Continuous Integration Server mit Hudson

Praxissemesterbericht

von

Michael Legner
aus Ellwangen/Jagst

Matrikelnummer: 27487

Martin-Luther-Str. 1
73463 Westhausen
07363/4999

HTW Aalen

Hochschule für Technik und Wirtschaft
Betreuer: Prof. Roy Oberhauser

Abgabetermin: 04.10.2010

Angaben zur Praxisstelle

holometric technologies

Unternehmen	Holometric Technologies GmbH
Branche	Softwareentwicklung
Abteilung	Softwareentwicklung
Straße	Willy-Messerschmitt-Str. 1
PLZ, Ort	73457 Essingen
Betreuer	Andreas Glaubitz
Tel.	07365-9645-31
E-Mail	andreas.glaubitz@holometric.de

Die Firma *Holometric Technolgies* wurde 1994 gegründet. Sie entwickelt Messsoftware, die in der Entwicklung, Produktion und Qualitätssicherung im Fahrzeugbau, Maschinenbau, Werkzeugbau und Raumfahrt eingesetzt wird.
Im Januar 2009 wurde die Firma von der Carl Zeiss IMT GmbH übernommen.

Kurzbeschreibung

Continuous Integration bzw. kontinuierliche Integration ist ein wesentlicher Bestandteil der agilen Sofwareentwicklung. Bei der Firma *Holometric Technologies* wird bereits der agile Prozess *Scrum* eingesetzt.
Bislang dauert es ca. 130 Minuten, um eine neue Version von Holos, einem der Hauptprodukte, zu bauen. Um die Buildzeit zu verkürzen und den Entwicklern schnelleres Feedback zu liefern, soll das Bauen der Applikation auf einen Continuous Integration Server portiert werden. Verwendet wird der quelloffene Hudson Continuous Integration Server.
Das Projekt umfasst die Installation und Konfiguration des Servers sowie das erstellen neuer Buildskripte. Zustätzlich sollen kleine Werkzeuge und Skripts erstellt werden, die die Administration und Wartung des Systems vereinfachen.
Das Projekt wurde erfolgreich abgeschlossen, der neue Buildserver wurde in den Produktivbetrieb übernommen.

Vorgehen

In den Grundlagen werden zunächst die Grundsätze der kontinuierlichen Integration behandelt. Danach wird näher auf die Struktur und die Besonderheiten des *Hudson CI-Servers* sowie der Anwendung *Holos* eingegangen. In Kapitel 2 wird auf die Umsetzung des Projektes beschrieben. Anschließend werden noch Erweiterungsmöglichkeiten für den Continuous Integration Server und das Programm kurz vorgestellt, die im Rahmen des Projekts nicht umgesetzt werden konnten. Im Anhang sind noch Informationen zur Administration und Wiederherstellung des Servers gesammelt.

Inhaltsverzeichnis

1 Grundlagen

1.1 Continuous Integration

Continuous Integration oder kontinuierliche Integration (auch permanente oder fortlaufende Integration) beschreibt den Prozess des regelmäßigen Neubauens einer Anwendung auf einem Server. Häufig wird es mit weiteren Komponenten wie automatische Tests oder statische Code-Analyse kombiniert. Das bereits ältere Prinzip wurde durch das Extrem Programming populär.
Die Idee hinter diesem Konzept ist sehr einfach und passt sehr gut zum Agile Manifesto [1]:

1. Jeder Entwickler soll seine Änderungen in kleine, inkrementel aufeinander aufbauende Teile zerlegen und mindestens einmal täglich in das Versionskontrollsystem (Versioncontrollsystem, VCS) übertragen.

2. Sobald eine Änderung in das VCS übertragen wurde, wird die Software automatisch neu gebaut. Dabei wird großen Wert darauf gelegt, dem Entwickler möglichst schnell eine Rückmeldung zu liefern, ob seine Änderungen in Konflikt zu denen anderer Entwickler stehen.

1.1.1 Best Practices

Um das volle Potential der kontinuierlichen Integration auszuschöpfen, müssen einige Grundvorrausetzungen erfüllt sein. Die folgende kurze Einführung basiert auf Martin Fowlers populärem Artikel [2] zu diesem Thema:

Ein einzelnes Repository

Alle zum Bauen der Anwendung nötigen Dateien müssen im Repository eines Versionskontrollsystems sein. Jeder Entwickler mit einer neu installierten Maschine soll sich nur die notwendigen Tools installieren (z.b. eine IDE oder einen Compiler), die Quelltexte und Konfigurationsdateien aus dem Repository auf seinen Rechner übertragen um sich eine aktuelle Version der Anwendung zu bauen.
Viele Versionskontrollsysteme unterstützen das Anlegen von Zweigen oder Branches des Programms an. Beispiele sind frühere, große Revisionen eines Programms oder Zweige für Experimente. Diese Funktion sollte allerdings nicht übertrieben oft genutzt werden, der Hauptzweig (mainline) ist der zentrale Punkt, in den jeder Entwickler seine Änderungen einfügt.

Automatisierung des Buildprozesses

Der Prozess des Bauens der Anwendung sollte so weit wie möglich automatisiert werden, da wiederkehrendes abarbeiten derselben Schritte durch den Entwickler nicht nur Zeitverschwendung ist, sondern auch Fehlern einen exzellenten Nährboden bietet. Systeme, um das Bauen zu automatisieren gibt es für alle ernst zu nehmende Plattformen und Programmiersprachen. Oft sind sie auch nicht an eine Plattform oder Sprache gebunden. Bekannte Werkzeuge sind make für Unix und Unix-like Plattformen oder MSBuild für

.NET. Daneben existieren spezialisierte Skriptsprachen wie Ant oder Rake. Integrierte Entwicklungsumgebungen (Integrated Development Environments, IDEs) haben häufig eigene, interne Buildprozesse. Diese sind aber meist an die IDE gebunden und lassen sich nur schwer von extern steuern und automatisieren.

Selbsttestender Code

Tests sind ein elementarer Bestandteil der Qualitätssicherung in der Softwareentwicklung. Viele Tests lassen sich über Test-Frameworks, wie die der XUnit-Familie oder Selenium, automatisieren. Die Technik des selbst testenden Codes hat mit dem Aufstieg des Extreme Programmings und der Testgetriebenen Entwicklung (Test Driven Development, TDD) enorm an Popularität gewonnen.

Tägliche Commits in den Hauptzweig

Der normale Entwicklungsablauf mit kontinuierlicher Integration sieht vor, dass ein Entwickler seine Änderungen am Code vornimmt, diese auf seiner Arbeitsstation testet und dann in das Repository überträgt. Auf diese Weise werden Konflikte zwischen zwei Versionen schnell erkannt und können ebenso schnell behoben werden. Potentielle Konflikte, die wochenlang nicht entdeckt werden, weil der Entwickler alleine an seinem Teil arbeitet und in dieser Zeit nichts überträgt, lassen sich später nur schwer finden und beheben.
Jeder Entwickler sollte mindestens einmal täglich seine von ihm getesteten Änderungen in das VCS übertragen. In der Praxis ist es aber sinnvoll, dies bei jeder Änderung zu tun.

Bauen auf einer zentralen Maschine

Bevor ein Entwickler seine Änderungen überträgt, muss er einen privaten Build auf seiner eigenen Maschine starten und testen. Um Fehler durch unterschiedlichen Konfiguration auszuschließen, sollte ein weiterer Build auf einer separaten Maschine erfolgen. Erst wenn dies erfolgreich ist, kann die Änderung als Abgeschlossen angesehen werden.

Schnelles bauen

Damit die Entwickler schnell Rückmeldung Als Idealwert wird im Extreme Programming eine Zeit von 10 Minuten pro Build angenommen [3]. Um an diese Zeit heran zu kommen ist es sinnvoll, die Anwendung in Module einzuteilen, die einzeln gebaut werden können. Zudem lassen sich mit vielen Buildwerkzeugen inkrementelle Builds erstelle: das Programm merkt hierbei, in welchem Teilen des Programms Änderungen gemacht wurden und kompiliert nur diese neu, um sie anschließend mit den anderen Teilen zu einer neuen, ausführbaren Datei zu binden.

Tests in einer Kopie der Einsatzumgebung

Tests sollte immer ein einem möglichst exakten Abbild der späteren Einsatzumgebung stattfinden, um Fehler durch unterschiedliche Konfigurationen auszuschließen. Beschränkungen

in der Wirtschaftlichkeit können durch den Einsatz von Virtualisierung gemindert werden.

Einfacher Zugriff auf den letzten Build

Die Ergebnisse eines Builds sollten an einem zentralen Ort zugänglich sein, um beispielsweise immer eine Version für Demonstrationszwecke zu haben. Es kann auch nützlich sein, mehrere Versionen vorrätig zu halten. Sollte z.b. in der letzten Versionen ein Fehler erst spät entdeckt werden, kann auf einen früheren, stabileren Build zurück gegriffen werden.

Für jeden sichtbaren Projekt-Status

Das als Extreme Feedback bekannte Konzept sieht vor, dass die Ergebnisse eines Builds für jeden gut sichtbar sein sollte. Dafür können Monitore an zentralen Stellen, Lampen oder akustische Signal eingesetzt werden, speziell wenn ein Build fehlschlägt.

Automatische Verteilung

Für die kontinuierliche Integration werden mehrere Umgebungen benötigt, beispielsweise für das Bauen oder Testen der Applikation. Die Verteilung der neuesten Builds in diese Umgebungen sollte automatisch ablaufen, da es mehrmals täglich neue Versionen geben kann und sich die Verteilung über Skripte einfach automatisieren lässt.
Der nächste logische Schritt ist, regelmäßig neue Builds für die Kunden freizugeben. Dann ist es aber auch sinnvoll, automatisch auf eine frühere stabile Version zurück wechseln zu können (automated rollback), sollte etwas schief gehen. Dies vermindert die Anspannung beim Freigeben einer neuen Version und durch die häufigeren Freigaben erreichen neue Funktionen den Kunden schneller.

