Projektautomatisierung mit Ant

Gunther Popp
(http://www.km-buch.de)

Online-Kapitel zum Buch *Konfigurationsmanagement mit Subversion, Maven und Redmine,* dpunkt Verlag (http://www.dpunkt.de), 2009.

Inhaltsverzeichnis

Vorwort zum Online-Kapitel

Ich bin ein großer Fan von Ant, daher ist es mir sehr schwer gefallen, das entsprechende Kapitel aus der 3. Auflage des Buches zu streichen. Da ich jedoch aus dem Feedback zur 1. und 2. Auflage wusste, dass das Thema *Änderungsmanagement* in der Neuauflage deutlich mehr Raum brauchte und ich Maven in jedem Fall im Buch behalten wollte, führte letztlich kein Weg an der Entscheidung „gegen Ant“ vorbei. Wie auch immer, rein inhaltlich ist das Ant-Kapitel natürlich immer noch aktuell und daher finden Sie den leicht überarbeiteten Text jetzt als Online-Version auf der Web-Seite zum Buch. Die Beispiele aus diesem Kapitel stehen ebenfalls auf der Web-Seite zum Download zur Verfügung (unter dem Punkt *Archiv*).

Viel Spass beim Lesen!

Gunther Popp
Oktober 2009

Projektautomatisierung mit Ant

In diesem Kapitel erstellen wir einen Build-Prozess mit dem Open-Source-Werkzeug Ant. Um nicht mit der Tür ins Haus zu fallen, starten wir mit einem relativ einfachen Skript und erweitern dieses dann Schritt für Schritt.

Wie schon im gedruckten Buch, verwende ich auch in den folgenden Abschnitten das *e2etrace*-Projekt als Grundlage für alle Beispiele. Ich habe es hierzu etwas erweitert, da für einen aussagekräftigen Build-Prozess ein Mindestmaß an Komplexität erforderlich ist. Aufmerksame Leserinnen und Leser werden daher einige neue Module und ein erweitertes Repository entdecken.

Umsetzung eines einfachen Build-Prozesses

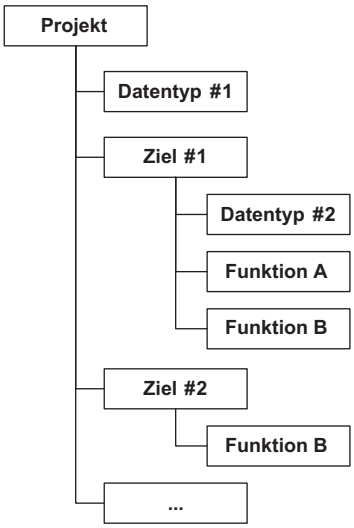
Dreh- und Angelpunkt eines Ant-Build-Prozesses ist das Build-Skript. Dieses heißt standardmäßig `build.xml` und beinhaltet, wie der Name schon andeutet, ein XML-Dokument. Im Skript beschreiben wir, welche Schritte im Laufe des Build-Prozesses in welcher Reihenfolge abzuarbeiten sind. Formuliert werden diese Schritte in einer speziellen Skriptsprache, die im Falle von Ant aus XML-Elementen besteht.

Aufbau des Build-Skriptes

Abbildung 1 zeigt den prinzipiellen, hierarchischen Aufbau eines Ant-Build-Skriptes. Als oberstes Element dient das *Projekt*. Darunter befinden sich die einzelnen Schritte des Build-Skriptes in der Form von *Zielen* (Targets). Beim Aufruf von Ant wird der Name des auszuführenden Ziels als Parameter übergeben. Jedes Ziel besteht wiederum aus einer oder mehreren *Funktionen (Tasks)*. Die Ziele erstellen wir während der Implementierung des Skriptes selbst, die Funktionen werden hingegen von Ant oder externen Bibliotheken bereitgestellt[1]. Man kann die Menge der verfügbaren Funktionen auch als den Befehlsumfang der Skriptsprache betrachten. Die in der Abbildung gezeigten *Datentypen* (Types) stellen demgegenüber die

Variablen der Sprache dar, mit denen die Ziele und Funktionen parametriert werden können. Die Bandbreite der bereitgestellten Datentypen reicht von einfachen Name-Wert-Paaren bis hin zu komplexen Typen, die den Inhalt kompletter Verzeichnisbäume repräsentieren.

