initialisierungen vorgenommen werden. Benötigt ein Plugin keine Lebenszyklus-Ereignisse, so kann diese Datei weggelassen werden.

9.2.3 Hudsons Erweiterungspunkte (Extension Points)

In diesem Abschnitt betrachten wir die rund 70 Erweiterungspunkte, die Hudson momentan anbietet. Eine aktuelle Übersicht, die stets frisch aus den Java-Quelltexten generiert wird, finden Sie unter wiki.hudson-ci.org/display/HUDSON/Extension+points. Sie sollten die folgende Liste auf jeden Fall wenigstens überfliegen: Erfolgreiche Hudson-Plugin-Entwicklung hängt maßgeblich von der Wahl des geeigneten Erweiterungspunktes ab!

Aber seien Sie gewarnt: Wie bei einer gut geschriebenen Speisekarte ist es nicht ausgeschlossen, dass Sie Appetit auf Dinge bekommen, deren Existenz Sie bisher nicht einmal erahnten...

AdministrativeMonitor

Prüft den Zustand eines Hudson-Subsystems und zeigt gegebenenfalls am Kopf der Seite *Hudson verwalten* eine Warnmeldung an. Ein Beispiel ist der `HudsonHomeDiskUsageMonitor`, der laufend überwacht, ob für das Hudson-Verzeichnis noch ausreichend Speicherplatz zur Verfügung steht.

AuthorizationStrategy

Erlaubt die Definition eigener Schemas zur Rechtevergabe in Hudsons Zugriffskontrolle, analog zu den vorhandenen Strategien *Matrix-basierte Sicherheit*, *Angemeldete Benutzer dürfen alle Aktionen ausführen*, *Jeder darf alle Aktionen ausführen* usw.

BuildWrapper

Erlaubt die Ausführung von Aktionen vor und nach dem Build, z.B. Starten und Stoppen einer Datenbank, die während des Builds für Testzwecke benötigt wird.

Builder

Führt einen Teilschritt während des Builds aus. Bekannte Beispiele sind *Windows Batch Datei ausführen*, *Ant aufrufen*, *Shell ausführen* usw. Builder ist einer der beliebtesten Erweiterungspunkte überhaupt. Im Schnellstart in Abschnitt 9.1.1 haben Sie bereits einen Builder kennengelernt.

CLICommand

Dient als Ausgangspunkt für Kommandos, die per Kommandozeile an eine Hudson-Instanz übermittelt werden können. Dies ist interessant, etwa um administrative Aufgaben zu automatisieren, die auf mehreren Hudson-Servern ausgeführt werden sollen. Ein Beispiel werden Sie in Abschnitt 9.3.4 kennenlernen.

ChangeLogAnnotator

Erlaubt die Anreicherung der Seite *Änderungen*, auf der die Commit-Kommentare des Versionskontrollsystems dargestellt sind, mit zusätzlichen Informationen. Beispielsweise könnten die Ticketnummern eines Issue-Trackers (»HUDSON-1234, »#1234« o.Ä.) automatisch erkannt und mit einem Link in den Issue-Tracker versehen werden.

Cloud

Erlaubt eine dynamische Erweiterung bzw. Verringerung der Anzahl an Slave-Nodes einer Hudson-Instanz.

ComputerLauncher

Ermöglicht alternative Startmethoden, um die Verbindung zwischen dem Hudson-Master und einem Slave-Knoten herzustellen – analog zu *Starte Slave-Agenten über JNLP, Starte UNIX-Agenten über SSH usw.*

ComputerListener

Erhält Benachrichtigungen, wenn sich die Verfügbarkeit eines Knotens verändert (online/offline).

ConsoleAnnotationDescriptor

Im Normalfall ist die Konsolenausgabe eine reine Textdatei, in der Zeile für Zeile die Ausgaben der Build-Werkzeuge gesammelt werden. Dieser Erweiterungspunkt erlaubt die Einbettung von zusätzlichen Informationsobjekten (`ConsoleNote`) in diese Konsolenausgabe. Während eines Ant-Builds könnte zum Beispiel jeweils zu Beginn eines neuen Targets ein `ConsoleNote`-Objekt eingefügt werden. Bei der späteren Anzeige der Konsolenausgabe können diese Objekte als Markierung verwendet werden, um die Ausgabe visuell zu strukturieren. Zur Abgrenzung: `ConsoleAnnotationDescriptor` fügt während des Builds zusätzliche Daten in die Konsolenausgabe ein – im Gegensatz zur `ConsoleAnnotationFactory` (siehe nächster Absatz), die eine bestehende Konsolenausgabe lediglich zur Darstellung »aufhübscht«.

ConsoleAnnotatorFactory

Erlaubt die Anreicherung der Konsolendarstellung eines Builds um zusätzliche Informationen. Beispielsweise könnten per Mustererkennung URLs automatisch identifiziert und mit einem HTML-Link auf eben diese URL versehen werden. Oder alle Zeilen, die das Wort »ERROR« enthalten, werden in Rot dargestellt. Oder voll qualifizierte Java-Klassennamen werden fett angezeigt usw. Wichtig ist: Die gespeicherten Daten der Konsolenausgabe werden dabei nicht verändert, sondern es wird lediglich die Ausgabe im Browser verschönert.

CrumbIssuer

Ein CrumbIssuer ist ein Algorithmus, der einen Zufallswert erzeugt, der als *crumb* bezeichnet wird und zur Abwehr von *Cross-Site-Request-Forgery*-Angriffen (XSRF) eingesetzt wird.

Downloadable

Stellt eine JSON-Datei dar, die in regelmäßigen Intervallen von einer URL heruntergeladen wird. Ein Beispiel dafür ist die Liste der verfügbaren Plugins, JDKs, Ant- und Maven-Versionen, die eine Hudson-Instanz regelmäßig vom öffentlichen Hudson-Update-Server abruft.

ExtensionFinder

Erlaubt die Implementierung eigener Strategien, wie Erweiterungspunkte gefunden und Hudson mitgeteilt werden können. Dadurch lassen sich beliebige Dependency-Injection-Frameworks anbinden, die eigene Verfahren zur Definition von Erweiterungspunkten verwenden.

FileSystemProvisioner

Bereitet den Arbeitsbereich eines Projekts vor dem Build vor bzw. entfernt den Arbeitsbereich (etwa beim Löschen eines Projekts).

ItemListener

Erhält Benachrichtigungen über wichtige Ereignisse im Lebenszyklus eines Item-Objekts, etwa nach dem Laden von der Festplatte, dem Kopieren, Umbenennen oder Löschen. Da beispielsweise Projekte Item-Objekte sind können mit diesem Erweiterungspunkt Veränderungen in Hudsons Projektliste überwacht werden.

Job

Ermöglicht die Definition eigener Job-Typen – analog zu den Typen, die Sie beim Anlegen eines neuen Jobs angeboten bekommen (Free-Style-Job, Maven2-Job, Multikonfigurationsprojekt und Externer Job).

JobProperty

Erlaubt die Definition von zusätzlichen Eigenschaften, die pro Job gespeichert werden und auf der Konfigurationsseite des Jobs gesetzt werden können. Auf diese Art und Weise werden beispielsweise die Zugriffsrechte der Rechtevergabestrategie *Projektbasierte Matrix-Zugriffssteuerung* realisiert.

LabelFinder

Ermöglicht die automatische Zuordnung von Labels zu Hudson-Knoten. Beispielsweise könnte ein Knoten ganz automatisch – je nach seinem Betriebssystem – die Labels »solaris«, »windows-xp«, »windows-7« etc. zugeordnet bekommen. Eine manuelle Konfiguration der Labels für diesen Aspekt würde dann entfallen. Diese Idee ist so naheliegend, dass sie bereits vom Plugin »Platform Labeler« umgesetzt wird.

LauncherDecorator

Erlaubt die Dekoration (also die Erweiterung) von Aufrufen zum Starten von Betriebssystemkommandos. Diese können beispielsweise so abgeändert werden, dass sie in einer besonders abgesicherten Umgebung ausgeführt werden (etwa mit fakeroot, sudo, pfexec usw.).

Lifecycle

Erlaubt es, Hudson zu starten, zu stoppen, neu zu starten oder gar zu deinstallieren. Dieser Erweiterungspunkt wird bereits für mehrere Betriebssysteme implementiert, aber nicht alle Methoden sind dabei unter allen Betriebssystemen verfügbar. So kann sich beispielsweise Hudson unter Windows nicht selbst neu starten. Unter Unix hingegen ist dies möglich.

ListViewColumn

Erlaubt die Ergänzung weiterer Spalten in der Jobtabelle auf Hudsons Übersichtsseite. Ein Beispiel zu diesem Erweiterungspunkt finden Sie in Abschnitt 9.3.2.

MailAddressResolver

Erlaubt es, die E-Mail-Adresse eines Hudson-Benutzers zu ermitteln. Dazu könnten etwa anhand der Informationen, die zu einem Benutzer in Hudson hinterlegt sind, Abfragen in Benutzerverzeichnissen getätigter werden. Auch wären Heuristiken denkbar: »Hubert J. Farnsworth« könnte zum Beispiel durch Zusammenschieben der Initialen automatisch in hjf@acme.de gewandelt werden usw. Hudson bringt bereits eine Implementierung dieses Erweiterungspunktes mit, bei der E-Mail-Adressen mithilfe eines LDAP-Systems aufgelöst werden.

ManagementLink

Fügt einen weiteren Abschnitt auf die Seite *Hudson verwalten* hinzu – analog zu *System konfigurieren*, *Plugins verwalten*, *Knoten verwalten* usw. Dadurch lassen sich Funktionen mit systemweiten Auswirkungen elegant in die Übersicht der administrativen Aufgaben einfügen.

MatrixAggregatable

Ein Publisher kann diesen Erweiterungspunkt implementieren und die Ergebnisse der einzelnen Builds eines Multikonfigurationsprojekts zusammenfassen. Das kann nützlich sein, um eine tabellarische Vergleichsübersicht über die einzelnen Builds zu erstellen.

MatrixAggregator

Führt die Zusammenfassung der Ergebnisse aus einem MatrixRun zu einem MatrixBuild durch. Ein MatrixAggregator wird jeweils nach jedem MatrixRun aufgerufen und kann entscheiden, ob weitere MatrixRuns durchgeführt werden sollen oder aber der komplette MatrixBuild abgebrochen wird.

Node

Dieser Erweiterungspunkt dient als Basis für eigene Typen von Slave-Knoten. Tipp: In der Praxis werden Sie in diesem Falle besser von der Klasse `hudson.model.Slave` ableiten, die `Node` erweitert.

NodeMonitor

Erlaubt die Überwachung von Eigenschaften angelegter Hudson-Knoten. Sie können mit diesem Erweiterungspunkt in der Knoten-Übersichtsseite zusätzliche Eigenschaften hinzufügen – analog zu *Antwortzeit*, *Freier Plattenplatz* oder *Zeitdifferenz*. Ein NodeMonitor kann einen Knoten abschalten, wenn die gemessene Eigenschaft einen zulässigen Bereich verlässt, z.B. wenn die Antwortzeit übers Netz zu groß wird oder der Festplattenplatz zur Neige geht.

NodeProperty

Erlaubt das Hinzufügen von eigenen Eigenschaften zu Knoten. Die Werte der Eigenschaften werden dauerhaft gespeichert und sind in der jeweiligen Konfigurationsseite des Knotens veränderbar. Beispielsweise lassen sich so die Umgebungsvariablen pro Knoten anpassen.

Notifier

Ein Notifier erweitert Publisher und wird ganz am Ende eines Builds aufgerufen und informiert Menschen wie Drittsysteme über den Ausgang des Builds. Ganz ähnlich wie Recorder, der ebenfalls Publisher erweitert, hat Notifier Zugriff auf die kompletten Build-Ergebnisse, etwa um eine zusammenfassende E-Mail mit den wichtigsten Resultaten zu verfassen. Im Gegensatz zu Recorder kann Notifier aber nicht das Gesamtergebnis des Builds verändern (erfolgreich, instabil, fehlgeschlagen).

PageDecorator

Erlaubt das Einschleusen von HTML-Fragmenten in alle HTML-Seiten, die von Hudson erzeugt werden. Die Fragmente können entweder am Seitenende unmittelbar vor dem </body>-Element eingefügt werden (z.B. für Markierungen zur Messung der Seitenaufrufe) oder aber unmittelbar vor dem </head>-Element (z.B. um eigene CSS-Dateien einzubinden). Auf diese Weise lässt sich beispielsweise das optische Erscheinungsbild Hudsons durch ein eigenes Stylesheet an das Corporate Design des eigenen Unternehmens anpassen. Tipp: Das Plugin »Page Markup« setzt diese Idee bereits um.

ParameterDefinition

Erlaubt die Definition eigener Parametertypen für parametrisierte Builds. Ein eigener Parametertyp könnte beispielsweise ein Text sein, der einen bestimmten regulären Ausdruck erfüllen muss. Auf diese Weise könnte man die konsistente Angabe von Ticketnummern, Hostnamen oder E-Mail-Adressen sicherstellen. Das Plugin »Validating String Parameter« leistet genau dies. Ein anderer Anwendungsfall dieses Erweiterungspunktes könnte ein Auswahlfeld sein, in dem dynamisch alle für ein Deployment verfügbaren Server angeboten werden.

PeriodicWork

Erlaubt die regelmäßige Ausführung einer (kurzen) Tätigkeit im Hintergrund. Typische Beispiele sind die Überwachung des freien Festplattenplatzes oder die Messung der momentanen Systemauslastung für statistische Zwecke.

PluginServletFilter

Erlaubt das Einhängen eines Servlet-Filters in die Servlet-Kette der Webanwendung. Dadurch können Anfragen an Hudson vor der Bear-

beitung in Stapler modifiziert werden. Gleiches gilt für Antworten, die von Hudson zurück an den Browser geschickt werden. Die bekannteste Implementierung dieses Erweiterungspunktes dürfte das äußerst verbreitete Plugin »Green Balls« sein, das in Hudsons Antworten alle Verweise auf Bilddateien mit blauen Bällen in Verweise auf grüne Bälle ändert.