1.1.2 Vorteile der kontinuierlichen Integration

Die Integration von Software Projekten war eine langwierige, undurchsichtige und unsichere Angelegenheit. Schätzung über die Dauer sind nur sehr schwer möglich und dann auch nur sehr ungenau, es bleibt ein hohes Risiko. Paul Duvall gibt in seinem Buch [4] das Ziel aus, Intergration zu einem Non-Event, einer Nebensache zu machen. Durch die kontinuierliche Integration wird das Risiko fast komplett eliminiert.
Projekte mit kontinuierlichen Integration haben im Schnitt weniger Fehler als vergleichbare Projekte. Die Moral der Entwickler wird durch weniger aufgestauten Fehler verbessert. Die Barriere zwischen Entwicklern und Kunden wird durch die regelmäßige und häufige Freigabe neuer Versionen kleiner.

1.2 Hudson Continuous Integration Server

Hudson ist ein in Java geschriebener Continuous Integration Server, der bei Oracle (vormals Sun Microsystems) entwickelt wird. Federführend bei der Entwicklung ist Kohsuke Kawaguchi. Er verließ Oracle Anfang April 2010 um seine eigene Firma zu gründen [5], die kommerziellen Support und Service für Hudson anbietet.
Hudson zeichnet vor allem durch seine leichte Installation und Konfiguration, seine große Anzahl an PlugIns sowie seine kurzen Releasezyklen aus. Daneben bietet er die Funktion, wenn benötigt Buildschritte auf andere Maschinen auszulagern, um den Buildvorgang zu beschleunigen und die Anwendung für mehrere Plattformen gleichzeitig zu bauen (Details dazu siehe Abschnitt 3.4).

1.2.1 Warum Hudson

Kriterien bei der Auswahl des Continuous Intergration Servers waren die Möglichkeit, sowohl ältere Projekte auf Basis von Visual Studio 6 und Visual Studio 2003, also auch zukünftige auf der .NET-Platform in einem System zu verwalten. Als Versionskontrollsystem soll Subversion weiter verwendet werden. Nur Hudson bietet hier die notwendige Flexibilität. Durch seine PlugIn-Schnittstelle und die große Zahl an verfügbaren PlugIns kann Hudson auf die Ansprüche des Entwicklerteams zugeschnitten werden. Neben der sehr aktiven Community, die eine langfristige Versorgung mit Updates gewährleistet, wird auch professioneller Support angeboten. Ein weiterer Punkt für Hudson ist seine einfache, webbasierte Oberflächen, die auch von Nicht-Entwicklern genutzt werden kann.

1.2.2 Installation

Hudson kann direkt von der Kommandozeile gestartet und genutzet werden, für die Oberfläche wird der integriete Winstone Webserver genutzt. Alternativ lässt er sich auch in einem Applikationsserver wie Apache oder Glassfish betreiben. Unter Windows kann Hudson als Systemdienst betrieben werden.
Da Hudson in Java geschrieben ist, setzt es in jedem Fall eine installierte Java Runtime Environment Version 1.5 oder höher voraus.

1.2.3 Administration

Hudson wird komplett über eine Web basierte Oberfläche gesteuert. Für jeden Build wird in Hudson ein sog. Job angelegt, für den individuelle Einstellungen gespeichert werden. Neben dem generischen 'Freestyle' Job können auch Jobs für Maven2 und sog. Matrix-Jobs angelegt werden. Letztere sind für Projekte, bei denen das Bauen und Testen auf mehreren Plattformen durchgeführt werden muss, z.b. Java-Anwendungen mit mehreren Java-Versionen.
Zusätzlich zur graphischen Oberfläche lässt sich Hudson über Groovy-Scripts oder ein Kommandozeileninterface administrieren. Alle Einstellungen werden in XML-Dateien gespeichert, die auch von Hand bearbeitet werden können. Damit die auf diese Weise getätigten Änderungen dann übernommen werden, muss der Server neu gestartet werden.

1.2.4 Jobs anlegen und konfigurieren

Hudson kann von Haus aus bereits mit den Versionskontrollsystemen (Versioncontrollsystem, VCS) Subversion (SVN) und Concurrent Version System (CVS) zusammen arbeiten, weitere lassen sich per PlugIn nutzen. Um einen neuen Build eines Jobs zu starten bietet Hudson mehrere Möglichkeiten: Zum einen kann ein Build Zeit gesteuert angestoßen werden, was Hudson als Ersatz für Cron-basierte Builds nutzbar macht. Allerdings entspricht dies nicht dem Prinzip der kontinuierlichen Integration und der CI-Server bleibt so deutlich unter seinen Möglichkeiten. Deshalb kann Hudson auch das VCS in regelmäßigen Abständen auf Änderungen prüfen und gegebenenfalls einen neuen Build starten. Dies ist allerdings nur für moderne VCS wie Subversion geeignet, da bei älteren wie CVS, diese Aktion eine hohe Laufzeitkomplexität hat und das Bauen von vielen, kleinen Jobs stark verlangsamen würde. Alternativ kann ein sog. Post-Commit-Hook eingesetzt werden, mit dem das VCS nach einem erfolgreichen Commit an Hudson eine Nachricht sendet, um einen neuen Build zu starten. Als weitere Möglichkeiten können Builds per E-Mail, über die URL zum Job oder über die Weboberfläche manuell gestartet werden.
Projekte von Maven oder Skripte aus Buildskriptsprachen wie Ant oder Rake können direkt genutzt werden (evtl. erfordern sie die Installation eines PlugIns und/oder des entsprechenden Programms; im Falle von Maven gibt es eine eigenen Typ für den Job), aber auch Shell- und Batch-Skripte können genutzt werden.
Die erzeugten Binärdateien werden als Artefakte bezeichnet und können per PlugIn an andere Jobs weitergereicht werden. So lässt sich auch die für dieses Projekt geforderte Aufteilung der Anwendung realisieren, um die Dauer des Buildvorgangs zu verkürzen. Artefakte können von Hudson archiviert werden, um sie später weiter zu nutzen oder nur um ihre Herkunft nach zu verfolgen.

1.2.5 File-Fingerprinting

Für jedes Artefakt kann Hudson einen sog. Fingerprint aufzeichnen, anhand dessen sich die Herkunft bzw. der Build, in dem die Datei erzeugt wurde, nachvollziehen lässt. Dabei handelt es sich um einen md5-Hash der Datei, den Hudson in der Konfigurationdatei des betreffenden Builds speichert. Über eine Funktionen in der Oberfläche können erzeugte Dateien auf ihre Herkunft überprüft werden, indem sie zum Hudson Masterserver hoch geladen und mit den Einträgen in der Datenbank verglichen werden.

1.2.6 PlugIns

Hudson ist sehr schlank konzipiert, allerdings verfügt er über einen PlugIn-Schnittstelle. Es können weitere Versionskontrollsystemen angebunden werden, neue Build-Schritte, Integration von Build-Werkzeugen, Anbindung an externe Anwendungen und einiges mehr realisiert werden. Für das Entwickeln von Erweiterungen ist zwingend der Einsatz des Java Buildmanagementservers Maven erforderlich.

1.2.7 Verteiltes Bauen

Hudson unterstützt die Anbindung sog. Slaves, um den Buildprozess auf mehrere Rechner zu verteilen. Dadurch lässt sich nicht nur die Builddauer verringern, sondern auch Builds für mehrere Platformen erstellen. Dazu können die Slaves spezialisiert werden, um die an die jeweilige Platform gebundnen Aufgaben auszuführen. Sie können auch als allgemeine Build-Knoten genutzt werden, dann verwaltet Hudson die Aufgaben selbstständig. Für dieses Projekt sind Buildslaves aber nicht vorgesehen, mehr dazu siehe Abschnitt 3.4.

1.3 Holos

Holos ist eines der beiden Hauptprodukte von *Holometric Technologies*. Es dient der Flächenmessung und wird in der Qualitätssicherung im Karosseriebau eingesetzt. Das in C und C++ geschriebene Programm besteht aus 54 einzelnen Projekten, die in einer VisualStudio-Solution zusammen gefasst und die durch Abhängigkeiten verbunden sind. Das Projekt ist in einem Subversion-Repository gespeichert.

1.3.1 Bisheriger Buildprozess

Bisher wird jede Nacht ein Ant-Skript aufgerufen, dass Holos neu baut. Der komplette Vorgang dauert etwa 130 Minuten, was für die kontinuierliche Intergration zu viel ist. Projekt wird in seine einzelne Module aufgeteilt, um die parallel zu bauen. Zum eigentlichen Projekt kommen noch die automatisierte Lokalisierung mit Passolo, sowie das Generieren eines Installers mit InstallShield dazu, diese sollen aber nur Nachts ausgeführt werden.

1.3.2 Problemstellungen

Das komplette Programm muss in seine einzelnen Module aufgespalten werden um die Bauzeit niedrig zu halten. Dabei müssen die Abhängigkeiten zwischen den Modulen beachtet werden. Die nachgelagerten Vorgänge der automatischen Lokalisierung und des Erstellens eines Installers durch externe Programme stellen ebenfalls eine Schwierigkeit dar.
Holos wird bereits seit über 20 Jahren entwickelt und wurde bereits einmal von Unix auf Windows portiert. Dementsprechend viele überflüssige Reste sind noch in den Projekten zu finden. Auch wurde das Projekte bisher immer statisch aus einem Ordner erstellt, weswegen doppelseitige Abhängigkeiten zwischen Projekten bislang nicht aufgefallen sind. Des weiteren wurden im Ant-Skript einige Programmbibliotheken, die meistens von Drittanbietern kommen und nur in Binärform vorliegen, statisch kopiert, was in der kontinuierlichen Integration nicht vorgesehen ist.
Bisher konnte nur ein Entwickler einen neuen Build starten, eine Dokumentation existiert nicht. In Zukunft soll es jedem Entwickler möglich sein, einen Hotfix-Build zu starten. Alle Entwickler soll per E-Mail Feedback über die Builds bekommen, an denen sie beteiligt sind.

