HANSER

Pragmatisch Programmieren

Projekt-Automatisierung

von Michael Clark

ISBN 3-446-40008-7

Leseprobe Kapitel 2.1

Weitere Informationen oder Bestellung
unter http://www.hanser.de/3-446-40008-8
sowie im Buchhandel

Kapitel 2

Builds in einem Schritt

Lassen Sie uns gleich richtig in die Automatisierung einsteigen. Mit einem Vorgang, den Programmierer immer wieder ausführen: dem Bauen und Testen von Quellcode.

2.1 Software bauen ist wie Brezelbacken

Wenn Sie sich hinsetzen und ein Computerprogramm schreiben, erschaffen Sie etwas Einzigartiges. Es ist ein menschlicher Vorgang, der Elemente der Kunst, des Handwerks, der Wissenschaft und der Technik enthält. Sie können es ruhig versuchen, aber Sie können keine beendete Programmiersitzung später genauso wiederholen. Deswegen ist das *Schreiben* von Software nicht wie der mechanische Prozess des Brezelbackens.

Software bauen hat andererseits sehr viel mit Brezelbacken zu tun. Für Anfänger ist es chaotisch. Sie wollen wirklich nicht wissen, wie Ihr schöner Quellcode in Bits zerlegt wird, damit ein Computer ihn verarbeiten kann. Es ist auch ein wiederholbarer Vorgang: jedes Mal, wenn Sie einen Build ausführen, bekommen Sie eine konsistente Kopie Ihres einzigartigen Programms.

Der Build-Prozess

Um ein Build zu „backen“, brauchen Sie zuerst ein Rezept – üblicherweise als *Build-Datei*. Die Build-Datei zählt alle zum Backen des Builds benötigten Bestandteile auf, inklusive der Quelldateien, Konfigurationsdateien und Bibliotheken von Fremdanbietern. Außerdem beschreibt sie Schritt für Schritt, wie diese Zutaten zu etwas Leckerem zusammengemischt werden. Entweder schreiben wir die Build-Datei von Grund auf neu oder sie wird, wie jedes großartige Rezept, von anderen Programmierern an uns weitergereicht.

Build-Datei

Build-Prozess

Ein *Build-Prozess* ist nichts weiter als eine Reihe von Schritten, die Ihre kreativen Artefakte in auslieferbare Software transformieren. Mit anderen Worten: ein Build-Prozess folgt einfach den Anweisungen in Ihrem sorgfältig vorbereiteten Build-Rezept. Der Prozess nimmt die Zutaten als Eingaben, knetet sie durch und zum Schluss kommen fertige Softwarepakete heraus. Tatsächlich ist es die Build-Maschinerie im Inneren der „Black Box“ in der Mitte von Abbildung 2.1, die uns mehr Zeit für das Schreiben von Software lässt.

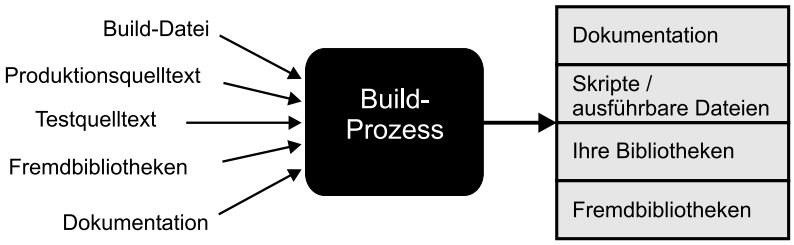

Abbildung 2.1: Der Build-Prozess

Zubereiten von CRISP Builds

Ein automatisierter Build-Prozess versetzt uns in die Lage, unsere Software per Knopfdruck in großen Mengen zu produzieren. Wenn Sie die Art und Weise ändern wollen, wie Sie Software zusammenstellen, ändern Sie das Rezept und drücken den Knopf erneut. Ansonsten können Sie dank der Automatisierung das Rezept ignorieren.