PluginStrategy

Dieser Erweiterungspunkt dient dazu, Hudson-Plugins in anderen Dependency-Injection-Containern zu laden. Sie werden auf diesen Erweiterungspunkt nur in sehr exotischen Fällen angewiesen sein.

QueueDecisionHandler

Entscheidet darüber, ob ein bestimmter Job in die Build-Warteschlange aufgenommen werden darf.

QueueSorter

Erlaubt eine Neusortierung der Build-Warteschlange nach beliebigen Gesichtspunkten. Dadurch lassen sich beispielsweise bestimmte Projekte bevorzugt bauen oder weniger dringliche Builds an das Ende der Warteschlange platzieren (»Die können laufen, wenn sonst nichts los ist«).

Recorder

Ein Recorder erweitert Publisher und wird am Ende eines Builds aufgerufen. Ganz ähnlich wie Notifier, der ebenfalls Publisher erweitert, hat Recorder Zugriff auf die kompletten Build-Ergebnisse, etwa um wichtige Resultate langfristig zu archivieren. Im Gegensatz zu Notifier kann Recorder aber das Gesamtergebnis des Builds bestimmen (erfolgreich, instabil, fehlgeschlagen). Daher kommen nach einem Build zunächst alle Recorder zum Zuge, bevor die Notifier Benachrichtigungen versenden. Somit ist gewährleistet, dass das Gesamtergebnis von allen Notifiern konsistent berichtet wird.

RepositoryBrowser

Erlaubt die Integration eines Repository-Browsers, wie z.B. ViewCVS oder Atlassian FishEye. Innerhalb von Hudson angezeigte Revisionsnummern und Dateipfade werden dadurch mit Links in die entsprechenden Stellen im Repository-Browser hinterlegt.

RetentionStrategy

Erlaubt die Definition einer eigenen Strategie zur Steuerung des An- und Abschaltzeitpunktes von Slave-Knoten. Hudson bringt bereits ein paar einfache Implementierungen dieses Erweiterungspunktes mit, z.B. *Slave immer angeschaltet lassen* oder *Slave zeitgesteuert anschalten*. Komplexere Strategien könnten beispielsweise dafür sorgen, dass aus einem Pool an Knoten eine bestimmte Mindestanzahl immer angeschaltet bleibt (»Mindestens 5 Linux-Rechner von 20 sollten immer laufen. Der Rest soll sich bei Bedarf zuschalten«).

RootAction

Aktionen, die diesen Erweiterungspunkt implementieren, werden direkt unterhalb der Stamm-URL / registriert und sind von jeder Seite aus verfügbar. Damit lassen sich sehr einfach globale Funktionen inklusive zugehörigem Symbol in der Funktionsleiste am linken Seitenrand hinzufügen.

Run

Ermöglicht die Definition eigener Typen, die das Ergebnis eines bestimmten Laufs eines Jobs festhalten. Run-Typen werden immer im Zusammenspiel mit entsprechenden Job-Typen eingesetzt, also ExternalRun mit ExternalJob, AbstractBuild mit AbstractProject usw. Typischerweise werden also die Erweiterungspunkte Job und Run gemeinsam implementiert. Ein Beispiel dazu finden Sie im Plugin »Ivy« in den Klassen AbstractIvyBuild bzw. AbstractIvyProject.

RunListener

Erhält Benachrichtigungen über wichtige Ereignisse im Lebenszyklus aller in Hudson stattfindenden Läufe, z.B. Start, Ende, Abschluss oder Lösung eines Laufs. Über diesen Erweiterungspunkt können Sie somit die komplette Build-Aktivität zentral mitverfolgen. Hudson verwendet dies beispielsweise intern bei Projekten, die Testergebnisse aus nachgelagerten Projekten zusammenfassen sollen: Über einen RunListener wird zunächst abgewartet, bis alle nachgelagerten Projekte ihren Abschluss signalisiert haben. Anschließend können die Ergebnisse dieser Projekte eingesammelt und zusammengefasst werden.

SCM

Erlaubt die Anbindung von Versionskontrollsystmen, wie etwa Subversion, Perforce, Git, Mercurial usw. Darüber hinaus können auch

etwas ausgefallenere Quellen angeschlossen werden, sofern sie die Schnittstelle eines Versionskontrollsystems erfüllen können: Das Plugin »URL SCM« beispielsweise lässt eine Datei, die sich per URL abrufen lässt, wie den Inhalt eines ausgewachsenen Versionskontrollsysteins aussehen. Wenn Sie die Implementierung eines eigenen SCM-Plugins vorhaben, finden Sie speziell für diesen Fall weitere Hinweise im Hudson-Wiki unter <http://wiki.hudson-ci.org/display/HUDSON/Writing+an+SCM+plugin>.

SCMListener

Erhält Benachrichtigungen über Hudsons SCM-Aktivitäten zu Beginn eines Builds, insbesondere über die zurückgemeldeten Änderungen im Versionskontrollsysteem.

SaveableListener

Erhält Benachrichtigungen, wenn sogenannte Saveable-Objekte abgespeichert wurden. Diese Objekte aus Hudsons Datenmodell werden als XML-Dateien dauerhaft im Dateisystem abgelegt. Hudson verwendet diesen Erweiterungspunkt intern beispielsweise in der Klasse OldDataMonitor, um zu protokollieren, welche Objekte noch in einem veralteten Datenformat vorliegen bzw. bereits in der Zwischenzeit in einem aktuellen Datenformat gespeichert wurden.

SecurityRealm

Erlaubt die Anbindung von beliebigen Benutzerverzeichnissen, die beispielsweise aus einer relationalen Datenbank, einem Excel-Dokument oder einer Textdatei gespeist werden könnten. Das Plugin »ActiveDirectory« implementiert ebenfalls diesen Erweiterungspunkt.

Solution

Dieser Erweiterungspunkt implementiert Vorschläge, wie man mit einem <HUDSON_HOME>-Verzeichnis umgehen könnte, wenn der Festplattenplatz zur Neige geht.

TestDataPublisher

Steuert TestActions zu Testergebnissen bei und ermöglicht es so, Testergebnisse um zusätzliche Aktionen und Benutzeroberflächenelemente anzureichern. Beispielsweise könnten auf diese Weise Anwender über die Links effizient Tickets für fehlgeschlagene Tests anlegen oder Tests mit unterschiedlichen Symbolen markieren.

TestResultParser

Erlaubt die Definition eines Parsers für Testergebnisse in einem beliebigen Dateiformat. Dadurch kann Hudsons bestehende Infrastruktur zur Darstellung von Testergebnissen verwendet werden, ohne dass ein zusätzlicher Publisher für jedes alternative Dateiformat entwickelt werden muss. Sollte Ihr Testwerkzeug beispielsweise Berichte als CSV-Datei ausgeben, so wäre `TestResultParser` ein guter Ausgangspunkt zur Einbindung dieser Daten. Hudson bringt eine Implementierung dieses Erweiterungspunktes für das Dateiformat JUnit mit.

ToolInstallation

Erlaubt die Definition von Werkzeugen, die automatisch auf einem Hudson-Knoten installiert werden sollen (»Was wird installiert?«). Hudson beinhaltet bereits Implementierungen dieses Erweiterungspunktes für die Installation von Ant, Maven und JDKs. Als weitere Werkzeuge würden sich vor allem Compiler, Development Kits oder Testwerkzeuge anbieten, die von Hudson automatisch auf angeschlossene Knoten verteilt werden sollen.

ToolInstaller

Erlaubt die Definition alternativer Installationsmethoden, mit denen ein Werkzeug auf einem Hudson-Knoten installiert werden kann (»Wie wird installiert?«). Hudson bringt bereits ein paar Implementierungen dieses Erweiterungspunktes mit, z.B. die Installation durch Aufruf einer Kommandozeilenanwendung, durch Herunterladen einer Datei von einer URL oder durch Entpacken eines ZIP-Archivs.

ToolLocationTranslator

Erlaubt die Definition einer Strategie für die Bestimmung des Installationspfades eines Werkzeugs für einen gegebenen Knoten. Auf diese Weise könnten sich beispielsweise firmenweite Konventionen zur Ablage von Werkzeugen auf Hudson-Knoten konsistent einhalten lassen. Die Alternative dazu wäre die manuelle Angabe des Installationspfades als Eigenschaft pro Werkzeug (siehe dazu Erweiterungspunkt `ToolProperty`).

ToolProperty

Erlaubt die Definition von zusätzlichen Eigenschaften, die pro Werkzeug gespeichert werden und in den Konfigurationseinstellungen des Werkzeugs gesetzt werden können.

TopLevelItem

Erlaubt die Definition von Objekten, die »auf oberster Ebene« in Hudson verwaltet werden. Informell können Sie sich darunter die Zeilen in der Jobtabelle in Hudsons Übersichtsseite vorstellen, also alle Free-Style-, Maven2- und Multikonfigurationsprojekte sowie überwachte externe Jobs.

TransientProjectActionFactory

Erlaubt das Hinzufügen von Aktionen zu allen angelegten Projekten. Normalerweise ergeben sich die angebotenen Aktionen auf Projektebene aus den definierten Build-Schritten dieses Projekts. Haben Sie beispielsweise während des Builds Javadoc-Dokumentation erzeugt, erscheint am linken Seitenrand automatisch die Aktion *Javadocs*. Manchmal möchten Sie jedoch pauschal eine weitere Aktion allen Projekten hinzufügen, ohne die gespeicherten Daten der Projekte auf der Festplatte verändern zu wollen. In diesem Fall würden Sie diesen Erweiterungspunkt einsetzen. Ein Beispiel dazu finden Sie im Plugin »Job Configuration History«, das die Konfigurationen angelegter Projekte versionieren kann.

Trigger

Erlaubt die Definition eigener Build-Auslöser. Sie kennen bereits Hudsons Implementierungen dieses Erweiterungspunktes, *Source Code Management System abfragen* bzw. *Builds zeitgesteuert starten*, die bei Änderungen im Versionskontrollsysteem bzw. streng nach Zeitplan neue Builds auslösen. Als weitere Implementierung könnte man sich auch einen »Sandra Bullock«-Trigger vorstellen, der in Anlehnung an den Action-Film »Speed« bei Unterschreiten einer Systemauslastung von 50% einen neuen Build startet...

UDPBroadcastFragment

Slave-Knoten können im Netz einen Master-Knoten in der Nachbarschaft entdecken und sich automatisch diesem anschließen. Technisch wird dies über UDP-Broadcast-Pakete realisiert, die von den Slaves ins Netz verschickt werden (»Hallo, ist da irgendwo ein Hudson-Server?«). Der Master antwortet daraufhin mit einem UDP-Paket, das Details zum Verbindungsaufbau beinhaltet (»Hallo, ich bin euer neuer Hudson-Master, Version 1.357. Ihr erreicht mich unter <http://hudson.acme.de>, unter Port 8888« usw.). Der Erweiterungspunkt **UDPBroadcastFragment** erlaubt das Einfügen kleiner XML-Schnipsel in diese UDP-Antwort des

Masters. So verwendet das Plugin »Swarm« beispielsweise diesen Erweiterungspunkt, um in UDP-Paketen eine Gruppenkennung mitzuschicken. Dadurch melden sich Slave-Knoten nur am jeweils »richtigen« Hudson-Server ihrer Gruppe an.

UpdateCenterConfiguration

Über diesen Erweiterungspunkt können Sie beeinflussen, von woher Hudson seine Updates bezieht, etwa von einem firmeninternen Server. Der Erweiterungspunkt hat inzwischen etwas von seiner Bedeutung verloren, da es seit Version 1.333 möglich ist, eigene Update-Server per Konfiguration in der Datei `hudson.model.UpdateCenter.xml` anzugeben – sogar mehrere parallel.

UserNameResolver

Erlaubt es, den vollen Namen (»Hubert J. Farnsworth«) eines Hudson-Benutzers zu ermitteln. Dazu könnten etwa anhand der Informationen, die zu einem Benutzer in Hudson hinterlegt sind, Abfragen in Benutzerverzeichnissen getätigten werden.

UserProperty

Erlaubt die Definition von zusätzlichen Eigenschaften, die pro Benutzer gespeichert werden und auf der Konfigurationsseite des Benutzers gesetzt werden können.

View

Erlaubt alternative Darstellungen von `TopLevelItem`-Objekten einer Hudson-Instanz – also der angelegten Jobs. Dadurch können diese nicht nur tabellarisch wie auf Hudsons Übersichtsseite angezeigt werden, sondern auch baumartig verschachtelt (Plugin »Nested View«) oder als plakative Farbflächen (Plugin »Radiator View«). Besonders in Systemen mit mehr als 100 Jobs kommt den Views große Bedeutung zu, da sie maßgeblich dazu beitragen, den Überblick über eine große Anzahl angelegter Jobs zu behalten.

Widget

Erlaubt die Definition von rechteckigen Bereichen, die in der Benutzeroberfläche in der linken Seitenleiste angezeigt werden. Eine Implementierung dieses Erweiterungspunktes kennen Sie bereits: den Kasten »Build-Verlauf«, der die Ergebnisse der jüngsten Builds eines Projekts enthält.

9.3 Beispiel: Das Plugin »Artifact Size«

Vermutlich sind Ihnen bei der Lektüre der momentan verfügbaren Erweiterungspunkte bereits Ideen für eigene Experimente und Erweiterungen gekommen. In diesem Abschnitt lernen Sie die wichtigsten Stationen der Plugin-Entwicklung an einem durchgängigen Beispiel kennen. Wir beginnen dabei zunächst mit einem »Minimal«-Beispiel und bauen das Plugin dann immer weiter aus.

Auf der Website zum Buch, www.ci-mit-hudson.de, finden Sie die vollständigen Quelltexte des hier besprochenen Plugins »Artifact Size« zum Herunterladen, so dass Sie die Erläuterungen in Ihrer IDE nachvollziehen können.

Quelltexte herunterladen

Um endlose Seiten mit Listings zu vermeiden, sind im Buch nur die entscheidenden Passagen aus den Quelltexten abgedruckt. Auch wurden aus Platzgründen import-Anweisungen und lange Kommentare entfernt. Darüber hinaus müsste produktionstauglicher Code an vielen Stellen wesentlich defensiver programmiert werden, also viel mehr Überprüfungen auf korrekte Eingabewerte enthalten. Um Hudson-spezifische Anweisungen nicht in diesen Überprüfungen untergehen zu lassen, wurden Letztere nur sehr reduziert eingesetzt.