2 Umsetzung

2.1 Buildserver mit Hudson

2.1.1 Aufbau des Buildservers

Der Buildserver soll komplett neu aufgesetzt werden. Hardwareseitg wird ein Dell PowerEdge R610 mit zwei Intel Xeon Quadcore Prozessoren mit je 2,27Ghz verwendet. Es stehen 8 Gigabyte Arbeitsspeicher und 160 Gigabyte Festplattenspeicher in einem Striping-RAID zur Verfügung. Als Systemsoftware soll der VMWare ESXi Server 4.0 eingesetzt werden, der Buildserver selber wird als virtuelle Maschine mit Microsoft Windows Server 2008 R2 erstellt. Als Ressourcen werden vier der acht Rechenkerne, 4 Gigabyte Arbeitsspeicher und 80 Gigabyte Festplattenspeicher der Maschine zugeteilt.

2.1.2 Installation

Der VMWare ESXi Server wird als bootbares CD-Image vertrieben, mit dem das Grundsystem installiert wird. In der spartanischen, grafischen Oberfläche lassen sich nur die grundlegendsten Einstellungen, wie die Netzwerkkonfiguration, festlegen. Alles weitere wird über den vSphere Client konfiguriert.

VMWare ESXi Server 4.0

Der kostenlose ESXi Server ist der kleine Bruder des kommerziellen ESX Servers, nun Teil der sog. VMWare vSphere. Der ESX Server verfügt über ein eingebautes Verwaltungswerkzeug, die linuxbasierte Service Console, mit dem der Server verwaltet werden kann. Beim ESXi Server wurde diese entfernt, am Gerät selber lassen sich nur sehr wenige Einstellungen vornehmen, die weiterführende Konfiguration und Wartung wird über den vSphere Client durchgeführt.
Auf dem ESX Server können über einen Skript-Konsole direkt Skripte ausgeführt werden, der ESXi Server verfügt nicht über diese Funktionalität. Stattdessen können über den vSphere Client Skripte gestartet werden, diese allerdings mit einer entscheidenden Einschränkung: Die Skripte haben nur lesend Zugriff, für mehr Rechte muss der Server in einem (kostenpflichtigen) vSphere Verbund laufen.
Seit der Version 4.0 steht der VMWare ESXi Server nur noch als 64Bit Version zur Verfügung und läuft nur noch auf entsprechender Hardware. Als Gastbetriebssysteme werden 32Bit Systeme weiterhin unterstützt.
[6]

VMWare vSphere Client

Mit dem vSphere Client kann ein Server oder eine Serverfarm von ESX, ESXi oder vSphere Servern verwaltet werden. Es können neue virtuelle Maschinen auf dem Server angelegt und konfiguriert werden. Jeder Maschine können Ressourcen, unabhängig von der Anzahl der CPU Kerne oder des Arbeitsspeichers im Hostrechner, zugewiesen werden. Zur genutzten Leistung bietet der Client umfangreiche Daten und Diagramme, mit

denen schnell Ressourcen-Knappheit oder Überfluss für einer Maschine festgestellt werden kann. Die Gastinstanzen selber können über ein integriertes Remote-Tool (Konsole genannt) fern gesteuert werden.

Windows Server 2008 R2

Der im Oktober 2009 veröffentlichte Windows Server 2008 R2 wurde parallel zu Windows 7 (NT 6.1) entwickelt und basiert auf der selben Plattform. Das Betriebsystem ist erstmalig nur in einer 64bit Version für x86-kompatible CPUs verfügbar. Der verfügbare Hyper-V Hypervisor für virtuelle Maschinen wird nicht genutzt.

Hudson CI

Der Hudson Continuous Integration Server wird als Windows-Systemdienst betrieben, als Webserver dient der integrierte Winstone-Server.

2.2 Buildprozess mit Hudson

2.2.1 Build Skripte

Die Projekte von Holos werden größtenteils gleich gebauen, weshalb für die meisten das selbe Buildskript verwendet wird. Drei Projekte, Passolo, InstallShield sowie die Programmbibliotheken, benötigten aber komplett andere Vorgänge, weshalb für diese Jobs eigene Skripte erstellt wurden. Bei der Erstellung aller Buildskripte wurde darauf geachtet, möglichst viele Variable zu verwenden, damit der Wartungsaufwand gering ausfällt.

Standard Skript

Das Standard Buildskript wird von den meisten Jobs verwendet und ist ein Skript für die Windows PowerShell. Diese wurde einem normalem Windows Batchskript vorgezogen, da die PowerShell wesentlich mehr Funktionen bietet. Für dieses Projekt waren vor allem die Möglichkeiten im Bereich der Stringverarbeitung entscheident. Für die Unterscheidung der Jobs in ihrer Projektzugehörigkeit wird der Name des Jobs verarbeitet.
Holos wird noch in mehreren Versionen gepflegt, die sich in ihrer Struktur allerdings nur unwesentlich unterscheiden. Für alle Module kann somit das gleiche Buildskript verwendet werden. Damit die Jobs in Hudson unterschieden werden können, haben ihre Namen eine spezielle Form, anhand der sowohl die Produktversion als auch das Modul unterschieden werden kann:

```
holos $Produktversion Job $Branch $Modul
```

Listing 2.1: Standard Buildskript Quelltextfragment Beginn

```
1 $FullJobName = $env:JOB_NAME
2 if( $FullJobName.contains( "Job" ) ){
3         $Project = $FullJobName.substring( $FullJobName.IndexOf
              ( "Job" ) + 4 )
```

Der Modulname und der Branch wird im Verlauf des Buildskripts mehrmals benötigt, deshalb wird er zu Beginn, Codefragment siehe Listing 2.1, heraus gefiltert. Das Schlüssel- bzw. Trennwort ist „Job“. Da Hudson den Name des Jobs auch als Ordnername verwendet wird, ist die Verwendung eines Sonderzeichens kritisch. Viele Sonderzeichen sind für Ordner- und Dateiname unter Windows nicht zulässig, die Nutzung der erlaubten kann trotzdem unerwünschte Nebeneffekte auslösen.

Als nächstes wird mit dem in Listing 2.2 abgebildeten Code die passende Variante zum starten des Programms zur Erhöhung der Versionsnummer gewählt. Die Unterscheidung ist notwendig, da einige wenige Projekte über keine VersionNo.h-Datei verfügen und die Fehler unnötig im Buildlog auftauchen und vom Hudson Warnings

Listing 2.2: Standard Buildskript Quelltextfragment HTVersionUp Auswahl

```
1  $Branch = $FullJobName.substring( 11, 1 )
2  switch( $Project ){
3      "BCGCBPro" { break }
4      default {
5        switch ( $Branch ){
6            "U" { }
7            "u" {
8                  if ( $env:NIGHTLY -eq "true" ) {
9                    .\HTVersionUp\HTVersionUp.exe /u /n $Project
10                 } else {
11                   .\HTVersionUp\HTVersionUp.exe /u /c $Project
12                 }
13                 break
14               }
15           default { Write-Host -ForegroundColor RED Unable to
                 determine Branch: $branch ; exit 1 }
16       }
17     }
18   }
```

PlugIn fälschlicherweise als hoch priorisierte Warnungen gezählt werden (detaillierte Beschreibung in Abschnitt 2.3.2). Primär handelt es sich dabei um lizenzierte Softwarepakete, die nicht von den Entwicklern bei *Holometric Technologies* gepflegt werden. Zudem muss der Branch unterschieden werden, auf dem momentan gearbeitet wird. Für Details dazu siehe Abschnitt 2.4.5. Anschließend werden noch einige Codemetriken mit Hilfe des Zusatzprogramms "C and C++ Code Counter„ bestimmt. Nähere Informationen dazu siehe Abschnitt 2.3.1.

Listing 2.3: Standard Buildscript Codefragment Aufruf Build Werkzeug

```
1  switch( $Project ){
2      "include" { break }
3      "RegelGeoAPI" { vcbuild.exe /r source\RegelGeo\RegelGeo.
           vcproj "ReleasewithoutGUI"; break }
4      default { vcbuild.exe /r source\$Project\$Project.vcproj "
           $env:BUILD_TYPE" }
5    }
```

Es folgt der Aufruf des eigentlichen Build Werkzeugs, beschrieben in Listing 2.3. Auch hier ist eine Unterscheidung notwendig, da das Projekt 'Regel Geometrie' zweimal gebaut werden muss: einmal mit graphischer Oberfläche und einmal ohne, letztere wird für den PowerConverter und weitere Module in Form der Bibliotheken benötigt. Den eigentlichen Bauvorgang übernimmt der Visual C++ Project Builder (`vcbuild.exe`), welcher ein Teil von Visual Studio 2005 ist.
Normalerweise wird die Art des Builds, also Debug oder Release und für welche Plattform, über die Umgebungsvariable `BUILD_TYPE` festgelegt. Im Falle des Projektes „include“ weicht der Ablauf gegenüber dem der meisten anderen Jobs ab, da dieser Job hauptsächlich für die Abholung und Bereitstellung von zentralen Programm-Bibliotheken zuständig ist. Im nächtlichen Build wird zudem noch die Tages-Versionsnummer geändert. Ein eigenes Skript für diesen Job ist aber nicht nötigt, da die darin benötigten Funktionen auch im normalen Skript vorhanden sind. Es werden lediglich einige Ausnahmen benötigt, welche die Pflege eines separaten Skriptes nicht rechtfertigen.

Post Build Commit

Die im Laufe des Buildvorgang geänderten Dateien mit Versionsnummern müssen wieder in das Repository zurück übertragen werden, damit sie beim nächsten Durchlauf wieder korrekt erhöht werden. Die Daten im Workspace müssen an einer anderen Stelle gesichert werden, da Hudson den Workspace leert und aus seinen Quellen neu Aufbaut, sollte er Unstimmigkeiten festtellen. Da an dieser Stelle nicht der in Hudson bzw. einem PlugIn integrierte Subversion Client genutzt werden kann, wird ein kommandozeilen basierter Client benötigt, in diesem Fall kommt SlikSVN [7] zum Einsatz. Im Post-Build-Task wird ein PowerShell Skript ausgeführt.