In der Lage zu sein, per Knopfdruck einen Build zu generieren, bedeutet jedoch nicht nur Konsistenz, sondern auch Effizienz. Das heißt, ein in einem Schritt durchführbarer Build-Prozess erlaubt uns Builds auszuführen, die `CRISP` sind:

- vollständig (engl. **C**omplete)
- wiederholbar (engl. **R**epeatable)
- informativ (engl. **I**nformative)
- planbar (engl. **S**chedulable)
- portabel (engl. **P**ortable)

Lassen Sie uns jede Eigenschaft eines `CRISP`-Builds der Reihe nach ansehen.

Vollständige Builds

Vollständige Builds werden von Grund auf neu erstellt und benutzen nur die im Build-Rezept angegebenen Bestandteile. Sie müssen also keine Dateien hinzufügen, bevor oder nachdem der Build ausgeführt wurde. Solche manuellen Schritte sind fehleranfällig und werden gerne mal vergessen. Und offen gesagt, Sie haben bessere Dinge zu tun, als jedes Mal Build-Abhängigkeiten zu beseitigen.

Wenn der Build-Prozess unabhängig ist, können wir ihn automatisieren. Dadurch bekommen wir jedes Mal einen vollständigen Build, wenn der Prozess ausgeführt wird.

Wiederholbare Builds

Der Schlüssel zum Erzeugen wiederholbarer Builds liegt im Speichern der Build-Datei und aller Build-Zutaten in einer Versionsverwaltung, wie z.B. CVS.[1] Dadurch erhalten Sie eine Zeitmaschine, die es Ihnen erlaubt, jede Version der Software durch Auschecken der Dateien per Zeitstempel oder Build-Label zu bauen. Wenn das gleiche Rezept und die gleichen Zutaten eines beliebigen Zeitpunktes gegeben sind, wird der Computer einen identischen Build „backen“.

Ein wiederholbarer Build ist außerdem konsistent. Das bedeutet, Sie können leicht frühere Softwareversionen erneut generieren, wenn Sie von Kunden berichtete Probleme diagnostizieren wollen oder wenn Sie einfach nur einige Schritte zurück gehen wollen.

Informative Builds

Informative Builds liefern wertvolle Informationen, damit wir immer über den Gesundheitszustand unserer Software Bescheid wissen. Als Detektoren für unerwartete Veränderungen spielen automatisierte Tests eine entscheidende Rolle bei diesen Rückmeldungen.

Wenn der Build erfolgreich war, gewinnen wir Sicherheit, dass unsere heutige Arbeit tatsächlich *funktioniert* – der Code konnte kompiliert werden, alle Tests sind erfolgreich durchgelaufen und alle anderen Build-Artefakte wurden ohne Fehler produziert. Mit dem Wissen, dass heute und an jedem folgenden Tag alles funktioniert,

[1] `http://cvshome.org`

müssen wir nicht die Daumen drücken und hoffen, dass alles am Tag der Auslieferung funktionieren wird.

Wenn der Build scheitert, wollen wir schnell wissen warum, damit wir nicht viel Zeit bei der Fehlersuche verbringen. Ein informativer Build sorgt für detaillierte Informationen, die die Ursache jedes Fehlers aufzeigen: eine fehlende Datei, die aber benötigt wird, eine Quelldatei, die sich nicht kompilieren lässt, oder ein fehlgeschlagener Test.

Zeitgesteuerte Builds

Wenn ein Build vollständig und wiederholbar ist, haben wir effektiv einen Build-Prozess, der zeitgesteuert ausgeführt werden kann. Da alle für einen Build benötigten Dateien in der Versionsverwaltung verfügbar sind, kann ein Computer einfach mehrmals am Tag oder auf Befehl neue Builds erzeugen.