Abb. 1
Hierarchische Struktur eines Ant-Build-Skriptes



Festlegung des Referenzverzeichnisses

In Listing 1 ist der erste Entwurf des *e2etrace*-Build-Skriptes zu sehen. Die erste Zeile macht klar, dass es sich um eine XML-Datei handelt. Darauf folgt die Definition des `project`-Elements mit zwei Attributen. Über `name` legen wir den Namen des Projektes fest. Das Attribut `basedir` bestimmt das Referenzverzeichnis für alle noch folgenden Ziele und Funktionen im Skript. Alle Pfadangaben im Skript werden immer im Bezug auf `basedir` interpretiert. Weiterhin kann man im Skript keinerlei Pfade ansprechen, die »oberhalb« von `basedir` liegen. Das Referenzverzeichnis wird in der Regel auf das Wurzelverzeichnis des Projektes gesetzt. In unserem Beispiel ist dies ..., da die Datei `build.xml` im Unterverzeichnis `ant` der Projektstruktur abgelegt wird.

Listing 1
Die »leere Hülle« des Ant-Build-Skriptes für e2etrace

```
<?xml version="1.0"?>
<project name="e2etrace" basedir="..">
    <description>
        Einfaches Build-Skript für e2etrace
    </description>

</project>
```

1. Man kann mit Hilfe der Ant-API auch relativ einfach selbst neue Funktionen schreiben. Dies ist der Grund dafür, dass sehr viele spezialisierte Funktionen als Erweiterungen für Ant existieren. Im Buch gehe ich auf diese Möglichkeit jedoch nicht weiter ein.

Da Ant-Build-Skripte Konfigurationselement sind, muss die Datei `build.xml` nun noch ins Repository aufgenommen werden:

```
>svn add ant\build.xml
A         ant\build.xml

>svn commit -m "Erster Entwurf des Ant-Skriptes"
Hinzufügen     ant\build.xml
Revision 26 übertragen.
```

In Zukunft gehe ich davon aus, dass alle Änderungen am Build-Skript auch im Repository nachgezogen werden, und verzichte auf die Angabe der entsprechenden Subversion-Kommandos.

Quellelemente ermitteln

Für den e2etrace-Build-Prozess sind die Konfigurationselemente *Java-Quelltext* und *JUnit-Tests* relevant. Im Skript müssen die zugehörigen Dateien dieser beiden Elemente selektiert und für die noch folgenden Schritte im Build-Prozess verfügbar gemacht werden. Damit dies gelingt, müssen wir unser Build-Skript zunächst mit der Projektstruktur von *e2etrace* vertraut machen.

Properties definieren

Ähnlich wie in einem »normalen« Programm sollte man auch bei der Implementierung eines Build-Skriptes fest kodierte Werte möglichst vermeiden. Daher werden im Skript alle Konstanten in der Form von sogenannten *Properties* definiert. Properties sind ein einfacher, aber sehr nützlicher Datentyp.[2] Sie bestehen aus einem eindeutigen Namen und einem zugeordneten Wert. Ein einmal zugewiesener Wert kann im Nachhinein übrigens nicht mehr verändert werden.

Geltungsbereich von Properties

Obwohl die hierarchische Struktur des Skriptes rein intuitiv etwas anderes vermuten lässt, gelten Properties unabhängig vom Ort ihrer Definition immer global für das ganze Skript. Wird eine `property` innerhalb eines `target`-Elementes definiert, kann man von einem anderen Build-Ziel aus problemlos darauf zugreifen.

Verfügbare Property-Varianten

Properties können auf sechs unterschiedliche Arten definiert werden. Im Folgenden beschränke ich mich jedoch auf die meiner Ansicht nach wichtigsten Varianten. Wer mehr in die Details einsteigen will, findet in der Ant-Dokumentation alle notwendigen Informationen.

2. Ohne an dieser Stelle unnötige Verwirrung stiften zu wollen, muss ich darauf hinweisen, dass ausgerechnet das `property`-Element in Ant nicht als Datentyp, sondern als Funktion implementiert wurde. Von der Charakteristik her ist es aber ein Datentyp, daher nenne ich es auch in Zukunft so.