Passolo

Mit Passolo [8] werden automatisch lokalisierte Versionen einer Software erzeugt. Dazu werden die Text-Strings aus fertigen Binärdateien ausgelesen, die passende Übersetzung gewählt und eine DLL mit der Übersetzung erstellt. Die Übersetzten Texte müssen bereits in einer eigenen Datei vorliegen. Zudem werden vom Skript noch separat die sog. System-Makros von Passolo vom Repository abgeholt, die für die erfolgreiche Ausführung notwendig sind. Diese müssen vom Buildskript abgeholt werden, da der Pfad zu selbigen in der Passolo-Installation konfiguriert werden muss, aber an dieser Stelle keine Variablen zulässig sind. Deshalb werden sie an einen zentralen Ort geschrieben, damit jeder Job unabhängig darauf Zugriff hat. Mit den in Hudson integrierten Mechanismen können Ordner nur in einen Unterordner des Workspaces des aktuellen Jobs abgeholt werden.
Passolo selber ist nicht für den Aufruf von einer Kommandozeile konzipiert, im automatisierten Modus öffnet sich die graphische Oberfläche. Dieser Modus lässt sich allerdings auch für Hudson nutzen, ohne dass die Oberfläche sichtbar wird. Allerdings lässt sich die Verarbeitung nur über den Umweg eines simplen Ant-Skriptes (Ausschnitt in Listing 2.4) erfolgreich starten. Der Aufruf aus der PowerShell oder der Kommandozeile schlägt

fehl, die Ursache dieses Verhaltens ist noch ungeklärt[1].
Die Vermutung geht dahin, dass die Anzeige von der Kombination aus Ant und Hudson unterdrückt wird - ein Verhalten, was auf dem bisherigen Build-Server, bei dem nur Ant genutzt wird, nicht auftritt.
Im Passolo Job wird zuerst unterschieden, ob es sich um einen Nightly-Build handelt. Nur in diesem Fall wird der Prozess gestartet, andernfalls wird nur eine Meldung ausgegeben und das Skript beendet.

Listing 2.4: Ausschnitt Passolo Buildscript mit Ant

```
1  <target name="hudson">
2    <if>
3      <equals arg1="${NIGHTLY}" arg2="true" />
4        <then>
5          <exec dir="." executable="cmd.exe">
6            <arg line="/c svn update c:\hudson\PassoloStuff\
                 Passolo --username build --password ***** --no-
                 auth-cache --non-interactive --trust-server-cert"/
                 >
7          </exec>
8          <exec executable="C:\Program Files (x86)\PASSOLO 5\psl.
               exe">
9            <arg value="${env.WORKSPACE}\Localization\Passolo\
                 Holos.lpj"/>
10           <arg value="/BATCH"/>
11         </exec>
12         </then>
13     <else>
14       <echo>no nightly, no passolo</echo>
15     </else>
16   </if>
17 </target>
```

InstallShield

InstallShield [9] ist eine weit verbreitete Lösung um Installations-Packete und -Assistenten zu erstellen. *Holometric Technologies* ist im Besitz einer Premier-Lizenz, welche es erlaubt, auf einem Buildserver getrennt die Installationspakete zu erstellen. Der Aufruf des sog. Standalone Builds erfolgt über ein separates PowerShell Skript, da sich der Bauvorgang vom Standardablauf stark unterscheidet.
Da der InstallShield Job der letzte im kompletten Prozess ist, werden am Ende noch

[1] Stand 10. Mai 2010

einige zusätzliche Schritte ausgeführt. Der erzeugte Installer wird mit dem OpenSource Packer 7-Zip [10] gepackt und auf einem Server archiviert, zudem wird die aktuelle Versionen auch auf einen anderen Server kopiert, damit Tester immer eine aktuelle Version zur Verfügung haben. Anschließend wird im Repository ein Tag zum aktuellen Build erzeugt und die aktuelle und nächste Versionsnummer in der Wiki des FogBugz-Bugtrackers hinterlegt, damit bei der Aufnahme von neuen Fehler durch den Support die Nummer der aktuellen Version zur Verfügung steht.
Im InstallShield Job wird zuerst unterschieden, ob es sich um einen Nightly-Build handelt. Nur in diesem Fall wird der Prozess gestartet, andernfalls wird nur eine Meldung ausgegeben und das Skript beendet.

Programm Bibliotheken

Bevor alle anderen Projekte gebaut werden, stehen einige Programm Bibliotheken (eng. Libraries) an. Da an diesen Projekten nur sehr selten Änderungen vorgenommenen werden, wurden sie zu einem Job in Hudson zusammengefasst. Ausschnitt des Skripts in Listing 2.5.
Es kommen weniger Variablen zum Einsatz, da der Name des Projektes nicht mehr aus dem Job-Namen gebildet werden kann, da mehrere Projekte in einem Job gebaut werden. Nach jedem Compiler-Aufruf wird der Exitstatus des letztes Programms überprüft. Sollte dieser einen Wert haben, der auf einen Fehler hin deutet, wird das Script beendet und der Build wird von Hudson als Fehlgeschlagen gekennzeichnet.

Listing 2.5: Ausschnitt Buildscript für Programm Bibliotheken

```
1 vcbuild.exe /r source\HTNlib\HTNlib.vcproj "$env:BUILD_TYPE"
2 if( $? -ne True) { exit 1}
```

weitere Sonderfälle

Für das Projekt „HTTools“ wird noch eine sog. ReleaseID generiert. Anhand dieser wird die korrekte Lizenzierung des Programms überprüft. Die ReleaseID wird für jeden Build von einem eigenen Tool erzeugt und beim Kompilieren in den Programmcode fest integriert, der Abgleich findet später mit einer einfachen Text-Datei statt.

2.3 Erweiterungen

Eine der großen Stärken von Hudson ist seine einfache Erweiterbarkeit durch Plugins. Momentan[2] stehen 300 PlugIns in 19 Kategorien zur Verfügung. Die Bandbreite reicht hier von Erweiterung um zusätzlichen Versionskontrollsystemen, Buildsysteme, Analyse- und Reporttools bis zu Benachrichtigung über den Buildstatus über mehrere Kanäle (Instant Messanger, Twitter, IRC, Ampelanzeigen) und kleinen Spielereien.

2.3.1 C and C++ Code Counter

C and C++ Code Counter (CCCC) [11] ist ein statisches Analysetool, das den Code analysiert und verschiedene Codemetriken berechnet. Der Aufruf des Programms erfolgt über das Standard Buildskript, details in Abschnitt 2.2.1. Da CCCC nicht das rekursive Durchsuchen von Ordnerstrukturen unterstützt, muss jeder Ordner mit Quelldateien explizit angegeben werden, was den Aufruf sehr unübersichtlich macht. Das Programm generiert einen Report im HTML-Format oder eine XML-Datei mit den Ergebnissen. Letztere wird vom PlugIn für Hudson gelesen und die Ergebnisse in die Oberfläche zu jedem Build eingebettet. Zusätzlich können über Graphen die Entwicklung des Programms in Sachen Größe und Komplexität veranschaulicht werden. Da die Analyse nur eine sehr kurze Laufzeit hat, kann sie bei jedem Build neu generiert werden.

2.3.2 Warnings

Compiler können Warnungen bei potentiell unsicherem Code bzw. der Nutzung unsicherer Funktionen und Methoden, Speicherverschwendung durch nicht genutzte Variablen oder anderer nicht offensichtlich kritischer Fehler ausgeben. Trotzdem können sie Fehler verursachen oder sind einfach nur unnötig, speziell wenn sie aus Bequemlichkeit der Entwickler entstehen. Eines der Ziele des Projekt ist es, die Anzahl der Warnungen zu reduzieren; allein das Modul Holos produzierte zu Projektstart über 2500, insgesamt lag die Zahl bei über 7500.
Das Warnings PlugIn zählt die aufgetretenen Warnungen im Buildlog und listet sie detailliert zu jedem Build auf. Über eine Grafik wird zudem der Verlauf der Anzahl an Warnungen visualisiert. Das PlugIn kann bei einer bestimmten Anzahl oder neu hinzugekommenen Warnungen den Status des Builds auf instabil oder fehlgeschlagen setzen. Dies ist allerdings etwas fehleranfällig, da manchmal vom vorherigen Build bekannte Warnings nicht mehr wiedergefunden werden und somit sowohl als behoben als auch als neu erkannt werden.

2.3.3 Continuous Integration Game

Die Motivation der Mitarbeiter ist ein entscheidender Faktor in der Softwareentwicklung. Das sog. Continuous Integration Game soll hier einen Anreiz für die Entwickler bieten, qualitativ hochwertigen Code zu entwickeln. Das Konzept dahinter ist einfach: für jeden

[2]Stand: 11. Mai 2010

neuen, erfolgreichen Build bekommt der Entwickler einen Punkt gutgeschrieben. Sollte ein Build fehlschlagen bekommt er wieder Punkte abgezogen, ein Vorschlag dazu ist 10 Punkte. Instabile Builds geben keine Punkte, ebenso wenn ein fehlerhafter Build auf einen bereits Fehlgeschlagenen folgt. Für eine neue Compiler-Warnung wird ein Punkt abgezogen, für eine behobene ein Punkt gutgeschrieben.
Das PlugIn kann auch die Daten anderer PlugIns auswerten, so werden zusätzlich Punkte für das beheben von Warnungen vergeben bzw. für neu hinzugefügte Punkte abgezogen. Weitere Möglichkeiten wären die Einhaltung von Coding-Style-Kriterien oder dem erstellen von statischen Tests.
Um den Ansporn für die Entwickler zu vergrößern wird ein Wanderpokal für den besten Entwickler der letzten Woche eingeführt, der jeden Freitag neu vergeben wird. Die Rangliste wird in den gleichen Abständen zurückgesetzt, damit sich kein Entwickler einen uneinholbaren Vorsprung erarbeiten kann. Dies würde die Motivation der anderen Mitarbeiter senken da sie keine Chance mehr haben, das Spiel zu gewinnen.