Ein zeitgesteuerter Build kann zu einer bestimmten Uhrzeit (z.B. Mitternacht), in regelmäßigen Abständen (z.B. jede Stunde), zu einem Ereignis (wenn wir z.B. Quellcode einchecken) oder einer nach dem anderen stattfinden. Und das Schöne daran ist, wir müssen dafür gar nichts tun. Die Builds werden im Hintergrund ausgeführt, während wir weiter am Quellcode schreiben.

Portable Builds

Nicht zuletzt können portable Builds in jeder vernünftig ausgestatteten Küche gebacken werden. Es kommt auf das Rezept und die Zutaten an und nicht auf die Besonderheiten des Ofens. Wir können ein Build nicht nur bauen, wann wir wollen, sondern auch *wo* wir wollen.

Ein portabler Build bedeutet nicht notwendigerweise, dass Sie eine Unix-Anwendung auch unter Windows bauen können. Aber wenn die Anwendung auf einem Unix-Rechner gebaut werden kann, dann sollte es einfach sein, die Anwendung auch auf *irgendeinem* anderen Unix-Rechner zu bauen. Ebenso sollte das Ausführen eines Builds nicht von einer bestimmten IDE, der IP-Adresse eines Rechners oder dem Verzeichnis abhängig sein, von wo der Build gestartet wird.

Die ganze Diskussion übers Backen muss Sie hungrig gemacht haben. Lassen Sie uns unsere Kochmützen aufsetzen und einen

CRISP-Build backen. Wir beginnen mit dem Definieren der Verzeichnisstruktur unseres Projektes.

Der „Compile“-Knopf ist kein Build-Prozess

So verlockend es auch erscheinen mag, ihn als solchen zu benutzen, der „Compile“-Knopf in Ihrer Lieblings-IDE (Entwicklungsumgebung) ist kein Build-Prozess, der einen „CRISP“-Build erzeugt. Eine IDE ist ein mächtiges Entwicklungswerkzeug, um Code zu kompilieren, seine Struktur zu durchsuchen und sogar für effizientes Code-Refactoring. Aber bei Java-Projekten scheint jeder seine persönliche Lieblings-IDE zu benutzen. Wenn Sie Ihren Build-Prozess hinter dem „Compile“-Knopf der IDE versteckt haben, dann muss sich das gesamte Team auf die *gleiche* IDE einigen. Viel Glück dabei.

Und selbst wenn sich alle auf eine IDE einigen konnten (oder Sie lassen den Gewinner im Armdrücken die Standard-IDE bestimmen), muss jeder im Team seine Installation identisch konfigurieren, damit jeder konsistente Builds erstellen kann. Sie könnten die IDE-Konfigurationsdateien unter eine Versionsverwaltung stellen, damit alle die gleiche Konfiguration benutzen – das erfordert Disziplin, aber es kann funktionieren.

Wie automatisieren Sie nun aber einen zeitgesteuerten Build-Prozess? Setzen Sie einen Programmierer vor die IDE, der immer dann den „Compile“-Knopf drückt, wenn er eine Banane erhält? Gut, wir sind Programmierer, also könnten wir ein Skript schreiben, das die IDE startet und den Knopf für uns drückt. Aber dafür wurde die IDE nicht entworfen, und auf diese Weise einen Build auszuführen, macht die Integration von Build-Planungswerkzeugen schwieriger.

Sie erhalten mehr Flexibilität, wenn Sie den Build-Prozess aus jeder Art von IDE auslagern. Das gibt jedem unabhängig von der Wahl seiner IDE die Freiheit, einen Build manuell auszuführen oder einen unbeaufsichtigten Build zu konfigurieren. Wenn Sie einen anerkannten Build-Prozess haben, opfern Sie nicht die Build-Konsistenz im Namen der Flexibilität. Glücklicherweise nutzt die neue Generation von Java-IDEs zunehmend Standard-Build-Systeme wie zum Beispiel Ant (mehr dazu etwas später). Das erlaubt Ihnen, den selben Build-Prozess sowohl von der IDE aus als auch außerhalb der IDE zu verwenden.