Die für uns interessanten Varianten des Property-Elementes lauten wie folgt:

- `<property name=«Name" value=«Wert«/>`
  Legt eine Property mit einem Namen fest und weist ihr einen Wert zu.
- `<property name=«Name" location=«Pfad«/>`
  Hierbei handelt es sich um eine Abwandlung der obigen Variante. Als Wert wird statt einer normalen Konstante eine Pfadangabe erwartet. Das Besondere ist, dass Ant *relative* in *absolute* Pfade konvertiert. Relative Pfade beziehen sich hierbei immer auf das `basedir` des Build-Skriptes (siehe Beispiel unten). Durch die Umwandlung in absolute Pfade ist das Skript unabhängig vom aktuellen Arbeitsverzeichnis.
- `<property file=«Dateiname«/>`
  Lädt alle Properties aus der angegebenen Datei. Der Aufbau der Datei muss dem üblichen Java-Properties-Format entsprechen.
- `<property environment=«Präfix«/>`
  Hierbei handelt es sich um einen Sonderfall, da mit dieser Syntax keine Property, sondern ein Präfix für die Umgebungsvariablen der Shell festgelegt wird. Anschließend kann auf diese Variablen über `Präfix.Name` zugegriffen werden.[3]

Verwendung einer Property im Skript

Nach der Definition kann der Wert einer Property an beliebiger Stelle im Build-Skript mittels `${Name}` abgefragt werden. Diese Platzhalter funktionieren sogar innerhalb von Property-Dateien, die mittels `<property file=«Datei«/>` geladen werden.

Vordefinierte Properties

Neben den selbst definierten gibt es in jedem Build-Skript vordefinierte Properties. Die wichtigste hiervon ist `${basedir}`, die den Wert des `basedir`-Attributs aus dem `project`-Element enthält. Zusätzlich können die folgenden Properties in jedem Skript verwendet werden:

- `${ant.file}`
  Pfad des Build-Skriptes
- `${ant.version}`
  Version von Ant

3. Bei der direkten Verwendung von Umgebungsvariablen im Build-Skript besteht allerdings die Gefahr der Abhängigkeit von einem konkreten Betriebssystem. Man sollte daher möglichst nur selbst definierte oder standardisierte Umgebungsvariablen (wie z. B. `%ANT_HOME%`) verwenden.

- `${ant.project.name}`
  Name des Projektes. Dieser Wert entspricht dem über das Attribut `name` im `<project>`-Element festgelegten Projektnamen.
- `${ant.java.version}`
  Version der JVM, die Ant ausführt
- Alle Java-System-Properties. `${user.home}` liefert also beispielsweise das Benutzerverzeichnis des angemeldeten Anwenders.

Properties für die Projektstruktur von e2etrace festlegen

Nachdem wir jetzt wissen, wie Properties prinzipiell festgelegt werden, können wir unser Build-Skript etwas ausbauen:

Listing 2
Build-Skript mit Properties für die Pfade der Projektstruktur

```
<?xml version="1.0"?>
<project name="e2etrace" basedir="..">
  <description>
    Einfaches Build-Skript für e2etrace
  </description>

  <!-- ==== Definition der Properties ==== -->
  <property name="src"
            location="${basedir}/src"/>
  <property name="target"
            location="${basedir}/target"/>

  <property name="src.java"
            location="${src}/java"/>
  <property name="src.junit"
            location="${src}/junit"/>

  <property name="target.java"
            location="${target}/java"/>
  <property name="target.junit"
            location="${target}/junit"/>
  <property name="target.javadoc"
            location="${target}/javadoc"/>
</project>
```

Das Skript enthält nun Properties für die Quellverzeichnisse der Konfigurationselemente *Java-Quelltext* und *JUnit-Tests*. Im Anschluss werden die Zielverzeichnisse für generierte Artefakte definiert. Für das Konfigurationselement *Java-Quelltext* existieren sogar zwei Zielverzeichnisse, eines für die kompilierten Java-Klassen und eines für die JavaDoc-Dokumentation. Die Zielverzeichnisse liegen allesamt unterhalb des temporären Ordners `target`. Dieser Ordner existiert aktuell in der Projektstruktur noch gar nicht, wir werden ihn etwas später im Build-Skript automatisch erstellen und bei Bedarf auch wieder löschen.