2.3.4 Locks & Latches

Manche Jobs dürfen nicht gleichzeitig mit anderen laufen oder sollen immer parallel zu bestimmten anderen laufen - das Locks & Latches PlugIn ermöglicht dies.
Mit Locks wird sichergestellt, dass bestimmte Hobs nicht parallel laufen. Jedes Lock entspricht einem Token, dass von den Jobs, die dem Lock zugeordnet werden, abwechselnd besetzt und wieder freigegeben wird. Wenn ein Job alle benötigten Tokens hat (er kann auch zu mehreren Locks gehören) kann er gestartet werden.
Latches bewirken das Gegenteil: Sie sollen sicherstellen, dass bestimmte Jobs parallel ausgeführt werden. Diese Funktioninalität ist aber bislang nicht implementiert.[3]

2.3.5 Ansichten

Die graphische Oberfläche von Hudson lässt sich zu einem gewissen Grad persönlich einrichten. Kernstück ist hierbei die Übersicht über alle Jobs, wobei die Standard-Liste schnell unübersichtlich werden kann. Um die Übersicht zu verbessern und wichtige Informationen kompakt dar zu stellen, hat die Hudson-Community einige PlugIns entwickelt. In jeder Ansicht können sowohl die Jobs als auch die Build-Warteschlange und Status der Rechenwerke gefiltert werden, damit bei vielen Jobs insgesamt die Übersicht erhalten bleibt.

Dashboard

Das sog. Dashboard PlugIn (nicht zu verwechseln mit der Startseite von Hudson, die ebenfalls als Dashboard bezeichnet wird) fügt eine modulare Ansicht hinzu. In ihr können Elemete wie Job Listen, Graphen für Tests und Kompilerwarnungen und anderes anhand von vier Positionen (oben, rechts, unten, links) angeordnet werden. Das Dashboard bietet

[3]Stand: 9.9.2010, Version 0.6

eine schnelle Übersicht über den Status der dargestellten Jobs, erfordert allerdings etwas Konfigurationsaufwand.

Radiator

Die sog. Radiator View stellt alle Jobs als farbige Kästen dar, die gleich den Farben über den Status des letzten Builds sind (Blau oder Grün für Erfolgreich, Gelb für Instabil, Rot für Fehlgeschlagen und Grau für Abgebrochen). Zur besseren Visualisierung wird auch die größte geändert, ist der letzte Build erfolgreich wird nur ein kleines, blaues/grünes Kästchen dargestellt, das Kästchen für Instabile Builds ist deutlich größer, ebenso das für Fehlgeschlagene.
Da bei dieser View außer den Kästchen nichts mehr von der restlichen Oberfläche sichtbar ist eignet sie sich nicht für die tägliche Arbeit, sondern für Extreme Feedback Mechanismen, siehe Abschlitt 3.2.

Nested View

Das Nested View PlugIn stellt an sich keine neue Ansicht zur Verfügung. Stattdessen lassen sich damit andere Ansichten schachteln, um z.b. die Jobs logisch zu gruppieren. Jeder Nested View können weitere Views jedes Typs untergeordnet werden, auch eine weitere Nested View. Seit Version 1.2 wird für jede untergeordnete View auch ein Wettericon angezeigt, die den Status aller Jobs in der Untergeordneten View zusammenfasst. So kann man schnell einen Überblick über den Status der untergeordneten Jobs bekommen und feststellen, in welchem Teil des Projektes es Probleme gibt.

2.3.6 Global Stats

Für Administratoren ist Auslastung der Server von großem Interesse. Das Global Stats Plugin kann in mehreren, konfigurierbaren Graphen die Auslastung des Servers in gewählten Zeiträumen darstellen, basierend auf den laufenden Jobs. Außerdem wird visuell dargestellt, wie viele Jobs erfolgreich waren, wie viele instabil oder gescheitert sind oder wie viele manuell abgebrochen wurden.

2.3.7 Monitoring

Das Monitoring-PlugIn basiert auf JavaMeldoy [12] und kann viele Details zur Auslastung, Zugriffsstatistiken und Systemdetails von Java-Server Anwendungen in verschiedenen Graphen darstellen. Zudem können automatisch Statistik-Berichte erstellt werden.

2.4 Versionsnummer

Alle selbst gepflegten Module von Holos verfügen über eine Datei VersionNo.h, die für das erzeugen des InstallShield Paketes und für die Zuordnung der erzeugten Artefakte benötigt wird. Bisher wurde Nachts eine neue Versionsnummer generiert, was aber für die kontinuierliche Integration nicht praktikabel ist. Deswegen wurde für die Versionsnummergenerierung ein neues Schema entworfen. Da die bisherige Erhöhung von einem propitärem Tool vorgenommen wurde, soll im Rahmen des Projektes ein eigenes Programm für diese Aufgabe entwickelt werden.

2.4.1 Schema

Die Versionsnummer von Holos besteht aus vier Stellen: die erste gibt das Hauptrelease an. Die zweite die Unterversion, wobei hier eine gerade Zahl für eine Stabile, eine ungerade für eine Entwicklungsversion steht. Die dritte Stelle gibt die nächtliche Versionsnummer des aktuellen Entwicklgungszyklus an. Auch bei diesen Nummern geben gerade Zahlen eine stabile Version und ungerade eine Entwicklungsversion an. Die vierte Stelle war bisher ungenutzt und soll nun für die fortlaufende Versionsnummer für Builds, die an einem Tag im Zuge der Entwicklung entstehen, genutzt werden. Sie werden jede Nacht auf Null zurückgesetzt und werden für jedes Projekt individuell vergeben.

2.4.2 Anforderungen

Die Hauptaufgabe ist das Einlesen von Textstrings und deren Verarbeitung. Die Datei, in der die Versionsnummer gespeichert wird, muss eine bestimmte Form haben, da sie von InstallShield zur Erzeugung der Versionsnummer für den Installations-Assistenten verwendet wird. Aus dieser Form müssen die Schriftzeichen ausgelesen, in entsprechende Zahlen umgewandelt und entsprechend des Typs des aktuellen Builds inkrementiert werden. Anschließend muss die neue Versionsnummer in der richtigen Form wieder in die Datei zurückgeschrieben werden.
Dabei wird die tägliche Version aus zwei separaten Dateien gelesen, da diese über das komplette Projekt hinweg gleich sein soll. Sie wird nur zu Beginn eines nächtlichen Builds erhöht, zurück in das Repository übertragen und später nur von jedem Job wieder abgeholt.

2.4.3 Branch Konzept

Es gibt drei Branches, auf denen Holos entwickelt wird:

- **Trunk:** Neue Entwicklungen und Fehlerbeseitigung laufen hier ab. Änderungen werden auf die jeweiligen Zweige gemerged.
- **Stable:** Branch aus dem Versionen für Kunden gebaut werden, keine neuen Features, keine Entwicklung, nur das Beheben kritischer Fehler, die vom Trunk hoch gemerged werden. Wird ein neuer Service Pack heraus gebracht, wird der bestehende Stable-Zweig archiviert und durch den bisherigen Unstable ersetzt.

- **Unstable:** Branch, auf dem die Fehlerbeseitigung stattfindet. Beseitigte Fehler werden vom Trunk hoch gemerged. Es wird manuell entschieden, wann der Service Pack fertig ist. Der Unstable Zweig wird dann zum neuen Stable, der bisherige Stable wird archiviert.

Der Grund für dieses System ist, dass kritische Fehler unabhängig von der restlichen Fehlerbeseitigung entfernt werden können, ohne dass man befürchten muss, dass im stabilen Build weitere, nicht vollständig ausgetestete Fehlerbehebungen stören könnten.

2.4.4 Konzeption

Das Programm wird über Kommandozeilen-Parameter gesteuert. Es muss zwischen nächtlichen Builds (nightly build) und während der Entwicklung gestarteten Builds (continuous build) unerschieden werden sowie die Unterscheidung auf welchem Branch der aktuelle Build läuft. Nähere Details dazu siehe Abschnitt 2.4.3. Entsprechend der Parameter werden die Versionsnummer erhöht, dabei können folgende Fälle eintreten:

- **Hilfe:** Aufruf der Hilfefunktion. Es wird eine Textnachricht mit einer Beschreibung des Programms auf der Konsole ausgeben, das Programm wird beendet. Beispiel siehe Abbildung 2.1.
- **Development Continuous Build:** Der häufigste Fall. Die vierte Stelle der Versionsnummer wird um eins erhöht, alle anderen Werte bleiben unangetastet.
- **Development Nightly Build:** Wird jede Nacht gestartet. Die dritte Stelle wird um zwei erhöht, damit der Build immer eine ungerade Zahl hat. Die vierte Stelle wird auf Null gesetzt. Die Versionsnummer des Development-Zweiges hängt nicht mit der des Stable- oder Unstablezweiges zusammen.
- **Unstable Continuous Build:** Sehr häufiger Fall. Tritt ein wenn Änderungen vom Development Zweig auf den Unstable Zweig gemerged werden. Die vierte Stelle der Versionsnummer wird um eins erhöht, alle anderen Werte bleiben unangetastet.
- **Unstable Nightly Build:** Wird jede Nacht gestartet. Die dritte Stelle wird je nach vorherigem Build erhöht (vorher Unstable um zwei, vorher Stable um eins; die Versionsnummer eines Unstable Builds ist immer ungerade). Die vierte Stelle wird auf Null gesetzt.
- **Stable Nightly Build:** Wird wenn benötigt manuell gestartet. Die dritte Stelle wird je nach vorherigem Build erhöht (vorher Unstable um eins, vorher Stable um zwei; die Versionsnummer eines Stable Builds ist immer gerade). Die vierte Stelle wird auf Null gesetzt.
- **Stable Continuous Build:** Dieser Fall darf nicht eintreten, da fortlaufende Builds nur bei der Entwicklung auftreten. Stabile Builds werden von Hand als solche gekennzeichnet bzw. dazu gemacht und dem entsprechend nur manuell gestartet.

Die Entscheidung, von welchem Branch der vorherige Build kam wird anhand der eingelesenen Dateien für die aktuellen Versionsnummern für Stable und Unstable getroffen. Stable-Versionen haben immer gerade Zahlen, Unstable ungerade. Entsprechend ist die Korrektheit der Angaben in den Dateien kritisch. Sie müssen zu Anfang einmal korrekt angelegt werden, später wird das automatisch durch das Programm erledigt.

2.4.5 Implementierung

Das Programm wurde in C mit Anleihen aus den ANSI C++ Standard Bibliotheken geschrieben, als Entwicklungsumgebung diente Microsoft Visual Studio 2005.