Kompakte Listings

Das Plugin »Artifact Size« wird zunächst in einer englischen Version realisiert. Tragen Sie das Durcheinander aus deutschen und englischen Texten in den Abbildungen zunächst mit Fassung – in Abschnitt 9.3.7 werden wir uns um eine vollständige Lokalisierung des Plugins ins Deutsche kümmern.

Wo ist die deutsche Version des Plugins?

9.3.1 Die Aufgabe

Kennen Sie das? Sie haben einen trickreichen Build-Prozess unter Hudson eingerichtet, der ganz vollautomatisch die Installationsdateien Ihrer Software erstellt. Diese Dateien werden innerhalb von Hudson als Artefakte gesichert und stehen so zum Abruf bereit. So weit, so gut.

Leider fehlt Ihnen inzwischen total der Überblick, welche Datens Mengen pro Build in Ihren Projekten auflaufen. Wäre es also nicht schön, gleich auf Hudsons Startseite pro Projekt die Gesamtgröße aller Artefakte des letzten Builds angezeigt zu bekommen?

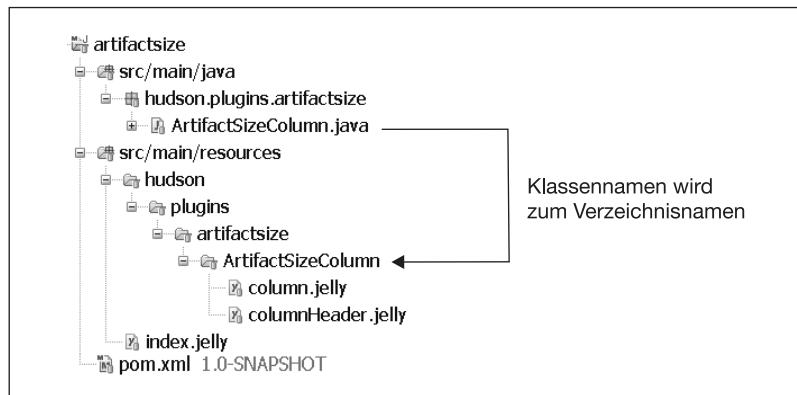
Es liegt auf der Hand, dass wir unsere Aufgabe durch Implementierung mehrerer Erweiterungspunkte lösen werden. Aber welche? In der Tat ist mit der richtigen Wahl der Erweiterungspunkte bereits einer der wichtigsten Schritte hin zur Lösung getan.

9.3.2 Erweiterungspunkt »ListViewColumn«

Wir haben uns nach ausgiebigem Studium der Erweiterungspunkte in Abschnitt 9.2.3 für `ListViewColumn` entschieden. Für eine erste Version unseres Plugins sind lediglich fünf Dateien mit zusammengenommen etwa 50 Zeilen Quelltext notwendig. Die Dateien müssen allerdings in einer bestimmten Verzeichnisstruktur abgelegt sein (Abb. 9–11):

Abb. 9–11

Datei-Layout des Plugins



ArtifactSizeColumn.java

Die Datei `ArtifactSizeColumn.java` enthält die eigentliche Funktionalität des Plugins in wenigen Zeilen Java-Code. Betrachten wir kurz den Code von oben nach unten (Listing 9–3):

`AbstractSizeColumn` erweitert die Klasse `ListViewColumn` – das Hudson-Framework stellt also bereits eine passende Basisklasse für neue Spalten zur Verfügung. Die Methode `getArtifactSize` berechnet die Gesamtgröße aller Artefakte eines Builds durch Auswertung des Datenmodells. Die Klassen `Job`, `Run`, `Artifact` entsprechen dabei den Konzepten, die Sie bereits von der interaktiven Nutzung der Weboberfläche kennen.

Am wichtigsten ist die innere Klasse `ArtifactSizeColumnDescriptor`: Sie ist zum einen mit der Annotation `@Extension` gekennzeichnet, zum andern erweitert sie `ListViewColumnDescriptor`. Diese zwei Aspekte sorgen dafür, dass Hudson die Java-Klassen als Plugin erkennt. Die Integration in die Benutzeroberfläche (»Wo taucht mein Plugin auf?«) ist damit bereits vorgegeben: Wenn Sie in Hudson eine neue Ansicht anlegen, erscheint – ganz ohne weiteres Zutun – eine neue Spalte *Artifact Size*.

Listing 9–3

ArtifactSizeColumn.java

```

package hudson.plugins.artifactsize;
[...]
public class ArtifactSizeColumn extends ListViewColumn {

    public String getArtifactSize(Job job) {
        ...
    }
}
  
```

```

        Run lastRun = job.getLastCompletedBuild();
        if (lastRun != null) {
            long totalSize = 0;
            List<Artifact> artifacts = lastRun.getArtifacts();
            for (Artifact artifact : artifacts) {
                totalSize += artifact.getFile().length();
            }
            return totalSize + " byte(s)";
        } else {
            return "No build found.";
        }
    }

    @Extension
    public static class ArtifactSizeColumnDescriptor extends
        ListViewColumnDescriptor {

        @Override
        public String getDisplayName() {
            return "Artifact Size";
        }
    }
}

```

Die zwei Jelly-Dateien `columnHeader.jelly` bzw. `column.jelly` enthalten HTML-Schnipsel, die später in der Benutzeroberfläche als Kopf bzw. Inhalt der neuen Spalte eingebaut werden. Die Namen dieser Jelly-Dateien sind durch den Erweiterungspunkt `ListViewColumn` vorgegeben. Jelly arbeitet mit HTML-Schablonen, in denen Ausdrücke durch dynamisch berechnete Werte ersetzt werden. Diese Ausdrücke erkennen Sie an vorausgehenden Dollarzeichen und den geschwungenen Klammern. Die Angabe `${it.getArtifactSize(job)}` wird so durch den Rückgabewert der Methode `getArtifactSize` des Objekts `it` (also des `ArtifactSizeColumn`-Objekts) ersetzt. Als Parameter wird der Job übergeben, der in der momentanen Zeile dargestellt werden soll.

*column.jelly und
columnheader.jelly*

```

<j:jelly xmlns:j="jelly:core">
    <th>Artifact Size</th>
</j:jelly>

<j:jelly xmlns:j="jelly:core">
    <td>${it.getArtifactSize(job)}</td>
</j:jelly>

```

Listing 9–4
columnHeader.jelly

Listing 9–5
column.jelly

Beachten Sie die Namenskonvention, die Hudson verwendet, um die Jelly-Dateien einer Java-Klasse zuzuordnen: Beide Jelly-Dateien befinden sich in einem Verzeichnis, das den Namen der zugehörigen Java-Klasse trägt – inklusive der Großschreibung und Struktur der Elternverzeichnisse. Eclipse kommentiert diese ungewöhnliche Konstellation

sogar mit der Warnung »The type ArtifactSizeColumn collides with a package.« Sie haben also paradoxerweise genau dann alles richtig gemacht, wenn Sie diese Eclipse-Warnung sehen.

index.jelly

In der Datei *jelly.xml* steht die Beschreibung des Plugins, so wie sie später in Hudsons Plugin-Manager auftauchen wird:

Listing 9–6

index.jelly

```
<div>
    Shows the total size of all artifacts of the last build
    in a list view column.
</div>
```

pom.xml

Die Datei *pom.xml* wird zur Definition von Abhängigkeiten und zur späteren Paketierung des Plugins im HPI-Format benötigt:

Listing 9–7

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <parent>
        <groupId>org.jvnet.hudson.plugins</groupId>
        <artifactId>plugin</artifactId>
        <version>1.358</version>
    </parent>

    <groupId>hudson.plugins</groupId>
    <artifactId>artifactsize</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>hpi</packaging>
    <name>Artifact Size Plugin</name>
    <description>Displays the size of build artifacts.</description>
</project>
```

Das neue Plugin testen

Mehr wird für unser erstes Plugin nicht benötigt. Sie können das Plugin mit dem Maven-Kommando `hpi:run` in einer Hudson-Testinstanz starten. Legen Sie ein paar Projekte mit archivierten Artefakten an und fügen Sie dann auf der Startseite eine neue Listenansicht hinzu (mit dem »+«-Symbol über der Liste der Projekte). Die neue Ansicht wird automatisch eine neue Spalte *Artifact Size* enthalten (Abb. 9–12).

Abb. 9–12

*Listenansicht mit neuer Spalte *Artifact Size**



The screenshot shows a Hudson project list. At the top, there's a header bar with tabs for 'Alle', 'Artifact Size View', and a '+' button. Below the header is a table with columns: S, W, Job, Letzter Erfolg, Letzter Fehlschlag, Letzte Dauer, and Artifact Size. The 'Artifact Size' column is circled in red. The table contains three rows of data:

S	W	Job	Letzter Erfolg	Letzter Fehlschlag	Letzte Dauer	Artifact Size
		freestyle	2 Stunden 48 Minuten (#1)	Unbekannt	2,3 Sekunden	8128790 byte(s)
		glotr	44 Sekunden (#2)	Unbekannt	0,26 Sekunden	1183834 byte(s)
		greetr	Unbekannt	Unbekannt	Unbekannt	No build found.

At the bottom of the table, there are links for 'Symbol: S M L', 'Legende', and three buttons: 'Alle Builds', 'Nur Fehlschläge', and 'Nur Fehlschläge aus letzten Builds'.

Obwohl das Plugin seinen Zweck – die Anzeige der erzeugten Datenmengen – bereits erfüllt, hat unsere Lösung einen kleinen Schönheitsfehler:

Die Anzeige in Bytes lässt sich bei großen Datenmengen schlecht ablesen (»Sind das jetzt 9 oder 10 Ziffern?«). Hier wäre eine Angabe in gerundeten Mega- oder Gigabytes sinnvoller. Diesen Aspekt sollte der Anwender also in einer Konfigurationseinstellung pro Ansicht festlegen können (Abb. 9–13).

Runden der Dateigrößen

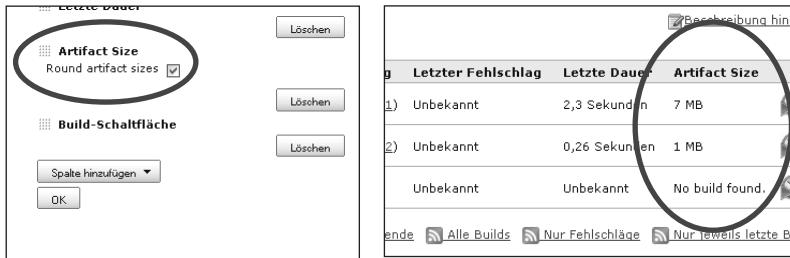


Abb. 9–13

Neue Konfigurationsoption (links), Auswirkung in der Listenansicht (rechts)

Wir werden unser neues Funktionsmerkmal in drei Schritten implementieren:

Im ersten Schritt erweitern wir das Datenmodell: Die Klasse `ArtifactSizeColumn` erhält dazu ein zusätzliches Feld `roundedFormat` vom Typ `boolean`, nebst einer zugehörigen Getter-Methode und einem Konstruktor mit Parameter. Die Getter-Methode wird benötigt, um für Konfigurationsseiten der Benutzeroberfläche den aktuellen Wert abzufragen. Zusätzlich erzeugen wir einen Konstruktor mit einem Parameter für `roundedFormat`. Den Konstruktor markieren wir mit der Hudson-Annotation `@DataBoundConstructor`. Dadurch erzeugt Hudson neue Instanzen von `ArtifactSizeColumn` für uns automatisch mit den Einstellungen aus der Konfigurationsseite. Zur Erinnerung: Dadurch, dass wir diese Angaben in der Plugin-Klasse und nicht im Deskriptor machen, gelten sie pro `ListView` und nicht global.

```
public class ArtifactSizeColumn extends ListViewColumn {
    [...]
    private final boolean roundedFormat;

    @DataBoundConstructor
    public ArtifactSizeColumn(boolean roundedFormat) {
        this.roundedFormat = roundedFormat;
    }

    public boolean isRoundedFormat() {
        return roundedFormat;
    }
    [...]
}
```

Listing 9–8

*Erweiterungen in
ArtifactSizeColumn.java*

Im zweiten Schritt erweitern wir die Benutzeroberfläche: Wir ergänzen dazu die Konfigurationsmaske auf der Seite *Ansicht bearbeiten*. Dies geschieht durch die Datei config.jelly, die wir neben den vorhandenen Jelly-Dateien unseres Plugins ablegen. In config.jelly definieren wir die Datenfelder, die in der Konfigurationsmaske dargestellt werden sollen – in unserem Falle lediglich eine Checkbox zum An- bzw. Abwählen der Rundung.

Listing 9–9

config.jelly

```
<j:jelly xmlns:j="jelly:core" xmlns:st="jelly:stapler"
    xmlns:d="jelly:define" xmlns:l="/lib/layout"
    xmlns:t="/lib/hudson" xmlns:f="/lib/form">
    <f:entry title="Round artifact sizes" field="roundedFormat">
        <f:checkbox checked="${it.isRoundedFormat()}" />
    </f:entry>
</j:jelly>
```

Beachten Sie, dass wir hier lediglich eine Jelly-Datei mit einem bestimmten Namen (config.jelly) ablegen mussten. Wo diese dann in der Benutzeroberfläche auftauchen wird, ist durch den Erweiterungspunkt vorgegeben. Manche Erweiterungspunkte unterstützen mehrere solcher Jelly-Dateien, z.B. global.jelly, die auf der Konfigurationsseite *System verwalten* eingefügt wird. Welche Jelly-Dateien von einem bestimmten Erweiterungspunkt unterstützt werden, entnehmen Sie der Javadoc-Dokumentation des jeweiligen Erweiterungspunkts.

Im abschließenden, dritten Schritt überarbeiten wir die Formatierung der Gesamtgröße der Artefakte in Abhängigkeit der neuen Konfigurationseinstellung roundedFormat. Die eigentliche Arbeit des Rundens delegieren wir dabei an eine Funktion aus dem Apache-Commons-Projekt:

Listing 9–10

Änderungen in

ArtifactSizeColumn.java