Auslagerung der Properties in eine Datei

Obwohl wir erst ganz am Anfang stehen, wirkt das Skript aus Listing 2 durch die vielen Properties schon etwas unhandlich. Daher empfiehlt es sich, die Properties in eine Datei auszulagern:

Listing 3
Datei build.properties mit den ausgelagerten Property-Definitionen

```
# Properties für das e2etrace-Build-Skript

# Basisverzeichnisse
src=${basedir}/src
target=${basedir}/target

# Quellverzeichnisse der KM-Elemente
src.java=${src}/java
src.junit=${src}/junit

# Verzeichnisse für generierte Artefakte
target.java=${target}/java
target.junit=${target}/junit
target.javadoc=${target}/javadoc
```

Windows-Anwender müssen bei der Angabe von Pfaden in einer Property-Datei übrigens darauf achten, statt des Backslash-Zeichens (\) normale Slashes (/) zu verwenden. Backslash-Zeichen werden in Property-Dateien als Steuerzeichen interpretiert.[4]

Die Datei `build.properties` wird im Verzeichnis `ant` erzeugt und enthält fortan alle Property-Definitionen des Projektes. Im Build-Skript genügt dann eine Zeile, um die Properties aus der Datei zu laden:

Listing 4
Build-Skript mit ausgelagerten Properties

```
<?xml version="1.0"?>
<project name="e2etrace" basedir="..">
  <description>
    Einfaches Build-Skript für e2etrace
  </description>

  <!-- ==== Definition der Properties ==== -->
  <property file="ant/build.properties"/>

</project>
```

Dateien über Filesets auswählen

Die eigentliche Auswahl von Dateien erfolgt in einem Ant-Skript über *Filesets*. Filesets zählen meines Erachtens zu den mächtigsten Funktionen von Ant. Ein `<fileset>`-Element ist im Prinzip wie folgt aufgebaut:

```
<fileset dir=«Verzeichnis« oder file=«Datei«>
  Patternsets
  und/oder
```

4. Alternativ kann man in Property-Dateien auch den doppelten Backslash (\\) verwenden. Dieser wird dann zur Laufzeit als einfacher Backslash interpretiert.

```
    Selektoren
  </fileset>
```

In der Regel setzt ein Fileset auf dem mit dem Attribut `dir` angegebenen Verzeichnis auf und sucht dann unterhalb davon nach Dateien, die bestimmten Kriterien entsprechen. Diese Kriterien werden mit Hilfe von *Patternsets*, *Selektoren* oder einer Kombination aus beiden definiert. Alternativ kann quasi als Abkürzung über das Attribut `file` von Anfang an eine bestimmte Datei festgelegt werden. Die Angabe von weiteren Kriterien erübrigt sich dann natürlich.

Selektion von Dateien mit Patternsets

Ein Patternset-Element selektiert Dateien mit Hilfe von Einschluss- und Ausschlussbedingungen:

```
<patternset>
  <include name=«Suchmuster«/>
  <exclude name=«Suchmuster«/>
</patternset>
```

In den Suchmustern kommen die von den Betriebssystem-Shells bekannten Platzhalter * und ? zum Einsatz. Beispielsweise selektiert das Muster `*.java` alle Dateien mit der Erweiterung `.java`. Ant unterstützt darüber hinaus den Platzhalter **, der komplette Verzeichnisbäume ersetzt. Das Muster `src/**/*.java` findet zum Beispiel alle `*.java`-Dateien in allen Unterverzeichnisbäumen von `src`. Mehrere `<include>`- und `<exclude>`-Elemente in einem Patternset werden jeweils untereinander mit einem logischen *Oder* verknüpft. Zwischen `<include>` und `<exclude>` besteht allerdings eine *Und*-Verknüpfung. Wer kurz darüber nachdenkt, wird diese Vorgehensweise einleuchtend finden.

Verwendung von Selektoren

Patternsets alleine sind schon recht leistungsfähig, die *Selektoren* treiben die Flexibilität sozusagen auf die Spitze. Die folgenden Selektoren unterstützt Ant von Haus aus:

- `<contains>`
  Wählt alle Dateien aus, die einen per Parameter angegebenen Suchtext enthalten.
- `<containsregexp>`
  Dieser Selektor entspricht `<contains>`, es werden aber zusätzlich reguläre Ausdrücke als Suchkriterium unterstützt.
- `<date>`
  Selektiert Dateien abhängig vom Änderungsdatum.
- `<depend>`
  Prüft, ob Dateien in einem Pfad zu einem späteren Zeitpunkt geändert wurden als äquivalente Dateien in einem zweiten Pfad.