Funktionsbeschreibung

Listing 2.6: Prototyp Hauptfunktion

```
1 int main(int argc, char* argv[])
```

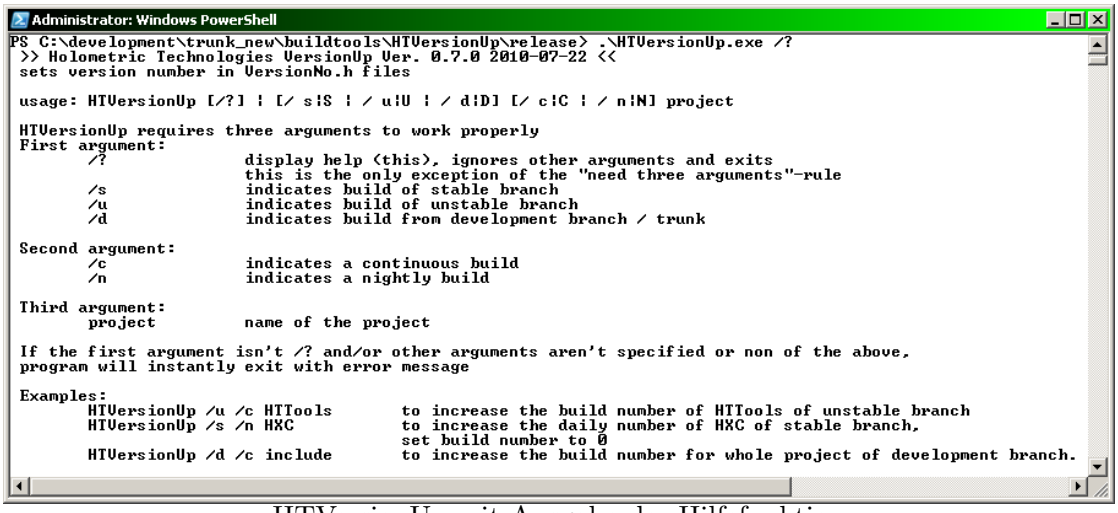
Die Hauptfunktion des Programmes. Hier werden die Eingabeparameter verarbeitet und entsprechend dieser die Funktionen aufgerufen.

Listing 2.7: Prototyp Hilfefunktion

```
1 void print_help();
```

Die Hilfefunktion hat weder Parameter noch einen Rückgabewert. Es werden nur Textstrings auf der Standardausgabe ausgeben und das Programm beendet.

Abbildung 2.1: HTVersionUp Hilfefunktion



HTVersionUp mit Ausgabe der Hilfefunktion

Listing 2.8: Prototyp Logmessage Ausgabe Funktion

```
1 void print_log_message( bool nightly, int branch, bool
    downstream, string project );
```

Bevor das Programm ordnungsgemäß beendet wird, wird eine Nachricht auf der Standardausgabe ausgegeben, die über den Typ des Builds genauer informiert. Diese Information erscheint in den Buildlogs und macht somit eine leichtere Zuordnung des Builds einfacher.

Listing 2.9: Prototyp Funktion zur Verarbeitung von VersionNo.h

```
1 bool VersionNo( string project, int version, bool nightly );
```

Liest und Verarbeitet die Datei VersionNo.h. Bei Nightly Builds wird die dritte Stelle Anhand der übergebenen Variable `version` gesetzt, die vierte Stelle auf 0 gesetzt und in die Datei mit den neuen Werten neu geschrieben. Bei Continuous Builds wird ebenfalls die Datei mit den Werten neu geschrieben, der Wert der vierten Stelle, der vorher ausgelesen wurde, wird allerdings um eins erhöht. Der Rückgabewert entspricht dem Ergebnis der Schreiboperation.

Listing 2.10: Prototyp Funktion um neue Tagesversionsnummer zu schreiben

```
1 bool write_day( int version, string file );
```

Die neue Tagesversionsnummer wird neu in die entsprechende Dateien stable.ver, unstable.ver oder development.ver geschrieben. Der Rückgabewert entspricht dem Ergebnis der Schreiboperation.

Listing 2.11: Prototyp Funktion um Tagesversionsnummer auszulesen

```
1 int read_daily_ver(int branch);
```

Liest die Tagesversionsnummern des aktuellen Branches aus und gibt die höhere von beiden zurück, die für die Weiterverarbeitung verwendet wird.

Listing 2.12: Prototyp Funktion zur Erhöhung der Tagesversionsnummer

```
1 int inc_daily( int version, int branch );
```

Erhöht die Tagesversionsnummer anhand der Angabe, auf welchen Branch der aktuelle Build kommt. Bei den Stable und Unstable Zweigen ist der Zweig des vorherige Builds und aktuelle Zweig entscheidend. Builds vom Stable-Zweig haben immer eine gerade Zahl, Unstable eine ungerade. Die Versionsnummer soll aber trotzdem für beide Zweige fortlaufend sein, weswegen die Unterscheidung notwendig ist. Bei Builds vom Development-Zweig wird genauso verfahren wie bei Unstable. Rückgabewert ist die neue Versionsnummer.

2.5 Administrative Werkzeuge

Um die Administration von Hudson zu verbessern, wurden noch zusätzliche kleine Skripte entwickelt.

2.5.1 Sicherungs Skript

Das für die PowerShell geschriebene Script sichert die wichtigsten Bestandteile des Hudson-Ordners, hauptsächlich die Konfigurationsdateien und Build-Historie. Da alle gesicherten Dateien Textdateien sind, lassen sie sich sehr gut komprimieren. Mit dem Einsatz des LZMA-Algorithmus im Kompressionsprogramm 7-Zip [10] lässt sich so die Größe der Sicherungen um fast 99% verringern.

2.5.2 Servicepack Branch Skript

Durch das Branch-Konzept (Details dazu siehe Abschnitt 2.4.3) ist es nötig, regelmäßig neue Branches zu erstellen. Um die zu automatisieren wurde ein PowerShell-Script erstellt, dass diese Aufgabe übernimmt. Es liest zuerst die bestehenden Branches und erstellt automatisch passende, neue. Abschließend werden an alle Entwickler E-Mails mit dem Pfad des neuen Branches verschickt. Zusätzlich zur PowerShell wird für die Benutzung des Skripte ein Kommandozeilen basierter Subversion Client benötigt.

2.5.3 Versionsnummer in Wiki Skript

Für die effektive Benutzung des FogBugz-Bugtracker ist es von Vorteil, schnell die aktuelle und nächste Versionsnummer parat zu haben. FogBugz bietet dazu eine integriete Wiki, die sich über eine ASP-basierte API ansprechen lässt. Über das PowerShell-Skript wird automatisch nach jedem Nightly-Build die aktuelle Versionsnummer sowie Verweise zu den aktuellen Installationspacketen in die Wiki eingetragen.
Seit Version 0.10.0 des Build-Skriptes für InstallShield ist dieses Skript in das Build-Skript des InstallShield-Installers integriert, die separate Version wird nicht mehr gepflegt.

2.5.4 Entfernen von .svn-Ordnern Skript

An einem Punkt der Erzeugung des Installers verwendet InstallShield den kompletten Ordner mit dem erstellten Programm als Quelle. Durch das Abholen von Dateien vom Subversion-SCM sind auch viele, unerwünschte Ordner mit dem Namen „.svn“ in diesem Ordner, die der Installer mit nimmt. Da diese Ordner nur Verwirrung stiften und bei der Erzeugung des Installers stören, werden sie mit einem PowerShell-Script automatisch entfernt, bevor das Installationspaket erzeugt wird.
Seit Version 0.10.0 des Build-Skriptes für InstallShield ist dieses Skript in das Build-Skript des InstallShield-Installers integriert, die separate Version wird nicht mehr gepflegt.

3 Weiterführendes

Hudson und die kontinuierliche Integration bieten noch einige Erweiterungsmöglichkeiten, die im Rahmen des Projektes aus unterschiedlichen Gründen nicht umgesetzt werden konnten.

3.1 Automatische Tests

Automatische Tests sind ein wichtiger Bestandteil der modernen Softwareentwicklung. Hudson kann die Ergebnisse von externen Testframeworks auswerten und direkt in die Oberfläche einbinden. So können schnell Fehler im Code identifiziert und behoben werden.
Holos wird bereits seit über 20 Jahre entwickelt und ist von der Architektur her nicht für die Einbindung von statischen Tests geeignet. Dazu ist das Programm mit über 500.000 Zeilen Programmcode bereits sehr umfangreich, der Aufwand für die Integration von automatischen Tests währe zu hoch. Ein Versuch, automatische Tests zu implementieren ist bereits am hohen Aufwand und der Komplexität der Applikation gescheitert. Zudem steht bereits fest, dass Holos nicht mehr weiter entwickelt wird. Der notwendige Aufwand ist somit nicht mehr rentabel. Dies ist aber für zukünftige Projekte vorgesehen.

3.2 Extreme Feedback

Entwickler müssen schnell und einfach den momentanen Status des Projektes erfahren. Hudson liefert bereits mit seinem Graphen einige Möglichkeiten der Visualisierung, allerdings sind diese zunächst nur für jeden einzelnen Entwickler sichtbar. Mit den Methoden des sog. Extreme Feedback können alle Entwickler schnell über fehlgeschlagene Builds informiert werden. An Hudson lassen sich z.b. Ampelsysteme anschließen, die den Status des letzten Builds wieder spiegeln. Genauere Informationen, z.b. über Verläufe von Testergebnisse, lassen sich über große Monitore an gut sichtbarer Stelle des Büros verbreiten. Einige Beispiele: [13] [14] [15]

3.3 Verteilen der Dokumentation

Mit Werkzeugen wie Doxygen [16] oder JavaDoc [17] lassen sich Dokumentationen direkt im Quelltext erstellen und automatisiert in unterschiedliche Formen erzeugen, als HTML-Seiten oder pdf-Dokument. Hudson bietet die Möglichkeit, die erstellten Dokumentationen automatisch zu erstellen und zu verteilen, damit alle Beteiligten des Projektes jederzeit die aktuelle Version der Dokumentation einsehen können.