```
public String getArtifactSize(Job job) {
    Run lastRun = job.getLastBuild();
    if (lastRun != null) {
        long totalSize = 0;
        List<Artifact> artifacts = lastRun.getArtifacts();
        for (Artifact artifact : artifacts) {
            totalSize += artifact.getFile().length();
        }
        if (roundedFormat) {
            return FileUtils.byteCountToDisplaySize(totalSize);
        } else {
            return totalSize + " byte(s)";
        }
    } else {
        return "No build found.";
    }
}
```

9.3.3 Erweiterungspunkte »Recorder« und »Action«

Die momentane Implementierung ermittelt die Artefaktgrößen bei *jedem* Aufbau der Listenansicht – wenn Sie also 100 Projekte mit jeweils 10 Artefakten angelegt haben, müssen die Dateigrößen von 1.000 Dateien aus dem Dateisystem abgefragt werden. In der Folge hängt die Benutzeroberfläche jedes Mal für einen Moment...

Da sich aber die Artefakte eines Builds nachträglich nicht mehr verändern, wäre es viel effizienter, die Gesamtgröße der Artefakte nur einmal am Ende eines Builds zu berechnen und dann diesen Wert abzuspeichern und wiederzuverwenden. Für die Anzeige in den Listenansichten könnte auf diesen einzelnen Wert wesentlich schneller zugegriffen werden, als 1.000 Dateigrößen aus dem Dateisystem abzufragen.

Der Erweiterungspunkt Recorder erlaubt, am Ende eines Builds den Arbeitsbereich eines Projektes zu durchsuchen und daraus den Erfolg oder Fehlschlag eines Builds abzuleiten. Recorder sind – der Name deutet es an – auch sehr praktisch, um Build-Ergebnisse dauerhaft festzuhalten. Sie kennen beispielsweise vermutlich bereits die Post-Build-Aktion *Veröffentliche JUnit-Testergebnisse*, die durch die Klasse `JUnitResultArchiver` realisiert wird, welche Recorder implementiert.

Recorder

Eine ideale Ergänzung zu Recorder stellt eine Action dar, mit der Hudson-Objekten zusätzliche Informationen hinzugefügt werden können. Der `JUnitResultArchiver` verwendet beispielsweise eine Action, um eine Zusammenfassung von JUnit-Ergebnissen im zugehörigen Build-Objekt zu speichern. Der Name Action ist ein Hinweis darauf, dass diese Daten oftmals durch eine eigene Aktion in der Benutzeroberfläche angezeigt werden können.

Action

In unserem Beispiel-Plugin werden wir mithilfe eines Recorder und einer Action am Ende eines Builds die Größe der Artefakte berechnen und dauerhaft im Build-Objekt speichern. Wir gehen wieder in drei Schritten vor:

Im ersten Schritt legen wir eine neue Klasse `ArtifactSizeRecorder` an. In deren Methode `perform`, die am Ende eines Builds aufgerufen wird, berechnen wir die Größe der Artefakte und speichern diese in einem neuen Action-Objekt. Zusätzlich geben wir diese Information über einen Logger in die Konsole aus.

*Erster Schritt: Anlegen von
ArtifactSizeRecorder und
ArtifactSizeAction*

```
package hudson.plugins.artifactory;
[...]
public class ArtifactSizeRecorder extends Recorder {
    @DataBoundConstructor
    public ArtifactSizeRecorder() {
        // Required by Stapler framework.
    }
}
```

Listing 9-11
`ArtifactSizeRecorder.java`

```

public boolean perform(AbstractBuild build, Launcher launcher,
                      BuildListener listener) throws InterruptedException,
                      IOException {
    long totalSize = 0;
    List<Artifact> artifacts = build.getArtifacts();
    for (Artifact artifact : artifacts) {
        totalSize += artifact.getFile().length();
    }
    build.addAction(new ArtifactSizeAction(totalSize, build));
    return true;
}

public BuildStepMonitor getRequiredMonitorService() {
    // No external synchronization with concurrent builds of the
    // same project required.
    return BuildStepMonitor.NONE;
}

@Extension
public static class ArtifactSizeRecorderDescriptor extends
    BuildStepDescriptor<Publisher> {

    @Override
    public String getDisplayName() {
        return "Record Artifact Sizes";
    }

    @Override
    public boolean isApplicable(
        Class<? extends AbstractProject> jobType) {
        return true;
    }

}
}

```

Die Klasse `ArtifactSizeAction` muss die drei Methoden `getDisplayName`, `getIconFileName` und `getUrlName` implementieren. Sie legen fest, wie die neue Aktion in der Benutzeroberfläche dargestellt wird und wohin der unterlegte Link dieser Aktion führt. Wir verwenden für unsere Aktion eines der in Hudson mitgelieferten Symbole, `monitor.gif`.

Listing 9-12

ArtifactSizeAction.java

```

package hudson.plugins.artifactsize;
[...]
public class ArtifactSizeAction implements Action {

    private final long totalSize;

    private final AbstractBuild build;

    public ArtifactSizeAction(long totalSize, AbstractBuild build) {
        this.totalSize = totalSize;
    }
}

```

```

        this.build = build;
    }

    public long getTotalSize() {
        return totalSize;
    }

    public AbstractBuild getBuild() {
        return build;
    }

    public String getDisplayName() {
        return "Show Artifact Size";
    }

    public String getIconFileName() {
        return "monitor.gif";
    }

    public String getUrlName() {
        return "artifactsize";
    }
}

```

Im zweiten Schritt erweitern wir die Benutzeroberfläche um eine Detailansicht der Artefaktgrößen. Dazu legen wir eine weitere Jelly-Datei `index.jelly` an, die sich im Verzeichnis `resources/hudson/plugins/artifactsize/ArtifactSizeAction` befinden muss. Beachten Sie auch hier wieder die exakte Namensübereinstimmung der Klasse `ArtifactSizeAction` und des Verzeichnisses, in dem `index.jelly` abgelegt wird. Die Jelly-Variable `it` bezieht sich in dieser Schablone auf die Instanz der Klasse `ArtifactSizeAction`, weil sich die Jelly-Datei im gleichnamigen Verzeichnis befindet. Über die Punkt-Notation können Datenfelder dieser Instanz ausgelesen und in die HTML-Ausgabe eingefügt werden.

*Zweiter Schritt:
Benutzeroberfläche
erweitern*

```

<j:jelly xmlns:j="jelly:core" xmlns:st="jelly:stapler"
    xmlns:l="/lib/layout">
<j:set var="build" value="${it.build}" />
<l:layout
    title="${build} ${build.number} Artifact Size"
    norefresh="true">
<st:include it="${build}" page="sidepanel.jelly" />
<l:main-panel>
    <h1>Total Artifacts Size</h1>
    This build eats up <b>${it.totalSize}</b> byte(s) of
    precious disk space to store ${build.artifacts.size()}
    artifact(s).
</l:main-panel>
</l:layout>
</j:jelly>

```

Listing 9-13
index.jelly

Dritter Schritt:
Aktualisierung von
ArtifactColumnSize

Listing 9–14
Änderungen in
ArtifactColumnSize.java

```
public String getArtifactSize(Job job) {
    Run lastRun = job.getLastBuild();
    if (lastRun == null) {
        return "No build found";
    }
    ArtifactSizeAction asa = lastRun
        .getAction(ArtifactSizeAction.class);
    if (asa == null) {
        return "Not recorded";
    }
    long totalSize = asa.getTotalSize();
    if (roundedFormat) {
        return FileUtils.byteCountToDisplaySize(totalSize);
    } else {
        return totalSize + " byte(s)";
    }
}
```

Abbildung 9–14 zeigt die Ansicht eines Builds mit unserer neuen ArtifactSizeAction (obere Hälfte): Beachten Sie hier den neuen Link *Show Artifact Size* in der linken Seitenleiste. Dieser führt zur Detailansicht der Aktion (untere Hälfte).

Abb. 9–14
Anzeige der neuen Aktion
Show Artifact Size



9.3.4 Erweiterungspunkt »CLICommand«

Um auch Systemadministratoren zu beglücken, entwickeln wir nun eine kommandozeilenbasierte Version unseres Plugins. Dadurch können Artefaktgrößen auch per Shell-Skript abgefragt und ausgewertet

werden. Hudson stellt uns dazu bereits eine Basisklasse `CLICommand` zur Verfügung, von der wir lediglich ableiten und ein paar Methoden überschreiben müssen:

```
package hudson.plugins.artifsize;  
[...]  
@Extension  
public class ArtifSizeCommand extends CLICommand {  
  
    @Override  
    public String getShortDescription() {  
        return "Lists artifact size of the last build.";  
    }  
  
    @Override  
    protected void printUsageSummary(PrintStream stderr) {  
        stderr.print("Lists artifact size of the last build ");  
        stderr.println("of a given job.");  
        stderr.println("If no job is given, all jobs are listed.");  
        stderr.println("Valid parameters are:");  
    }  
  
    @Argument(metaVar = "JOB", usage = "Job name")  
    public AbstractProject<?, ?> job;  
  
    @Override  
    protected int run() throws Exception {  
        List<AbstractProject> jobsToList  
            = new ArrayList<AbstractProject>();  
        if (job != null) {  
            jobsToList.add(job);  
        } else {  
            jobsToList.addAll(Hudson.getInstance().getAllItems(  
                AbstractProject.class));  
        }  
  
        for (AbstractProject job : jobsToList) {  
            stdout.printf("%-20s", job.getName());  
            Run lastRun = job.getLastBuild();  
            if (lastRun == null) {  
                stdout.println("No build found");  
            } else {  
                ArtifSizeAction asa = lastRun  
                    .getAction(ArtifSizeAction.class);  
                if (asa == null) {  
                    stdout.println("Built prior to installation of "+  
                        "ArtifSize");  
                } else {  
                    long totalSize = asa.getTotalSize();  
                    stdout.printf("%14d byte(s)", totalSize);  
                    stdout.printf("%3d artifacts(s)\n", lastRun
```

```
        .getArtifacts().size());
    }
}
}
return 0;
}
}
```

Die Methoden `getShortDescription` bzw. `printUsageSummary` verwendet der Kommandozeilen-Client, um die Anwendung des Kommandos in der Konsole zu erklären.

Die Methode `run` enthält die eigentliche Funktionalität des Kommandos, also die Ausgabe der Artefaktgrößen. Werden keine weiteren Parameter übergeben, werden die Artefaktgrößen aller angelegten Projekte ausgegeben. Zusätzlich unterstützt unser Kommando die optionale Angabe eines Jobs, dessen Artefaktgröße ausgegeben werden soll. Beachten Sie dazu das Feld `job`, das mit der Annotation `@Argument` markiert ist und daher automatisch mit einem eventuell angegebenen Parameter initialisiert wird.

*Testlauf eines neuen
CLI-Kommandos*

Wie kann unser neuer Befehl in der Kommandozeile aufgerufen werden? Zunächst starten Sie eine Hudson-Testinstanz mit `hpi:run`. Dann laden Sie aus dieser Instanz den Kommandozeilen-Client über die URL `http://localhost:8080/jnlpJars/hudson-cli.jar` lokal herunter. In der Kommandozeile starten Sie diesen Client mit dem Schlüsselwort unseres neuen Befehls. Dieses leitet Hudson aus dem Namen der Java-Klasse ab: An Übergängen von Klein- zu Großbuchstaben wird ein Trennstrich eingefügt, dann alle Buchstaben kleingeschrieben und schließlich die Endung `Command` abgeschnitten: Aus `ArtifactSizeCommand` wird so das Schlüsselwort `artifact-size`:

```
$ java -jar hudson-cli.jar -s http://ng-luna:8080/ artifact-size meinProjekt1
meinProjekt1          2367668 byte(s) 4 artifacts(s)
```

Der Jobname ist optional. Fehlt er, so werden einfach die Informationen für alle Jobs angezeigt:

```
$ java -jar hudson-cli.jar -s http://localhost:8080/ artifact-size
freestyle              8128790 byte(s) 3 artifacts(s)
meinProjekt1           2367668 byte(s) 4 artifacts(s)
glotr2                 Built prior to installation of ArtifactSize
greetr                 0 byte(s) 0 artifacts(s)
greetr2                766 byte(s) 1 artifacts(s)
prognosr               Built prior to installation of ArtifactSize
```

Beachten Sie, dass unser neues CLI-Kommando auf dem Hudson-*Server* installiert und ausgeführt wird und nicht Teil des heruntergeladenen Kommandozeilen-Clients ist. Dies erklärt auch die geringe und konstante Größe von `hudson-cli.jar` (ca. 230kB) – unabhängig davon, wie viele CLI-Kommandos zur Verfügung stehen. In diesem Archiv sind lediglich Funktionen zum Aufruf von Methoden auf dem Server und zur Parameterübertragung enthalten. Die Ausführung geschieht dabei immer serverseitig. Da die Schnittstelle zwischen Kommandozeilen-Client und Hudson-Server sich nur selten ändert, kann man durchaus auch mit einem betagten Kommandozeilen-Client eine aktuelle Hudson-Instanz ansprechen.

CLI-Kommandos befinden sich auf dem Server, nicht im Client.

9.3.5 Bereitstellen der Artefaktgröße per XML-API

Neben der Kommandozeile bietet Hudson über die Remote-API einen weiteren Weg, Daten bereitzustellen – etwa im XML-Format. Dies erleichtert die Integration mit Drittssystemen, da technisch gesehen lediglich eine URL abgerufen und deren Inhalt als XML-Datei ausgewertet werden muss. Um so die Artefaktgröße eines Builds nach außen bereitzustellen, müssen Sie lediglich die Klasse `ArtifactSizeAction` mit der Annotation `@ExportedBean` markieren. Mit der zweiten Annotation `@Exported` kennzeichnen Sie anschließend diejenigen Datenfelder, die von außen sichtbar sein sollen:

```
[...]  
@ExportedBean  
public class ArtifactSizeAction implements Action {  
    @Exported(visibility=2)  
    public long getTotalSize() {  
        return totalSize;  
    }  
    [...]  
}
```

Listing 9-16
*Änderungen in
ArtifactSizeAction.java*

Welche Rolle spielt dabei der Parameter `visibility`? Im Beispiel in Abbildung 9-15 werden Informationen über einen Build abgerufen. Da ein Build eine baumartige Datenstruktur darstellt, die sehr umfangreich sein kann, würde die entsprechende XML-Datei entsprechend groß ausfallen. In der Praxis benötigt man aber oft nicht die komplette Baumstruktur, sondern nur ausgewählte Informationen von besonderer Wichtigkeit – die Sie über den Parameter `visibility` festlegen können. Was bedeutet dabei ein Wert von 2? Im Beispiel des Builds liefert das Remote-API alle exportierten Datenfelder des Build-Objekts selbst, die exportierten Datenfelder seiner Kindobjekte mit einer visi-

bility von mindestens 2, seiner Enkelobjekte mit einer visibility von mindestens 3 usw. zurück. Je wichtiger also das Feld, desto höher der Wert für visibility, desto eher ist es in einer Antwort des Remote-API enthalten.

Das war's schon – den Rest übernimmt Hudson für Sie automatisch. Dem Abruf der Artefaktgrößen per URL steht nichts mehr im Wege (Abb. 9–15).

Abb. 9–15

Abruf der Artefaktgröße
über die XML-API

```

<freeStyleBuild>
  <action>
    <cause>
      <shortDescription>Gestartet durch Benutzer anonymous</shortDescription>
      <userName>anonymous</userName>
    </cause>
  </action>
  <action>
    <totalSize>8128790</totalSize>
  </action>
  <artifact>
    <displayPath>distribution.zip</displayPath>
    <fileName>distribution.zip</fileName>
    <relativePath>distribution.zip</relativePath>
  </artifact>
  <building>false</building>
  <duration>1750</duration>
  <fullDisplayName>freestyle #10</fullDisplayName>
  <id>2010-05-16_22-36-12-fd</id>
  <keepLog>false</keepLog>
  <number>10</number>
  <result>SUCCESS</result>
  <timestamp>1274042172000</timestamp>

```

9.3.6 Online-Hilfetexte

In den vorangegangenen Abschnitten haben wir uns ausschließlich mit der *Funktionalität* unseres neuen Plugins beschäftigt. Für Anwender unseres Plugins sind jedoch auch Aspekte wie eine einfach zu erreichende Online-Hilfe von Interesse.

Zuordnung von
Hilfetexten per
Namenskonvention

Sie haben sicherlich bereits an vielen Stellen der Hudson-Benutzeroberfläche die Hilfesymbole gesehen (blauer Kreis mit weißem Fragezeichen), die bei Anklicken eine kontextsensitive Hilfestellung anzeigen. Um unser Plugin mit solchen Texten auszustatten, müssen wir sie lediglich in HTML verfassen und als Datei `help.html` in eine Verzeichnishierarchie ablegen, die den voll qualifizierten Namen der zugehörigen Klasse nachstellt (also dort, wo meistens schon die passenden Jelly-Dateien liegen). Abbildung 9–16 zeigt dies beispielhaft für unser Plugin: Hudson greift diese Dateien automatisch auf und bindet sie an der passenden Stelle in die Benutzeroberfläche ein. Die Kurzzusam-

menfassung lautet also: Einfach eine HTML-Datei `help.html` an der richtigen Stelle ablegen, und Hudson macht den Rest.

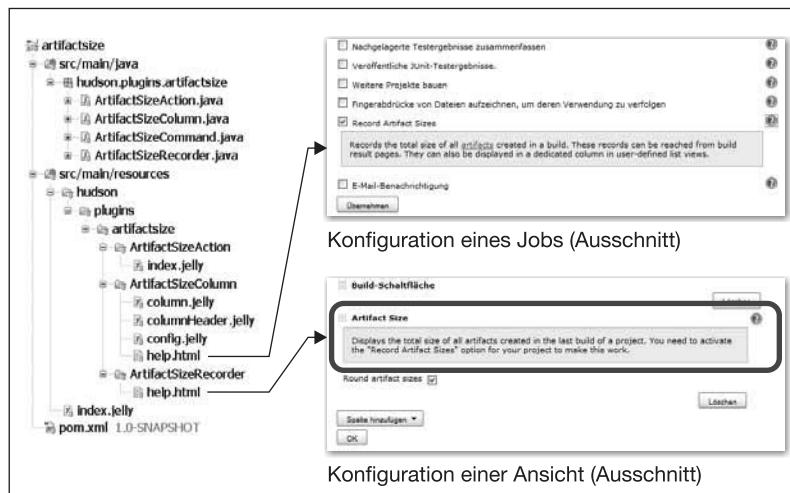


Abb. 9-16
Automatische Einbindung
von Hilfertexten

Beachten Sie, dass die HTML-Dateien auch Formatierungen und Links enthalten können. Durch relative Adressierung können Sie zudem hilfreiche Querverweise einbauen. Im Listing 9–17 wird etwa auf die Artefakte des letzten Builds verwiesen (siehe Hervorhebung).

```
<div>
  Records the total size of all <a href="lastBuild/artifact/">
  artifacts</a> created in a build. These records can be reached
  from build result pages. They can also be displayed in a
  dedicated column in user-defined list views.
</div>
```

Listing 9-17
`help.html`: Hilfestellung für
`ArtifactSizeRecorder.java`

9.3.7 Lokalisierung und Internationalisierung

Unter Lokalisierung wird die Anpassung einer Software an eine bestimmte Kultur verstanden. Der offensichtlichste Schritt dabei ist die Übersetzung aller Texte in eine andere Sprache. Eine vollständige Lokalisierung berücksichtigt aber auch etwa die Notation von Zahlen oder Kalenderdaten oder die Schreibrichtung (z.B. links-nach-rechts bzw. rechts-nach-links). Alle kulturspezifischen Einstellungen werden in Java als *Locale* bezeichnet und typischerweise durch eine Buchstabenkombination identifiziert, z.B. `fr` für Französisch, `de` für Deutsch, `de_CH` für Deutsch, wie es in der Schweiz gesprochen wird, usw.

Lokalisierung

Eine lokalisierbare Software muss an allen Stellen, an denen Locale-abhängige Daten ausgegeben werden, entsprechende »Wei-

Internationalisierung

chen« beinhalten, etwa um Textausgaben in der gerade eingestellten Landessprache auszugeben. Man spricht dann von einer internationalisierten Software.

Dies betrifft auch Sie als Plugin-Entwickler: Soll Ihr Plugin in verschiedene Landessprachen lokalisierbar sein, müssen Sie es zuvor internationalisieren. Wie das geht, lernen Sie in den folgenden Unterabschnitten. Es gibt dabei drei Fälle, die wir für eine vollständige Internationalisierung und anschließende Lokalisierung berücksichtigen müssen:

- Textausgaben der Online-Hilfe
- Ausgaben mithilfe von Jelly-Schablonen
- Ausgaben, die in Java-Klassen erzeugt werden

Texte der Online-Hilfe

Dieser Fall ist der einfachste: Sie legen einfach neben die vorhandenen Hilfedateien `help.html` weitere Dateien nach der Namenskonvention `help_<locale>.html` ab, also `help_de.html` für eine deutsche Lokalisierung. Hudson verwendet dann automatisch eine möglichst passende Variante zur Anzeige in der Benutzeroberfläche.

*HTML-Dateien müssen
UTF-8 verwenden.*

Beachten Sie, dass diese HTML-Dateien in UTF-8 kodiert sein müssen – sonst werden Sonderzeichen und Umlaute nicht korrekt in Hudson darstellt. Unter Eclipse können Sie die Kodierung einer Datei einfach überprüfen, indem Sie den Eigenschaften-Dialog der Datei öffnen. Auf der Unterseite *Resource* wird Ihnen die momentan verwendete Kodierung im Abschnitt *Text file encoding* angezeigt. Hier sollte *UTF-8* stehen, also *nicht* etwa *Cp1252* oder *ISO-8859-1*.

Ausgaben mithilfe von Jelly-Schablonen

*Texte in Jelly-Ausdrücke
umwandeln*

Etwas trickreicher wird in den Jelly-Schablonen gearbeitet. Zunächst markieren Sie lokalisierbare Texte als Jelly-Ausdrücke, statt `Artifact Size` schreiben Sie nun beispielsweise `${%Artifact Size}`. Dadurch passieren drei Dinge:

- Hudson (genauer: das eingesetzte Stapler-Framework) erkennt diesen Text als lokalisierbare Passage und fügt zur Laufzeit an dieser Stelle die Übersetzung in der eingestellten Sprache ein.
- Der englische Originaltext wird als Schlüssel zum Nachschlagen der entsprechenden Übersetzung in den lokalisierten Sprachdateien verwendet.
- Der englische Originaltext dient als Vorgabewert, sollte sich keine explizite Übersetzung in einer bestimmten Locale finden.

Wie Sie in Listing 9–18 sehen, können Sie in Jelly-Schablonen auch Parameter verwenden, um Platzhalter in Textbausteinen durch Informationen zu ersetzen, die zur Laufzeit berechnet werden. In unserem Beispiel sind dies die Gesamtgröße der Artefakte und deren Anzahl.

```
<j:jelly xmlns:j="jelly:core" xmlns:st="jelly:stapler"
  xmlns:l="/lib/layout">
  <j:set var="build" value="${it.build}" />
  <l:layout
    title="${build} #${build.number} ${%Artifacts Size}"
    norefresh="true">
    <st:include it="${build}" page="sidepanel.jelly" />
    <l:main-panel>
      <h1>${%Total Artifacts Size}</h1>
      ${%message(it.totalSize, build.artifacts.size())}
    </l:main-panel>
  </l:layout>
</j:jelly>
```

Parameter für
Textbausteine

Listing 9–18
Internationalisierte
Variante von index.jelly für
ArtifactSizeAction

Anschließend legen Sie pro Locale jeweils eine Datei mit den Übersetzungen an. Diese ist über den Namen mit der zugehörigen Jelly-Schablone verknüpft: Die deutsche Lokalisierung der Schablone index.jelly liegt demnach in index_de.properties. Texte für die (meist englische) Originalversion liegen in index.properties.

Listing 9–19 bzw. Listing 9–20 zeigen die englische bzw. deutsche Lokalisierung der ArtifactSizeAction. Beachten Sie, dass in der englischen Datei nur ein Schlüssel enthalten ist, in der deutschen hingegen drei. Das liegt daran, dass für die im Englischen fehlenden zwei Schlüssel der Vorgabewert aus der Jelly-Datei direkt verwendet werden kann.

Englische Lokalisierung
kann Vorgabewerte
nutzen.

```
message=This build eats up <b>\n  {0,choice,0#0 bytes|1#1 byte|1<{0} bytes}</b> \n  of precious disk space to store \n  {1,choice,0#0 artifacts|1#1 artifact|1<{1} artifacts}.
```

Listing 9–19
index.properties für
ArtifactSizeAction

```
Artifact\ Size=Artefakt-Größe\nTotal\ Artifacts\ Size=Artefakt-Gesamtgröße\nmessage=Dieser Build verschlingt <b>\n  {0,choice,0#0 Bytes|1#1 Byte|1<{0} Bytes}</b> \n  wertvollen Festplattenplatzes, um \n  {1,choice,0#0 Artefakte|1#1 Artefakt|1<{1} Artefakte} \n  zu speichern.
```

Listing 9–20
index_de.properties für
ArtifactSizeAction

Durch Fallunterscheidungen in den *.properties-Dateien (fett hervorgehoben) lassen sich unschöne Klammerdarstellungen wie *12345678 Byte(s) in 123 Datei(en)* vermeiden, indem je nach Anzahl der ermittelten Bytes entweder *0 Bytes*, *1 Byte* oder *x Bytes* ausgegeben wird. Mehr zum hier verwendeten choice-Konstrukt finden Sie in der Java-API-Dokumentation der Klasse `java.text.MessageFormat`.

Fallunterscheidungen

Häufige Stolperfallen

Sie können beim Erstellen der *.properties-Dateien von vornherein drei häufige Stolperfallen vermeiden:

- Diese Dateien müssen in ISO-8859-1 kodiert sein (also hier *nicht* in UTF-8). Dies ist durch das Java Development Kit vorgegeben. Benötigen Sie Zeichen, die nicht in ISO-8859-1 enthalten sind, können Sie das Programm native2ascii verwenden (im Java Development Kit enthalten), um Unicode-Texte in ISO-8895-1 zu kodieren.
- Lange Texte können Sie über mehrere Zeichen umbrechen, indem Sie die umgebrochene Zeile mit einem umgekehrten Schrägstrich \ beenden. Dies wird gerne vergessen, wenn man kurz zuvor in HTML-Dateien gearbeitet hat – dort wird kein Umbruchzeichen benötigt.
- Leerzeichen in den Schlüsseln müssen ebenfalls mit einem umgekehrten Schrägstrich \ gekennzeichnet sein. Erscheint im Browser immer nur das englische Original statt der Übersetzung, prüfen Sie zunächst die Leerzeichen in Schlüsseln auf fehlende Schrägstriche als erste Gegenmaßnahme.

Tipp: Schnelles Umschalten der Lokalisierung im Browser

Hudson wählt die anzuseigende Lokalisierung in Abhängigkeit der Einstellungen Ihres Webbrowsers aus. Wenn Sie zu Testzwecken häufig zwischen unterschiedlichen Lokalisierungen wechseln, es aber leid sind, sich jedes Mal durch die Einstellungen Ihres Browsers zu hangeln, können Sie unter Firefox auf das sehr nützliche, kostenlose Firefox-Add-on »Quick Locale Switcher« zurückgreifen (<https://addons.mozilla.org/en-US/firefox/addon/1333/>). Es installiert in der Statusleiste ein kleines Flaggensymbol, über das Sie mit lediglich zwei Klicks die bevorzugte Lokalisierung Ihres Browsers umschalten könnten.

Ausgaben, die in Java-Klassen erzeugt werden

In manchen Fällen ist das Zusammenstückeln lokalisierter Texte über Jelly-Ausdrücke zu mühselig oder sogar unmöglich – etwa weil die Ausgabe nicht über Jelly erfolgt, sondern vielleicht direkt in eine Datei geschrieben wird.

Localizer

In diesen Situationen setzt Hudson auf den Localizer (<https://localizer.dev.java.net>), ein weiteres Framework aus der Tastatur von – Sie ahnen es vermutlich bereits – Hudson-Schöpfer Kohsuke Kawaguchi. Localizer arbeitet in drei Schritten:

1. Die lokalisierten Texte werden einer Datei `Message.properties` angelegt.

2. Aus der Datei `Message.properties` wird Java-Quelltext generiert, mit dem auf die enthaltenen Texte zugegriffen werden kann. Pro Schlüssel wird dabei eine `public`-sichtbare Methode erzeugt. Diese Codegenerierung erledigt Maven übrigens automatisch für Sie, wenn Sie das Plugin-Projekt kompilieren.
3. Aus anderen Java-Klassen kann nun auf die Methoden der soeben generierten Quelltexte zugegriffen werden: Deren Rückgabewert enthält dann die lokalisierten Texte in der gewünschten Sprache.

Am besten lässt sich dieses zunächst ungewohnte Vorgehen an einem Beispiel verstehen: Wir verwenden den Localizer, um die Beschriftung *Show Artifact Size* des Symbols am linken Seitenrand zu internationalisieren und anschließend auf Englisch und Deutsch zu lokalisieren:

Zunächst legen wir also die benötigten Schlüssel in den Dateien `Message.properties` (Listing 9–21) und `Message_de.properties` (Listing 9–22) im Verzeichnis `src/main/resources/hudson/plugins/artifsize` an:

Lokalisierungen anlegen

`ArtifactSizeAction.ShowArtifactSize=Show Artifact Size`

Listing 9–21

Messages.properties

`ArtifactSizeAction.ShowArtifactSize=Artefakt-Größen zeigen`

Listing 9–22

Messages_de.properties

Danach rufen wir Maven mit `mvn generate-sources` auf. Dadurch erzeugt der von Maven aufgerufene Localizer eine neue Java-Datei `Message.java`. Diese befindet sich im Verzeichnis `target/generated-sources/localizer/hudson/plugins/artifsize`.

Um die vom Localizer generierten Quelltexte von Eclipse berücksichtigen zu lassen, nehmen wir das Verzeichnis `target/generated-sources/localizer` in den Build-Pfad auf. Dazu klicken Sie im *Package Explorer* mit der rechten Maustaste auf das Verzeichnis `localizer` und wählen im Kontextmenü den Befehl *Build Path → Use As Source Folder*.

Generierte Quelltexte in Eclipse berücksichtigen

Schließlich ersetzen wir »hart verdrahtete« Texte in unseren Java-Klassen durch Methodenaufrufe. In `ArtifactSizeAction.java` ändern wir dann