- `<depth>`
  Wählt Dateien abhängig von der Verschachtelungstiefe der Unterverzeichnisse aus. Beispielsweise können alle Dateien selektiert werden, die zwischen drei und fünf Ebenen tief im Verzeichnisbaum abgelegt sind.
- `<different>`
  Vergleicht, ähnlich wie `<depend>`, zwei Verzeichnisbäume und selektiert alle unterschiedlichen Dateien. Hierbei kann wahlweise sogar der komplette Dateiinhalt in den Vergleich einbezogen werden.
- `<filename>`
  Dieser Selektor entspricht dem oben beschriebenen Patternset, d. h., er verwendet Einschluss- und Ausschlussmuster.
- `<modified>`
  Dieser recht komplexe Selektor wählt nur geänderte Dateien aus. Als Grundlage für die Entscheidung, ob geändert oder nicht, dient ein intern berechneter Hash-Wert. Der Selektor kann diesen Hash-Wert in einem Cache vorhalten und auf diese Weise Veränderungen zwischen zwei Läufen des Build-Skriptes erkennen.
- `<present>`
  Findet nur Dateien, die ein Äquivalent in einem zweiten Verzeichnisbaum haben. Dieser Selektor ist quasi das Gegenstück zu `<different>`, bietet aber leider nicht dieselbe Flexibilität bei der Auswahl der Vergleichsmethode.
- `<scriptselector>`
  Erlaubt die Entwicklung eines eigenen Selektors in einer Skript-Sprache. Unterstützt werden alle Sprachen, die zum JSR 223[5] oder zum Apache BSF (*Bean Scripting Framework*) kompatibel sind. Hierzu gehören z.B. Groovy, Python, Ruby und natürlich JavaScript.
- `<size>`
  Sucht Dateien abhängig von der Dateigröße.
- `<type>`
  Sucht Dateien abhängig vom Typ (Datei oder Verzeichnis).

---

5. JSR steht für *Java Specification Request*, also die Beschreibung einer neuen Anforderung an die Java-Technologie. In diesem speziellen JSR wird im Prinzip eine Schnittstelle beschrieben, die Skript-Sprachen implementieren müssen, um problemlos mit einem Java-Programm zusammenarbeiten zu können. Ab Java 6 wird der JSR 223 standardmäßig unterstützt.

Verknüpfung von Suchbedingungen mit Containern

Zusätzlich zu den oben genannten Selektoren kennt Ant *Container*, welche die Ergebnisse mehrerer Selektoren über Operatoren miteinander verknüpfen. Unterstützt werden hierbei sowohl logische Operatoren (`<and>`, `<or>`, `<not>` etc.) als auch eher exotische Verknüpfungen wie z. B. `<majority>`. Letzterer liefert als Endergebnis nur diejenigen Dateien, die von der Mehrzahl der im Container zusammengefassten Selektoren ausgewählt werden. Wer nun wirklich noch einen Anwendungsfall hat, der von den mitgelieferten Selektoren nicht abgedeckt ist, kann auch eigene Selektoren entwickeln und in das Skript einbinden.

Filesets für das Beispielprojekt

Listing 5 zeigt die Filesets für die beiden Konfigurationselemente *Java-Quelltext* und *JUnit-Tests* des Beispielprojekts:

Listing 5
Filesets für das e2etrace-Build-Skript

```
<!-- ==== Selektion der KM-Elemente ==== -->
<!-- Java-Quelltext -->
<fileset id="JavaQuelltext" dir="${src.java}">
  <include name="**/*.java" />
</fileset>

<!-- JUnit-Tests -->
<fileset id="JUnitTests" dir="${src.junit}">
  <include name="**/*Test.java" />
</fileset>
```

Das Fileset für Java-Quelltext startet im Java-Quelltext-Verzeichnis und schließt alle Dateien mit der Endung `.java` in allen Unterverzeichnissen ein. Für JUnit-Tests wird mit einem angepassten Suchmuster und Ausgangsverzeichnis ähnlich verfahren.

Die in Listing 5 gezeigte Kurzform zur Definition eines Filesets verzichtet auf die Angabe der `<patternset>`-Elemente oberhalb und unterhalb der Einschlusskriterien. Man spart sich damit zwei Zeilen, kann aber dafür keine zusätzlichen Selektoren mehr angeben.