3.4 Cluster

Hudson bietet die Möglichkeit, bei Bedarf weitere Buildprozessoren einfach hinzu zu fügen. Jede Maschine, auf der die benötigte Software installiert ist, lässt sich als sog. Slave verwenden, die vom Masterserver, auf dem Hudson läuft, gesteuert wird. Den Slaves

werden Jobs zugewiesen, sie holen die Quelldateien und liefern nach dem erfolgreichen Bauen der Applikation die Buildartefakte zurück an den Master.
Auf jedem Slave benötigte Software für Holos:

- Java Runtime Environment Version 1.5 oder höher
- Hudson Slave-Dienst
- PowerShell 2.0 für die Ausführung der Buildskripte (In Windows Server 2008 R2 und Windows 7 enthalten)
- Visual Studio 2005 SP1
- C and C++ Code Counter (CCCC) siehe Abschnitt 2.3.1

Nur für Nightly Builds benötigt:

- Apache Ant 1.8.0 für die Ausführung des Passolo Buildskripts
- Passolo 5 ASCII Version mit gültiger Anbindung zum Dongle
- InstallShield 2010 Stand Alone Build
- 7-Zip

Die Cluster von Hudson können auch heterogen organisiert sein, um Applikationen automatisch für unterschiedliche Plattformen und Frameworks kompilieren und testen zu können. Für die Konfiguration bietet Hudson den sog. Matrix-Job an.

3.5 InfraDNA's Certified Hudson CI

Im April 2010 verließ Hudson-Erfinder Kohsuke Kawaguchi Sun Microsystems / Oracle, um seine eigene Firma zu gründen, die sich ganz Hudson widmet, InfraDNA. Neben professionellem Service und Support biete sie seit Juli 2010 [18] auch eine eigene Version von Hudson an. Diese Version erhält Updates weniger schnell als die reguläre Version, da sie auf älteren, aber bewährten Versionen von Hudson basiert. Zudem wurden einige Funktionen, die bisher nur als PlugIn verfügbar waren, direkt integriert, u.a. werden als SCM Git und Mercurial von Haus aus unterstützt, ebenso die Authentifizierung über Active Directory. Ebenfalls enthalten ist eine Lösung für Backups und weitere Funktionen, die der OpenSource-Version fehlen.
Da die Lizensierung dieser Version etwas problematisch ist (sie ist nur zusammen mit einem Servicevertrag erhältlich) und unklar ist, wieviel Mehrwert die kostenpflichtige Version gegenüber der regulären bietet, muss diese Version erst evaluiert werden, was den Zeitrahmen des Projektes aber überstieg.
Nähere Informationen: [19]

4 Appendix

4.1 Administration

Dieser Abschnitt enthält Informationen für die Administration des Servers.

4.1.1 Zugang über Remotedesktop

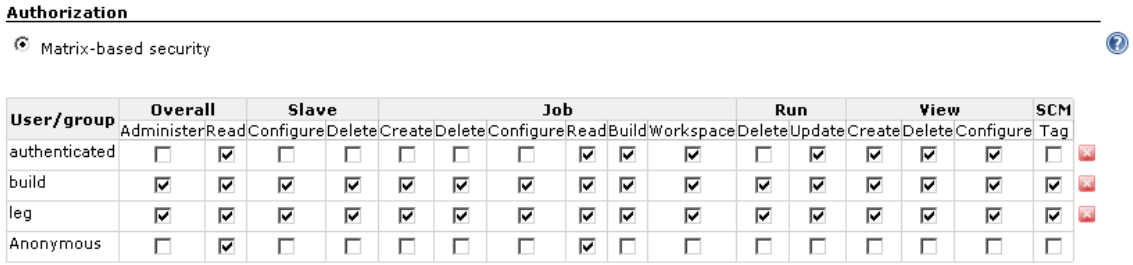
IP (dynamisch): 10.5.92.6
Hostame: build-6

Für Hudson Systemdienst:
Benutzer: build
Passwort: *bekanntes Passwort*

4.1.2 Hudson

- Systemdienst läuft als Benutzer „build“ mit bekanntem Passwort.
- Administratoren mit vollen Rechten auf dem Hudson Masterserver sind die User „build“ und „leg“.
- Authentifizierte Nutzer haben vorwiegend lesende Rechte, sie können zudem neue Builds starten und eigene Views anlegen.
- Anonyme Nutzer haben nur lesende Rechte.

Abbildung 4.1: Hudson Masterserver Rechtekonfiguration



Stand: 10. 9. 2010

4.1.3 Hudson PlugIns

Notwendige für den Build-Prozess und Betrieb

- Hudson Active Directory PlugIn
  http://wiki.hudson-ci.org/display/HUDSON/Active+Directory+Plugin
- Copy Artifact PlugIn
  http://wiki.hudson-ci.org/display/HUDSON/Copy+Artifact+Plugin

- Hudson Locks and Latches PlugIn
  http://wiki.hudson-ci.org/display/HUDSON/Locks+and+Latches+plugin
- Hudson Parameterized Trigger PlugIn
  http://wiki.hudson-ci.org/display/HUDSON/Parameterized+Trigger+Plugin
- Hudson Post build task
  http://wiki.hudson-ci.org/display/HUDSON/Post+build+task
- Hudson Subversion PlugIn (sollte im Standard WAR-Container integriert sein)
  http://wiki.hudson-ci.org/display/HUDSON/Subversion+Plugin

Infrastruktur

- Hudson Global Stats PlugIn
  http://wiki.hudson-ci.org/display/HUDSON/Global+Build+Stats+Plugin
- Monitoring
  http://wiki.hudson-ci.org/display/HUDSON/Monitoring
- Disk Usage
  http://wiki.hudson-ci.org/display/HUDSON/Disk+Usage+Plugin

GUI / Views

- Dashboard View
  http://wiki.hudson-ci.org/display/HUDSON/Dashboard+View
- Green Balls
  http://wiki.hudson-ci.org/display/HUDSON/Green+Balls
- Locale Plugin
  http://wiki.hudson-ci.org/display/HUDSON/Locale+Plugin
- Nested View PlugIn
  http://wiki.hudson-ci.org/display/HUDSON/Nested+View+Plugin
- Hudson Radiator View PlugIn
  http://wiki.hudson-ci.org/display/HUDSON/Radiator+View+Plugin

Buildprozess Optional

- Static Analysis Utilites
  http://wiki.hudson-ci.org/display/HUDSON/Static+Code+Analysis+Plug-ins
- Hudson CCCC PlugIn
  http://wiki.hudson-ci.org/display/HUDSON/CCCC+Plugin

- Hudson Continuous Integration game
  http://wiki.hudson-ci.org/display/HUDSON/The+Continuous+Integration+Game+plugin
- Warnings PlugIn
  http://wiki.hudson-ci.org/display/HUDSON/Warnings+Plugin

4.1.4 Hudson Authentifizierung deaktivieren

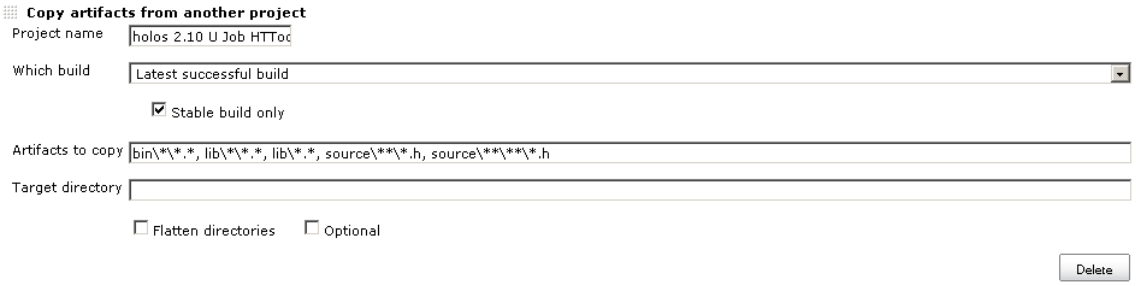
Sollte der Zugang zum System per Administrator nicht mehr möglich sein, lässt sich die Authentifizierung deaktivieren. Dies sollte aber nur als letzte Möglichkeit in Betracht gezogen werden! Bis zur Reaktivierung der Authentifizierung haben **alle** Nutzer volle Rechte, auch unangemeldete!

1. Hudson Dienst über Verwaltungskonsole anhalten
2. im Hudson-Ordner Datei "config.xml" öffnen
3. Zeile `<useSecurity>true</useSecurity>` in `<useSecurity>false</useSecurity>` ändern
4. Dienst neu starten

4.1.5 Abhänigkeiten ändern

Wenn sich Abhänigkeiten zwischen den Projekten ändern, muss der Job in Hudson entsprechend konfiguriert werden, da jeder Job nur die notwendigen Dateien zur Verfügung hat.

Abbildung 4.2: Hudson Job Copy Artifact Beispiel



Stand: 28. 9. 2010

Die Abhäningkeiten werden über das Copy-Artifact PlugIn realisiert. Das Vorgang wird als neuer Buildstep im Job festgelegt und muss bevor das Buildskript aufgerufen wird erfolgen. Es muss er Job, von dem kopiert werden soll sowie die Dateien angebenen werden, Wildcards können genutzt werden. Das „Project name"-Feld schlägt auf Eingabe mehrer Zeichen mögliche Jobs vor. Der Buildstep kann per Drag & Drop in der Reihenfolge verschoben werden.

4.2 Neu Installation

Alle wichtigen Daten von Hudson liegen im Programmverzeichnis `c:\users\build\.¬ hudson`. Das Backup-Script sichert die wichtigesten Bestandteile: Konfigurationdateien, Logs, Artefakte und Build-Historie. Quelltexte und Build-Scripts befinden sich im Repository und werden von diesem geholt, eine manuelle Wiederherstellung ist nicht nötig. Damit Hudson betrieben werden kann, sind einige Programme erforderlich:

1. Java Development Kit (JDK) 1.5 oder höher
2. Visual Studio 2005 SP1
3. Apache Ant 1.8.0
4. Passolo 5
5. InstallShield 2010 Stand-Alone Build
6. C & C++ Code Counter
7. SlikSVN (oder vergleichbarer, Kommandozeilenbasierter Client)
8. 7-Zip

4.2.1 Hudson

Hudson muss als Windows-Systemdienst eingerichtet werden:

1. `hudson.war` von `www.hudson-ci.org` herunterladen
2. Datei nach `c:\hudson` kopieren und per `java -jar hudson.war` starten. Der lokale Server ist unter `http://localhost:8080` verfügbar.
3. Im Verwaltungsmenü (`http://localhost:8080/manage`) „Als Windows Dienst installieren“, als Pfad `c:\hudson` angeben.
4. Im Hudson-Verzeichnis liegt eine Datei `hudson.xml`, über welche die Startparameter für den Windows-Dienst festlegt werden können. Standardmäßig werden Java Anwendungen nur 256MB Speicher zugeteilt, was für Hudson zu wenig ist. Deshalb wurde der Speicher auf 1024MB erhöht. Ausschnitt in Listing 4.1. Übergabe per Parameter im Dienst-Menü ist nicht möglich. Details unter [20].
5. PlugIns per PlugIn Manager (`http://localhost:8080/pluginManager`) Installieren. Liste der Plugins in Abschnitt 4.1.3.