```
public String getDisplayName() {
    return "Show Artifact Size";
}
```

Texte durch Methodenaufrufe ersetzen

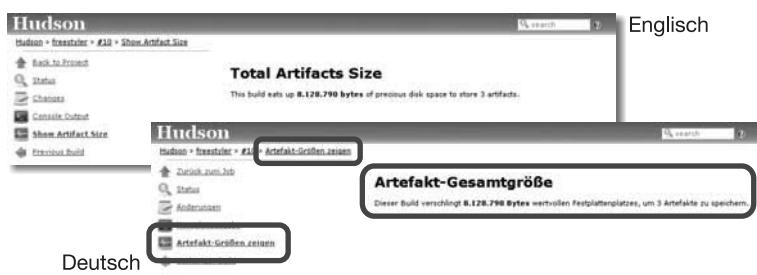
durch Einfügen eines Methodenaufrufs zu:

```
public String getDisplayName() {
    return Messages.ArtifactSizeAction_ShowArtifactSize();
}
```

Abschließend starten wir unser Plugin nochmals in einer Hudson-Testinstanz mit `mvn hpi:run`. Die Umschaltung zwischen englischer und deutscher Locale im Browser liefert dann das in Abbildung 9–17 gezeigte Ergebnis.

Abb. 9–17

Vollständige Lokalisierung
des Plugins »Artifact Size«



9.4 Zusammenfassung

In diesem Kapitel haben wir zunächst mithilfe der Maven-Werkzeuge des Hudson-Projekts einen kleinen HelloWorld-BUILDER erzeugt, getestet und zur Distribution als HPI-Datei verpackt. Nach einer Einführung in Hudsons Plugin-Konzept sind wir auf die über 70 Erweiterungspunkte eingegangen, die eine punktgenaue Anpassung Hudsons auf eigene Bedürfnisse erlauben. Zum Abschluss haben wir – Schritt für Schritt – ein Beispiel-Plugin entwickelt, das gleich mehrere Erweiterungspunkte verwendet (`ListViewColumn`, `Recorder`, `Action`, `CLICommand`), seine Daten per XML-API nach außen zur Verfügung stellen kann, eine Online-Hilfe bereitstellt und sowohl Englisch als auch Deutsch spricht.

Dieses Kapitel beschließt den zweiten Teil des Buches, in dem wir uns ganz konkret mit Hudson befasst haben. Im dritten und letzten Teil des Buches werden wir auf Fragestellungen eingehen, die typischerweise bei der Einführung von Continuous Integration in Ihrer Organisation auftauchen werden. Im nächsten Kapitel werden wir dazu mit einer Betrachtung des Aufwands beginnen, den eine CI-Einführung erfordert.

10 Aufwand einer CI-Einführung

In diesem Kapitel betrachten wir die notwendigen Investitionen für eine CI-Einführung in den eigenen Softwareentwicklungsprozess. Um eine wichtige Erkenntnis gleich vorwegzunehmen: Wir werden feststellen, dass es mit der Installation und Konfiguration eines CI-Servers – dem inhaltlichen Schwerpunkt der vorangegangenen Kapitel – nicht getan ist.

Die notwendigen Posten sind jeweils mit einer groben Zeitschätzung versehen, die allerdings nur eine Größenordnung andeuten kann. Der konkrete Gesamtaufwand einer CI-Einführung variiert naturgemäß sehr stark, je nachdem im welchem Zustand sich der vorhandene Build-Prozess befindet und um welchen Quelltextumfang es sich dabei handelt.

10.1 Erstaufwand für Vollautomatisierung des Builds

Die wichtigste Grundlage eines funktionierenden CI-Systems ist ein vollautomatisch ablaufender Build-Prozess, der keine menschlichen Eingriffe erfordert darf.

Glücklicherweise sind die Tage der proprietären Build-Werkzeuge integrierter Entwicklungsumgebungen (IDEs) gezählt. Hersteller- sowie plattformübergreifende Werkzeuge wie Ant oder Maven haben sich durchgesetzt. Trotzdem sind viele Arbeitsgruppen weit entfernt von einem »Ein-Knopf-Build«: Stattdessen ist in der Kommandozeile eine undokumentierte Sequenz aus Ant-, Shell- und Perl-Skripten in der richtigen Reihenfolge zu starten. Dass dabei auch noch ganz bestimmte Umgebungsvariablen gesetzt sein müssen und das ganze Verfahren sowieso nur auf ausgewählten Rechnern funktioniert, fällt hier in das sprichwörtliche Build-»Druidenwissen«.

Ziel: Der »Ein-Knopf-Build«

Vollautomatisierung stellt oft die größte Herausforderung während einer CI-Einführung dar.

Zwang zu Analyse und Dokumentation ist ein Vorteil.

Fehlerfreies Komplizieren reicht nicht aus.

Auch Tests müssen automatisiert sein.

Nachrüsten von Unit-Tests oft schwierig

Bevor Sie also auch nur an die Installation eines CI-Servers denken sollten, müssen Sie zunächst die Vollautomatisierung Ihres Builds angehen. Der Aufwand hierfür kann – je nach Ausgangslage und Unternehmensgröße – leicht mehrere Wochen betragen. Dieser Schritt wird als »vorbereitende Maßnahme« gerne unterschätzt. Nicht selten verursacht er jedoch den Löwenanteil des Einführungsaufwandes. Ein einfaches Verketten der bestehenden Build-Schritte ist nämlich in den wenigsten Fällen ausreichend, da beispielsweise Quelltexte manuell in geeigneten Revisionen ausgecheckt werden müssen, Drittbibliotheken händisch zusammengestellt werden oder die abschließende Produktivsetzung nur in der Umgebung eines speziellen Benutzerkontos gelingt.

Das Gebot der Vollautomatisierung zwingt daher zunächst zur genauen Analyse und Dokumentation des momentanen Build-Prozesses. Erst dann kann die eigentliche Automatisierung vorgenommen werden. Was sich wie ein Nachteil anhört, ist jedoch ein wichtiger Vorteil: Die Offenlegung von ansonsten unzugänglichem Druidenwissen über die Abläufe und Abhängigkeiten des Build-Prozesses.

10.2 Erstaufwand für Erstellung von fehlenden Tests

In machen Teams gilt bereits die Übersetzung des Quelltextes ohne Compilerwarnungen als ein wichtiges Qualitätsindiz. Dies ist häufig in C/C++-Projekten zu beobachten, in denen mit geringer IDE-Unterstützung gearbeitet wird. Doch auf dem Weg zur fehlerfreien Software sind fehlende Compilerwarnungen lediglich eine notwendige, längst aber nicht hinreichende Bedingung.

Angemessene Qualitätssicherung setzt hingegen einen umfangreichen Testplan auf mehreren Ebenen voraus (Unit-, Integrations- und Systemtest), der zudem vollautomatisch durchführbar sein muss. In machen Fällen existieren nur interaktiv ausgeführte Tests auf Systemebene (»Ich starte den Webserver und klicke dann ein bisschen in der Anwendung herum«). Hier muss dann nachträglich ein »Fangnetz« aus Tests aufgebaut werden. Für Integrations- und Systemtests ist dies in der Regel einfacher möglich als für Unit-Tests, weil erstere die inneren Abläufe der zu testenden Software als Black-Box betrachten und ignorieren können. Für das Auswählen und Aufsetzen eines Testframeworks sollten Sie wenige Wochen einplanen, ebenso für das Erstellen der Testfälle – je nach Umfang der zu testenden Software.

Das Nachrüsten von Unit-Tests erweist sich in der Praxis oft als schwierig, wenn der zu testende Code nicht bereits sehr modular und testfreundlich entworfen wurde. Hier kann es aus betriebswirtschaftlichen Erwägungen sinnvoller sein, nur neuen Code mit Unit-Tests zu

prüfen und bestehenden Code mit engmaschigen Integrationstests abzusichern.

10.3 Optimieren des Build-Prozesses für CI

Beim Umstieg von Builds, die liebevoll interaktiv auf einem Arbeitsplatzrechner gebaut werden, hin zu vollautomatischen Builds auf einem CI-Server zeigen sich oftmals erstmalig Einschränkungen des bestehenden Verfahrens, z.B.

■ *fehlende Modularisierung:*

Jeder Build erfordert stets aufs Neue das vollständige Auschecken und Bauen aller verwendeten Quelltexte – selbst wenn die Änderungen nur klein und lokal begrenzt sind.

■ *fehlende Strukturierung in Build-Phasen:*

Der Build muss stets am Stück komplett »monolithisch« durchlaufen werden und verfügt über keine parallelisierbaren Abschnitte.

■ *hoher Ressourcenbedarf:*

Builds, die interaktiv auf einem Arbeitsplatzrechner gebaut werden, stehen in der Regel die vollen Ressourcen des Systems zur Verfügung. Ein zentraler CI-Server muss hingegen mehrere solcher Builds gleichzeitig bauen und deren Ergebnisse archivieren können.

Selbst wenn also bereits ein vollautomatischer Build existiert, kann dessen Entflechtung, Strukturierung und Ressourcenoptimierung nochmals einen deutlichen Aufwand von wenigen Wochen bedeuten. Das Anpassen eines Builds auf »CI-Fähigkeit« ist nicht trivial und setzt neben tiefem Verständnis der beteiligten Build-Werkzeuge auch idealerweise gute Kenntnisse des eingesetzten CI-Servers voraus. In dieser Phase kann es sehr effizient sein, die Unterstützung eines externen Spezialisten in Anspruch zu nehmen, der Erfahrung in dieser Aufgabenstellung mitbringt.

10.4 Hardwaredressourcen für ein CI-System

Startet die CI-Einführung als Feierabendprojekt eines motivierten Mitarbeiters, steht als Hardwareplattform oftmals nur ein betagter Rechner zur Verfügung. Dies scheint kein Problem zu sein, denn »prinzipiell funktioniert es ja – die Builds brauchen dann eben ein bisschen länger«.

Für ein wirklich effektives CI-System ist dies aber keine Lösung, denn kurze Build-Zeiten sind kein Luxus, sondern Voraussetzung des

Kurze Builds sind kein Luxus, sondern eine Voraussetzung.

CI-Konzepts. Der CI-Server sollte somit nicht das lahmste, sondern das eher schnellste Pferd im Server-Rack sein! Es ist unverständlich, warum viele Führungskräfte die Anschaffung eines eigenständigen CI-Servers hinauszögern: Rechner werden immer leistungsfähiger und preiswerter. Gleichzeitig wird die menschliche Arbeit immer wertvoller. Die Zeiten, in denen ein gut ausgestatteter Server dem Gegenwert mehrerer Jahresgehälter entsprach, sind längst passé. Somit lässt sich mit einer Modellrechnung leicht belegen, dass ein CI-Server sich allein durch eingesparte Arbeitszeit in wenigen Monaten amortisieren kann.

10.5 Installation des CI-Systems

*Erst automatisieren,
dann installieren.*

Obwohl die Installation und Konfiguration eines CI-Servers die offensichtlichen Schritte bei der CI-Einführung darstellen, dürften hier erfreulicherweise nur überschaubare Aufwände anfallen. Zum einen können Lizenzkosten vermieden werden, wenn freie Open-Source-Produkte zum Einsatz kommen. Zum anderen geht bei benutzerfreundlichen Produkten die Konfiguration schnell von der Hand – aber nur, wenn die Hausaufgaben bei der Vollautomatisierung und CI-Optimierung des Builds gut erledigt wurden.

Viele CI-Einführungen *beginnen* jedoch leider mit diesem Schritt, ohne die vorangegangenen Aspekte beachtet zu haben. Nicht selten artet dieser Ansatz dann zu einer wochenlangen Odyssee in den Konfigurationsoptionen eines CI-Servers aus. Mit überflüssigen Saltos und Kunstgriffen wird dabei versucht, den bestehenden Build-Prozess mit all seinen gewachsenen Ecken, Kanten und Ungereimtheiten in einen CI-Server zu pressen – anstatt die Chance zur Vereinfachung und Vereinheitlichung zu nutzen: »Was? Der CI-Server hat da keinen Knopf, um Builds auf Rechnern mit IP-Adressen zu starten, deren Quersumme eine Primzahl ist? Aber das ist doch ein *Standardfall!*« – Hören Sie so einen Seufzer des Unverständnisses mehrmals am Tag, sind Sie mit hoher Wahrscheinlichkeit Zeuge einer solchen Odyssee.

10.6 Administration des CI-Systems

*Vorhandene Strukturen
wiederverwenden*

Auch ein CI-Server benötigt von Zeit zu Zeit die professionellen Streicheleinheiten eines Administrators. Sie können jedoch die anfallenden Aufwände reduzieren, indem Sie die in Ihrem Umfeld etablierten Prozesse auch für den CI-Server übernehmen. Dies betrifft technische Aspekte, z.B. Installation und Aktualisierung des Servers über den Paketmanager der eingesetzten Linux-Distribution, als auch organisa-

torische, z.B. die Integration in bestehende Alarmierungsketten per SMS bei Serverausfall.

10.7 Schulung

Um das volle Potenzial eines CI-Systems zu nutzen, sollten Sie die Schulung der beteiligten Personenkreise von Anfang an einplanen. In der Praxis hat es sich bewährt, Schulungen an drei unterschiedliche Zielgruppen zu richten:

Schulungen erschließen das volle CI-Potenzial.

■ **Systemadministratoren:**

In diesen Schulungen sollten der Betrieb eines CI-Servers aus technischer Sicht behandelt werden, insbesondere Installation, Konfiguration, Überwachung, Sicherung und Aktualisierung des CI-Servers. Je nach Komplexität des eingesetzten Produktes können hier 1–2 Schulungstage ausreichend sein.

■ **Hauptbenutzer:**

In diesen Schulungen sollten der Betrieb eines CI-Servers aus fachlicher Sicht behandelt werden, insbesondere das Anlegen, Konfigurieren und Verwalten von Projekten. Außerdem sollten Hauptbenutzer einen Überblick über die verfügbaren Erweiterungsmöglichkeiten (z.B. Plugins) eines Produktes haben, um Lösungen für neuartige Anforderungen aus ihren Arbeitsgruppen vorschlagen zu können. Hauptbenutzer können auch als Multiplikatoren und Ansprechpartner für andere Benutzer fungieren. Je nach Komplexität des eingesetzten Produktes können hier 2–4 Schulungstage ausreichend sein. Es kann sinnvoll sein, die CI-Server-Schulung in den größeren Kontext einer Fortbildung einzubetten, in der die vollständige Build-Werkzeugkette thematisiert wird (Versionsmanagementsysteme, Maven, Repository-Manager, Issue-Tracker, automatisiertes Deployment usw.).

■ **Benutzer:**

In diesen Schulungen sollte die tägliche Benutzung des CI-Systems im Vordergrund stehen, etwa das Abrufen von Berichten oder die spontane Analyse von Build-Ergebnissen. Als Benutzer kommen übrigens nicht nur Entwickler in Frage, sondern beispielsweise auch Tester, die aus einem CI-System zu testende Artefakte entnehmen. Eine Schulungsdauer von bis zu einem Tag ist hier ausreichend.

10.8 Zusammenfassung

Die wichtigste Botschaft aus diesem kurzen Kapitel lautet: Die Installation und Konfiguration eines CI-Servers sollte *nicht* der erste Schritt einer CI-Einführung sein. Zunächst müssen die Hausaufgaben in Form der Vollautomatisierung und CI-Optimierung des Builds erledigt werden. In den meisten Fällen sollte das Gesamtbudget einer CI-Einführung daher für diese »vorbereitenden Maßnahmen« den größten Posten vorsehen.

Nachdem wir uns in diesem Kapitel mit Aufwänden einer CI-Einführung beschäftigt haben, lernen wir im nächsten Kapitel praktische Tipps zur organisatorischen Umsetzung einer solchen Einführung kennen.

11 Tipps zur Einführung von CI

Die vorausgegangenen Kapitel beschäftigten sich hauptsächlich mit den technischen Aspekten einer CI-Einführung. Ob diese jedoch auch wirklich gelingt und die erhofften Früchte trägt, hängt nicht allein von der Technik ab: Software wird immer noch (größtenteils) von Menschen entwickelt. In diesem Kapitel finden Sie daher Tipps, welche die »menschliche Komponente« einer CI-Einführung unterstützen.

11.1 Mit einem Pilotprojekt starten

In welchem Maßstab sollte eine CI-Einführung erfolgen? Ist es besser, gleich die »große, einheitliche, richtige« Lösung umzusetzen, also alle Projekte eines Unternehmens oder einer Arbeitsgruppe auf einmal der Überwachung eines CI-Servers anheimzustellen – oder ist eher ein schrittweises Vorgehen Projekt für Projekt angezeigt? In der Praxis hat sich der zweite Ansatz als wesentlich besser umsetzbar erwiesen. Zum einen kann man sich auf der Suche nach dem »großen Wurf«, der alle gewachsenen Ecken und Kanten der bestehenden Abläufe berücksichtigt, leicht in endlosen Diskussionen verzetteln. Zum anderen dauert es schlichtweg zu lange, bis der erste konkrete Nutzen in den Augen der Anwender sichtbar wird. Bis dahin kann der anfängliche Elan einer CI-Initiative längst verpufft sein (»Diese CI-Geschichte dauert ja *noch* länger als die Erstellung unseres neuen Webauftritts ...«).

Kleiner bedeutet hier schneller: Nicht selten haben »Guerilla-CI-Server«, die durch persönliche Initiative eines Mitarbeiters entstanden sind, als Demonstrationsobjekte den Weg für umfassendere Installationen freigemacht.

*Master-Plan oder
Salami-Taktik?*

Guerilla-CI-Server

11.2 Build-Zeiten kurz halten

Wie mehrfach hervorgehoben, sind kurze Build-Zeiten eine wichtige Säule des CI-Konzepts. Beobachten Sie mit einem wachen Auge, ob zeitlich ausufernde Builds bereits zu Veränderungen in der Arbeitskultur geführt haben.

*Werden Commit und
CI-Build als Einheit
gesehen?*

Ein Indikator dabei ist, ob ein Entwickler seinen Commit ins Versionsmanagementsystem mental noch mit der anschließenden Überprüfung im CI-Server verbindet (»Bin gespannt, was gleich aus *meinem* Build rauskommt...«) – oder diese als voneinander entkoppelt wahrnimmt (»einchecken, Jacke anziehen, heimgehen«). In diesen Fällen sollten Sie gegensteuern, indem Sie Maßnahmen wie Aufstockung der Hardware und Build-Parallelisierung prüfen.

11.3 Codeanalyse: Sanft starten, konstant steigern

*Automatische Prüfungen
ja – aber welche?*

Im Verlauf des Buches wurden bereits mehrfach automatisierte Codeanalysen angesprochen, die Programme auf mögliche Schwachstellen abklopfen können. Obwohl die Sinnhaftigkeit solcher Werkzeuge von den meisten Entwicklern ausdrücklich bejaht wird, gibt es deutlich unterschiedliche Ansichten darüber, wie streng diese Prüfungen sein dürfen. Da sich die meisten Werkzeuge über Profile konfigurieren lassen, hat es sich als hilfreich erwiesen, zunächst mit sehr »gutmütigen« Einstellungen zu starten, gleichzeitig aber eine »Null-Toleranz-Strategie« hinsichtlich gemeldeter Warnungen zu verfolgen. Es gibt also nur wenige Regeln zu beachten, diese müssen aber zu 100% eingehalten werden.

*Zu viel Nachbesserung auf
einmal demotiviert.*

Dies kann durchaus Nacharbeiten an bestehenden Quelltexten bedeuten, etwa die nachträgliche Ergänzung von Dokumentation in bereits funktional vollständigen Programmen. Unbesehen die umfangreichen mitgelieferten Profile der entsprechenden Werkzeuge zu verwenden (»Das sind doch die Sun-Konventionen – die können ja nicht verkehrt sein!«), kann hier stark demotivieren: Die wenigsten Entwickler wollen sich – parallel zu ihrem Tagesgeschäft – um 12.833 Formatierungswarnungen in alten Quelltexten kümmern.

Im Laufe der Zeit können Profile dann mehr und mehr »verschärft« werden. Hierbei sollte ein Tempo angeschlagen werden, das zum einen konstanten Fortschritt hinsichtlich der Codequalität erzielt, andererseits die erforderlichen Nachbesserungsarbeiten in einem praktikablen Rahmen hält.

Eine große Hilfe ist hierbei, wenn sowohl IDE als auch der automatische Build auf dieselben Profildateien zugreifen und der Entwickler die verbleibenden nachzubessernden Stellen bereits in Echtzeit im Editor sehen kann und nicht jeweils auf einen CI-Build warten muss. Tipp: Stellen Sie zusätzlich für die IDEs Ihrer Entwickler passende Konfigurationen für die automatische Quelltext-Formatierung bereit, so dass die Anpassung an die geforderten Codekonventionen weitestgehend auf Knopfdruck geschehen kann.

IDE und CI-Builds sollten gleiche Konfigurationsdateien verwenden.

11.4 Nur messen, was auch beachtet wird

Widerstehen Sie der Versuchung, Metriken und Berichte auf Vorrat zu erzeugen. Sie schaden sich damit gleich mehrfach: Ihre Builds werden nicht nur langsamer, die wirklich wichtigen Kennzahlen gehen außerdem in der Menge unter und entfalten so keine Wirkung mehr.

Messen Sie also nur »aktive« Metriken, die verbindlich Aktivität einfordern können (»Das muss sofort behoben werden«). Dazu müssen Sie für jede Metrik einen binären Schwellwert festlegen (»Ab hier ist es ein Problem, darunter keines«). Weichen Sie hier nicht in unscharfe Qualitätsziele aus wie etwa »ziemlich wenige FindBugs-Fundstellen« oder »fast keine Checkstyle-Warnungen«. Zugegeben: Die Festlegung von binären Schwellwerten ist nicht einfach und wirkt im Detail meist auch einen Hauch beliebig. Kneifen gilt hier jedoch trotzdem nicht!

Aktive Metriken

11.5 Erscheinen neuer Plugins verfolgen

Eine sehr effiziente Möglichkeit, um aktuelle Build-Management-Trends zu verfolgen, ist ein regelmäßiger Blick in Hudsons Plugin-Verzeichnis:

Hat nämlich ein relevantes Build-Werkzeug seine experimentelle Phase überstanden, ist es meist nur noch eine Frage von Wochen, bis ein passendes Hudson-Plugin zur Verfügung steht. Am einfachsten abonnieren Sie dazu den RSS-Feed des Hudson-Projekts¹, der Sie nicht nur über neue Plugins, sondern auch über Aktualisierungen der Kernanwendung informiert.

1. <https://hudson.dev.java.net/servlets/ProjectRSS?type=news>

11.6 Den Spaß nicht vergessen

Zum Schluss: Vergessen Sie den Spaßfaktor nicht! Unterschätzen Sie nicht die positive Wirkung, die ein gewisses Augenzwinkern für die Akzeptanz eines neu eingeführten IT-Systems haben kann. Im Folgenden ein paar Anregungen dazu:

CI-Game

Das in Kapitel 8 bereits vorgestellte Plugin »Continuous Integration Game« macht aus dem Wechselspiel von Commits und anschließenden CI-Builds einen sportlichen Wettkampf unter Entwicklern: Für erfolgreiche Builds können Punkte gesammelt, für fehlschlagende aber auch wieder verloren werden.

Plugin »Chuck Norris«

Das Plugin »Chuck Norris« gehört zu den beliebtesten Plugins überhaupt. Es reichert Hudsons Benutzeroberfläche mit Zitaten über die inzwischen sprichwörtlichen Superkräfte des bekannten Schauspielers an. Obwohl dies für sich genommen schon unterhaltsam genug sein kann, existieren inzwischen unzählige Varianten, bei denen wahlweise Projektleiter, Geschäftsführer oder Vorstandsvorsitzende des eigenen Unternehmens den Platz von Chuck Norris einnehmen. Da das Plugin im Quelltext vorliegt, ist eine solche Variante problemlos in einer Mittagspause zu bauen. Das Kultpotenzial einer solchen Anpassung muss wohl nicht weiter erläutert werden ...

eXtreme Feedback Devices (XFDs)

Sperren Sie Continuous Integration nicht in die virtuelle Welt eines Browserfensters ein. Installieren Sie eXtreme Feedback Devices, um den Status Ihres Systems auch in die reale Welt zu tragen. Eine Installation aus Lava-Lampen, Roboterhäschchen oder Leuchtbären leistet nicht nur zweckmäßige Signalisierung, sondern kann im besten Fall auch das Gemeinschaftsgefühl unterstützen (»*Unsere Bärenampel ist grün. Und eure?*«).

12 Fazit und Ausblick

Am Ende dieses Buches angekommen, haben wir uns einen guten Überblick über Continuous Integration und deren momentanen Stand der Technik erarbeitet.

Aber selbstverständlich steht auch in diesem Bereich der Softwaretechnik die Entwicklung nicht still. Zum Ausklang des Buches möchten wir daher einen abschließenden »Blick in die Glaskugel« wagen und mögliche Auswirkungen aktueller Trends auf Continuous Integration und natürlich auch auf Hudson betrachten.

12.1 Continuous Integration – wie geht's weiter?

Vollautomatisierte Builds, zwingende Vorbedingung für CI, werden immer mehr zum *main stream*. Dabei verschiebt sich der Fokus der Build-Werkzeuge von der einfachen Ablaufsteuerung immer mehr in Richtung des Lebenszyklusmanagements einer Anwendung (*application life-cycle management, ALM*). Es geht also nicht mehr um das bloße Anstoßen von Compilern in der richtigen Reihenfolge. Das Build-Werkzeug kommuniziert stattdessen auch mit dem Versionsmanagement, verwaltet Abhängigkeiten in Bibliotheken und bringt neue Softwareversionen in den Produktivbetrieb aus. Dieser Trend ist etwa in Teams zu beobachten, die von Shell- oder Ant-Skripten auf Maven migrieren. Für ein CI-System bietet dies die Gelegenheit, durch Auswertung der verfügbaren Maven-Metadaten (Stichwort: POM-Datei) Projekte automatisch zu konfigurieren.

Verteilte Versionsmanagementsysteme (*distributed version control systems, DVCS*) wie Git oder Mercurial lösen zunehmend in vielen Unternehmen den Platzhirsch Subversion ab – und das nicht nur in sehr großen oder sehr innovativen Projekten. Das einfache Anlegen (*branching*) und spätere Wiederzusammenführen (*merging*) von Entwicklungszweigen in verteilten Versionsmanagementsystemen unterstützt eine aufgabenorientierte Arbeitsweise, in der neue Funktions-