Listing 4.1: Beispiel hudson.xml

```
<service>
  <id>hudson</id>
  <name>Hudson</name>
  <description>This service runs Hudson continuous integration
      system.</description>
  <env name="HUDSON_HOME" value="%BASE%" />
  <executable>java</executable>
  <arguments>-Xrs -Xmx1024m -Dhudson.lifecycle=hudson.lifecycle
      .WindowsServiceLifecycle -jar "%BASE%\hudson.war" --
      httpPort=8080</arguments>
  <logmode>rotate</logmode>
</service>
```

4.2.2 Visual Studio 2005

Die Installation des Visual Studios benötigt zwingend die Installation des Service Pack 1 sowie eines weiteren Updates für Windows Vista, da es ansonsten zu unlogischen Fehlern während des Kompilierens kommt.

4.2.3 Apache Ant

Ant wird zur Ausführung des Build-Scripts für Passolo benötigt. Nach der Installation des Programmpaketes muss die `%PATH%`-Variable um den Pfad zum *bin*-Ordner im Programmverzeichnis ergänzt werden.
Neben der manuellen Installation bietet Hudson die Möglichkeit, Ant automatisch über die Administrationsoberfläche zu installieren. Dies war aber aus bislang noch ungeklärten Gründen nicht erfolgreich.

4.2.4 Passolo

Passolo bentöigt nach der Installation noch etwas Konfiguration.
Zuerst wird eine Verbindung zum USB-Dongle benötigt, das auch über das Netzwerk erreicht werden kann.

1. Passolo (ASCII) im Demo-Modus starten
2. Hilfe → Über Passolo → Dongle
3. Kästchen „Netzwerk Dongle [...]“ anhaken

Da Passolo zudem noch viel mit Makros arbeitet, müssen diese installiert und konfiguriert werden.

1. In Hudson-Ordner einen Ordner „PassoloStuff“ an anlegen

2. per Subversion den Ordner
   `https://repository-7:8443/holos_development/trunk/Localization/Passolo`
   abholen (Ziel Ordner und Pfad sind fest im Build-Script eingebaut)

3. Passolo (ASCII) mit dem Holos-Projekt starten

4. Extras → Optionen → System → Abschnitt „Makros“ → Verzeichnis: `c:\hudson¬\PassoloStuff\Passolo`

5. Extras → Makros → Allgemeine Makros → „Holos Textdateien.bas“ auswählen → System-Makro setzen

6. Extras → System Makro → System Makro starten

Nach der Installation muss der Sentinel Protection Client auf die Version 7.50 updated werden.

4.2.5 InstallShield

Für den InstallShield reicht die StandAlone-Version. Bei der Installation muss aber darauf geachtet werden, dass die Komponenten „Automation Interface“ und „InstallScript Objects Support“ mit installiert werden.

4.2.6 CCCC

Vom C & C++ Code Counter muss nach der Installation das Programmverzeichnis in die `%PATH%`-Variable mit eingetragen werden. Das Programm wird benötigt, da das Plugin nur die erzeugten Reports in die Oberfläche von Hudson einbindet und im Build-Archiv speichert.

4.2.7 SlikSVN

Der kommandozeilenbasierte Subversion Client wird für die Post-Build-Commints sowie für das Passolo-Projekt benötigt, da an diesen Stellen nicht auf den internen Client von Hudson zugegriffen werden kann. Auch hierbei muss der Pfad zum *bin*-Ordner im Programmverzeichnis in die `%PATH%`-Variable eingefügt werden.

4.3 Begriffserklärung

Artefakt Bezeichnung in Hudson für eine von einem Job erzeugte Datei, meist eine ausführbare, binäre Datei oder ein Zwischenformat.

Build Bezeichnung für einen Bauauftrag eines Jobs. Die letzten zehn und der evtl. letzte erfolgreiche davon werden in Hudson behalten, um die Übersichtlichkeit zu wahren.

Branch engl. Zweig

Hauptzweig Zweig in einer Quellcodeverwaltung, auf dem die Entwicklung stattfindet; zentrale Anlaufstelle für alle Entwickler.

Holos Hauptprodukt der Holometric, die mit Hudson gebaut wird. Beschreibung in Abschitt 1.3.2

Hudson Continuous Integration Server, entwickelt bei Sun Microsystems bzw. Oracle. Detailierte Beschreibung in Abschnitt 1.2.7

InstallShield Anwendung zur Erstellung von Installations-Paketen und -Assistenten. Beschreibung in Abschnitt 2.2.1

Job Bezeichnung für ein in Hudson angelegtes Projekt. Für jeden Branch und jedes Projekt von Holos gibt es einen Job, zudem gibt es noch weitere für administrative Jobs.

Integration Bezeichnung für das Zusammenführen von Programmteilen. Früher am Ende der Entwicklung einer Applikation, mit Continuous Integration bei jeder Änderung an den Quelltexten.

Mainline Alternative Bezeichnung für den Hauptzweig

Master Hauptserver eines Hudson-Clusters. Steuert zentral den Bauvorgang, kann Bauvorgänge an Slaves auslagern.

Passolo Programm zu automatischen Erstellung von lokalisierten Software-Versionen. Beschreibung in Abschnitt 2.2.1

Quellcodeverwaltung Systeme zur Verwaltung von Quelltexten. Beispiele: Congruent Version System (CVS), Subversion (SVN), Git, Mecurial.

Repository Bezeichnung für den Ort der Daten in einer Quellcode-Verwaltung (Sourcecode Management SCM)

Slave Knoten innerhalb eines Hudson-Clusters, dem vom Master Aufgaben zugewiesen werden.

Solution Zusammenfassung eines oder mehrerer Projekte in Visual Studio seit Version 2003.

Sourcecode Management engl. Quellcodeverwaltung

Tag Momentaufname des Repositories, die einen bestimmten Entwicklungsstand kennzeichnet. [21] S. 147ff

Trunk In Subversion die Bezeichnung für den Hauptzweig.

Workspace Ort, an dem der Buildvorgang von jedem Job durchgeführt wird.

Zweig Bezeichnet in Subversion und anderen Quellcodeverwaltungen die Abspaltung einer Version des Programms vom Hauptzweig.

Literaturverzeichnis

[1] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for agile software development. http://agilemanifesto.org/ [Last accessed on 2010-04-12].

[2] Martin Fowler. Continuous integration. http://martinfowler.com/articles/continuousIntegration.html [Last accessed on 2010-04-12].

[3] Don Wells. Extreme programming - dedicated intergration computer. http://www.extremeprogramming.org/rules/dedicated.html [Last accessed on 2010-06-29].

[4] Paul M. Duvall, Steve Matyas, and Andrew Glover. *Continuous Integration - Improving Software Quality and Reducing Risks*. Addison-Wesley, 2007.

[5] Kohsuke Kawaguchi. Good bye, sun/oracle. http://weblogs.java.net/blog/kohsuke/archive/2010/04/05/good-bye-sunoracle [Last accessed on 2010-04-13].

[6] Vmware esx and esxi 4.0 comparison. http://kb.vmware.com/kb/1015000 [Last accessed on 2010-04-29].

[7] Slik svn. http://www.sliksvn.com/en/download/ [Last accessed on 2010-05-06].

[8] Sdl passolo overview. http://www.sdl.com/en/sites/sdl-passolo/about/ [Last accessed on 2010-05-11].

[9] Installshield - msi windows installer and installscript msi installation tool - flexera software. http://www.flexerasoftware.com/products/installshield.htm [Last accessed on 2010-05-11].

[10] Offizielle webseite von 7-zip. http://7-zip.org/ [Last accessed on 2010-06-08].

[11] Cccc - a c and c++ code counter - software metrics investigation. http://cccc.sourceforge.net/ [Last accessed on 2010-05-11].

[12] javamelody - project hosting on google code. http://javamelody.googlecode.com/ [Last accessed on 2010-06-30].

[13] Adrian Woodhead. Last.fm - the blog - quality control. http://blog.last.fm/2008/08/01/quality-control [Last accessed on 2010-06-29].

[14] Simon Wiest. „red bear alert!“ - the hudson bear lamps - hudson - hudson wiki. http://wiki.hudson-ci.org/pages/viewpage.action?pageId=20250625 [Last accessed on 2010-06-29].

[15] Hartmut Lang. „watch the bikes!“ extreme feedback with traffic lights - hudson - hudson wiki. http://wiki.hudson-ci.org/pages/viewpage.action?pageId=38633731 [Last accessed on 2010-06-29].

[16] Doxygen. http://www.doxygen.org/ [Last accessed on 2010-06-29].

[17] Javadoc tool home page. http://java.sun.com/j2se/javadoc/ [Last accessed on 2010-06-29].

[18] Announcing beta availability of infradnas certified hudson ci (ichci) server | the hudson company. http://infradna.com/content/announcing-beta-availability-infradna%E2%80%99s-certified-hudson-ci-ichci-server [Last accessed on 2010-09-09].

[19] Infradnas certified hudson ci | the hudson company. http://infradna.com/ichci [Last accessed on 2010-09-09].

[20] java - the java application launcher - arguments. http://download.oracle.com/javase/1.5.0/docs/tooldocs/windows/java.html [Last accessed on 2010-09-14].

[21] C. Michael Pilato, Ben Collins-Sussmann, and Brian W. Fitzpatrick. *Versionskontrolle mit Subversion, 3. Auflage*. O'Reilly, 2009.

Abbildungsverzeichnis

Listings