*Von der Ablaufsteuerung
zum ALM*

*Verteilte Versionsmanage-
mentsysteme*

merkmale oder Fehlerkorrekturen isoliert zu anderen Codeänderungen entwickelt werden können. CI-Server können diese Eigenschaft verwenden, um Codeänderungen in privaten Builds auf dem Server zu testen, *bevor* diese Änderungen in ein gemeinsam genutztes Repository einfließen. Dadurch wird vermieden, dass ein fehlerhafter Commit eine ganze Arbeitsgruppe lahmlegt. Ausgewählte CI-Server (z.B. Jet-Brains TeamCity) unterstützen dieses Verfahren bereits jetzt – wenngleich durch proprietäre Erweiterungen für die wichtigsten Entwicklungsumgebungen und nicht durch Verwendung eines verteilten Versionsmanagementsystems.

Agile Softwareentwicklung

Praktiken der agilen Softwareentwicklung, insbesondere Extreme Programming (XP) und Scrum, finden zunehmend Akzeptanz. Da agile Konzepte sehr gut mit der Idee einer kontinuierlichen, vollautomatisierten Überprüfung von kleinen, inkrementellen Veränderungen harmonieren, besitzt Continuous Integration einen festen Platz im »agilen Werkzeugkoffer«.

Virtualisierung und Cloud Computing

Rechenkapazitäten werden durch den technischen Fortschritt immer preiswerter. Virtualisierung und Cloud Computing verändern zudem die Nutzungsweise und Abrechnung vorhandener Hardwaresressourcen. Intelligente Parallelisierung von Builds wird in der Zukunft daher eine immer wichtigere Rolle spielen, sowohl auf Ebene der Build-Werkzeuge als auch der CI-Server.

12.2 Hudson – wie geht's weiter?

Hudson: Produkt oder Plattform?

In der Verwendung von Hudson lassen sich momentan zwei Extreme ausmachen:

- Auf der einen Seite stehen die Anwender, die Hudson als *Produkt* verstehen: In diesem Fall wird Hudson installiert, konfiguriert und höchstens um ein paar nachinstallierte Plugins erweitert. Für gängige Werkzeugketten (Java/Subversion/Maven) ist dies völlig ausreichend.
- Auf der anderen Seite stehen Anwender, die Hudson als eine *Plattform* zur Build-Automatisierung verstehen: Gerade große Unternehmen schätzen die offenen Schnittstellen Hudsons, um bestehende Arbeitsabläufe innerhalb der Hudson-Infrastruktur nachbilden zu können – ohne dabei das Rad für die Grundfunktionalitäten eines CI-Systems neu erfinden und implementieren zu müssen.

Durch Hudsons Plugin-Konzept kann den Bedürfnissen beider Extreme Rechnung getragen werden. Allerdings erfordern Veränderungen in Hudsons Kernanwendung zunehmend mehr Fingerspitzengefühl, weil dabei natürlich keine Inkompatibilitäten mit bestehenden Installationen entstehen sollen. Je weitgehender aber der Zugriff auf Hudsons internes Datenmodell für eigene Anpassungen gestattet wird, desto schwieriger wird es, dort wirkliche Neuerungen einzuführen. Die Zukunft wird zeigen, ob diese Balance zwischen Innovation und Schnittstellenstabilität weiterhin so glücklich gehalten werden kann wie in den vergangenen Jahren.

Mit dem Weggang des Hudson-Gründers Kohsuke Kawaguchi von Sun/Oracle und der Gründung seines eigenen Unternehmens InfraDNA kamen Fragen nach der Zukunft des Hudson-Projektes auf: Wird Kawaguchi sich nun auf Hudson-Beratung und die Erstellung kommerzieller Erweiterungen konzentrieren und sich aus der Weiterentwicklung der Kernanwendung zurückziehen? Oder bleibt das Projekt bei der bisherigen Arbeitsweise, bei der eine kleine Gruppe von Committern rund um Kawaguchi die Kernanwendung unter MIT-Lizenz vorantreibt, während eine Vielzahl an Freiwilligen immer neue Ideen in Form von Plugins beisteuert? Kawaguchi hat sich deutlich zu letzterem Modell bekannt. Aus gutem Grund: Neben allen technischen Finessen dürfte wohl die große Gemeinschaft aus Nutzern und Entwicklern den maßgeblichen Unterschied zwischen Hudson und anderen CI-Servern ausmachen, seien sie Open Source oder kommerziell.

In diesem Sinne wäre dieses Buch nicht komplett ohne die Einladung an Sie, sich am Hudson-Projekt zu beteiligen. Sie müssen dazu nicht gleich zwangsläufig an der Kernanwendung »mitschrauben« oder komplette Plugins implementieren – obwohl das selbstverständlich höchst willkommen wäre. Sie leisten einen ebenso wichtigen Beitrag mit...

*Entwicklung der
»Community«*

Machen Sie mit!

- einem Verbesserungsvorschlag, den Sie auf einer der zwei Mailinglisten einbringen (users@hudson.dev.java.net bzw. dev@hudson.dev.java.net),
- einem gut dokumentierten Bericht Ihres Lieblingsfehlers ins Hudson-Fehlermanagementsystem (<http://issues.hudson-ci.org>),
- einem Einführungsvortrag zu Hudson in Ihrem Unternehmen bzw. der nächstgelegenen Java-User-Group,
- einem Blog-Eintrag, in dem Sie einen selbst entdeckten Hudson-Kniff mit anderen Anwendern teilen, oder
- gestifteter Arbeitszeit, in der Ihre besten Java-kundigen Mitarbeiter an Hudson bzw. an Plugins arbeiten dürfen (falls Sie Führungskraft sind).

12.3 Zusammenfassung

Continuous Integration wird an Bedeutung eher zunehmen.

Viele Zeichen sprechen dafür, dass Continuous Integration aufgrund der skizzierten Trends einen festen Platz im Softwareentwicklungsprozess einnehmen und an Bedeutung sogar noch zunehmen wird.

Mit Hudson steht Ihnen dazu ein äußerst anpassbarer CI-Server zur Verfügung, den Sie – je nach Ihren Anforderungen – als Produkt betreiben, aber auch als Ausgangsplattform für umfangreichere Erweiterungen verwenden können.

Die Rettung aus der »Integrationshölle«

Erinnern Sie sich noch an die »Integrationshölle« zu Beginn des erstens Kapitels? Ich hoffe, dieses Buch konnte einen Beitrag dazu leisten, unangenehme Situationen wie diese aus Ihrem Alltag zu verban-

nen. Ich wünsche Ihnen daher allzeit schnelle, aber vor allem allzeit stabile Builds mit Hudson.

Literatur und Quellen

- [Austvik09] Jorgen Austvik: *Hudson – Introduction and Experiences*,
URL <http://blogs.sun.com/austvik/resource/free_test_2009_hudson_intro_experiences.pdf>, März 2009.
- [Beck99] Kent Beck: *Extreme Programming Explained: Embrace Change*, Addison-Wesley Longman, 1999.
- [Clark04] Mike Clark: *Pragmatic Project Automation: How to Build, Deploy, and Monitor Java Applications*, Pragmatic Programmers, 2004.
- [Cockburn01] Alistair Cockburn: *Agile Software Development*, Addison-Wesley Professional, 2001.
- [Duvall07] Paul M. Duvall, Steve Matyas, Andrew Glover: *Continuous Integration: Improving Software Quality and Reducing Risk*, Addison-Wesley, 2007.
- [Fielding 2000] Roy T. Fielding: *Architectural Styles and the Design of Network-based Software Architectures*. Dissertation. University of California at Irvine, 2000.
- [Fowler00] Martin Fowler: *Continuous Integration*,
<http://martinfowler.com/articles/continuousIntegration.html>
- [Hudson10] *Hudson, an Extensible Continuous Integration Server*,
<http://hudson-ci.org>
- [Hüttermann07] Michael Hüttermann: *Agile Java-Entwicklung in der Praxis*, O'Reilly, 2007.
- [Hüttermann10] Michael Hüttermann: *Agile ALM*, Manning, 2010.
- [Kawaguchi09] Kohsuke Kawaguchi: *Blueprints for Deploying Software Projects on Hudson*, Webinar-Aufzeichnung mit Downloadmöglichkeit,
<https://slx.sun.com/1179275729>, Sun Microsystems, Oktober 2009.

- [Larman10] Craig Larman, Bas Vodde: *Practices for Scaling Lean and Agile Development: Large, Multisite, and Offshore Product Development with Large-Scale Scrum*, Addison-Wesley Longman, 2010.
- [Popp09] Gunther Popp: *Konfigurationsmanagement mit Subversion, Maven und Redmine*, dpunkt.verlag, 2009.
- [Scott06] Ambler Scott, Sadalage Pramod: *Refactoring databases: Evolutionary database design*, Addison-Wesley, 2006.
- [Smart08] John Ferguson Smart: *Java Power Tools*, O'Reilly, 2008.
- [Smart10] John Ferguson Smart: *Continuous Integration with Hudson*, O'Reilly Media, 2010. Kostenloser Download des Open-Source-Buches: <http://www.wakaleo.com/books/continuous-integration-with-hudson-the-book>
- [ThoughtWorks10] ThoughtWorks Inc.: *CI Feature Matrix*, URL<<http://confluence.public.thoughtworks.org/display/CC/CI+Feature+Matrix>>

Index

A

- Aktive Metriken 291
- Amazon Web Services 206
- Ansicht 175
- Artefakt 52
- Atlassian Bamboo 76
- Auditierung 218
- Aufwand einer CI-Einführung 283
- Auslösen von Builds
 - bei Änderungen im Versionskontrollsystem 128
 - durch Abhängigkeiten zwischen Projekten 129
 - durch Plugins 131
 - manuell 127
 - zeitgesteuert 127
- Authentifizierung 213
- Automatisierte Ausbringung (Deployment) 42
- Automatisierte Berichte 41
- Autorisierung 217

B

- Backups 100
- Benutzerschnittstelle 55
 - Kommandozeile 59
 - REST-Schnittstelle 57
 - Webbrowser 55
- Benutzerverwaltung 213
 - Active Directory 216
 - LDAP 215

Build

- automatisierter 32
- Parallelisierung 179
- parametrisierter 171
- selbsttestender 35
- Verteilung 194

C

- Checkstyle integrieren 151
- Cloud Computing und CI 206
- Command Line Interface, CLI 59
- Continuous Integration
 - Blick in die Zukunft 293
 - Definition 13
 - Die 10 Praktiken 31
 - Vorteile 23
- CruiseControl 69
- CVS 120

D

- Datensicherungen Siehe Backups
- Deskriptoren 237

E

- Eclipse-Plugin 142
- E-Mail-Benachrichtigungen 136
- Erweiterungspunkt 249
- Erweiterungspunkte 237
- Extension Point Siehe Erweiterungspunkt
- eXtreme-Feedback-Geräte anbinden 144

F

FindBugs integrieren 151

H

Hudson

- Aktualisierung 99
- Datei-Layout 93
- Deinstallation 103
- Installation 87
- Sicherheit 212
- Systemkonfiguration 89
- Systemvoraussetzungen 91

Hudson Build Monitor 141

I

Informationsradiatoren 143

Installation 95

- als Windows-Dienst 96
- manuell 95
- mit Paketmanager 95

J

Javadoc integrieren 146

Jelly 247

JetBrains TeamCity 81

Job 51

- einrichten 108
- externer 51, 118
- Free-Style-Projekt 116
- Maven-2-Projekt 117
- Multikonfigurationsprojekt 118, 189

JUnit integrieren 132

P

Plugin

- Atlassian JIRA 164
- Backup 102
- Build Timeout 188
- Claim 220
- Cobertura 158
- Configuration Slicing 221
- Continuous Integration Game 227
- Copy Artifact 183
- Dashboard View 178
- Description Setter 225
- Doc-Links 149
- Downstream Build View 130
- Downstream View 182
- Downstream-Ext 130
- Email-ext 138
- Green Balls 226
- Hudson Promoted Builds 224
- JIRA 164
- Join 185
- JUnit Attachments 135
- Locale 227
- Locks and Latches 186
- Mantis 168
- Parameterized Trigger 187
- Promoted Builds (Simple) 223
- Scriptler 222
- Sonar 161
- Static Analysis Utilities 155
- Task-Scanner 157
- Twitter 141
- Violations 155
- Warnings 156
- xUnit 135

K

Kawaguchi, Kohsuke 45

Plugins

- Deaktivierung 103
- Deinstallation 104
- Entwicklung eigener 230
- Installation 103
- Lokalisierung eigener 277

PMD integrieren 151**Produktivsystem** 49**Projekt** 51

- Free-Style-Projekt 51
- Maven-2-Projekt 51
- Multikonfigurationsprojekt 51

Q**Queue** 54**R****Repositories** 49**Repository-Browser** anbinden 124**RSS-Feeds** abonnieren 140**S****Schulung** 287**Slave** 54**Slave-Knoten** 195**Software Configuration Management,**
SCM 48**Stapler** 245**Statusanzeige (Bälle)** 112**Subversion** 120**T****TestNG** integrieren 132**ThoughtWorks Cruise** 72**Ticketing-System** 48**Touchstone Builds** 191**U****UML-Diagramme** erzeugen 147**V****Versionsmanagement**

- integrieren 120

Versionsmanagementsysteme

- verteilte 293

View Siehe **Ansicht****Virtualisierung und CI** 201**VMware Server** 202**X****XStream** 247