Wiederverwendung von Filesets

Im Beispiel verwende ich außerdem das optionale `id`-Attribut des `<fileset>`-Elements. Über die `id` kann ein einmal definiertes Fileset im Skript an beliebiger Stelle wieder verwendet werden. Alternativ könnte man auf die `id` auch verzichten und das Fileset einfach direkt an der Stelle definieren, an der es benötigt wird. Meiner Ansicht nach ist jedoch einfach sauberer, die Filesets zentral festzulegen. Dadurch werden eventuelle Fehler durch doppelt definierte und leicht unterschiedliche Filesets von vornherein vermieden.

Standard-Ausschlusskriterien (Default Excludes)

Eine Reihe von Dateien und Verzeichnissen werden von Ant standardmäßig aus jedem Fileset ausgeschlossen. Hierzu gehören z. B. auch die `.svn`-Verzeichnisse in einem Subversion-Arbeitsbereich. Die vollständige Liste dieser *Default Excludes* sowie die Möglichkeiten, diese Liste zu verändern, können unter [URL: AntDefaultExcludes] nachgeschlagen werden.

Projektexterne Dateien einbinden

Bisher haben wir uns ausschließlich mit den Elementen des *e2etrace*-Projektes selbst beschäftigt und alle projektexternen Dateien außer Acht gelassen. In der Praxis kommt allerdings kaum ein Projekt ohne externe Bibliotheken und Frameworks aus, schließlich will man nicht andauernd das Rad neu erfinden. Auch *e2etrace* baut auf zwei »Fremdbibliotheken« auf. Für die Ausgabe von Log-Meldungen greift es auf das commons-logging-Paket[6] des Jakarta-Projekts der Apache Software Foundation zurück. Weiterhin verwende ich das JUnit-Framework zur Erstellung und Durchführung der Modultests.

Prinzipielle Vorgehensweise

Beide Bibliotheken tauchen in der Liste der Konfigurationselemente von *e2etrace* bisher nicht auf und sind auch nicht Bestandteil des Repositorys. Allerdings benötigen wir zweifelsohne beide Produkte, um *e2etrace* überhaupt erstellen zu können. Wäre es daher nicht besser, auch die externen Bibliotheken als Konfigurationselemente zu identifizieren und im Repository zu verwalten?

Leider lässt sich diese Frage nicht eindeutig beantworten. Neben den erwähnten Bibliotheken benötigen wir durchaus noch andere Werkzeuge zur Erstellung von *e2etrace*. Beispielsweise ist ein installiertes JDK notwendig, denn dieses beinhaltet den Java-Compiler. Das JDK setzt wiederum eine bestimmte Betriebssystemversion voraus, die unter Garantie nicht im Repository verwaltet werden kann. Das Ziel, die Nachvollziehbarkeit des Build-Prozesses alleine mit dem Repository zu gewährleisten, kann also von vornherein nicht erreicht werden.

Ich persönlich schließe daher in der Regel alle externen Werkzeuge und Bibliotheken aus der Liste der Konfigurationselemente und damit auch von der Verwaltung im Repository aus. Im Fall von e2etrace bedeutet dies, dass JUnit und commons-logging separat installiert werden müssen und nicht Bestandteil der Projektstruktur werden.

---

6. *http://jakarta.apache.org/commons/logging*

Um die Nachvollziehbarkeit trotzdem zu gewährleisten, lege ich in »echten« Projekten für jedes Release des Produktes ein separates Archiv auf DVD mit der Entwicklungsumgebung (Ant, JDK, IDE, externe Bibliotheken etc.) an. Im Buch werde ich auf diesen separaten Schritt jedoch nicht eingehen.

Einbindung in das Build-Skript

Im Build-Skript benötigen wir nun eine Referenz auf die einzubindenden externen Bibliotheken. Es gibt mehrere Möglichkeiten, dies mit Ant umzusetzen:

1. Die Installation der externen Bibliotheken erfolgt auf allen Entwicklerrechnern in denselben, fest vorgegebenen Pfaden. Im Skript werden diese Pfade dann über eine Property definiert und verwendet.
2. Jeder Entwickler kann eine persönliche Property-Datei in einem vorgegebenen Verzeichnis anlegen. Die Installationspfade der externen Bibliotheken werden in diese Datei eingetragen und vom Build-Skript geladen.
3. Die Installationspfade werden als Umgebungsvariablen in der Shell definiert. Das Skript verwendet direkt diese Variablen als Properties.

Für *e2etrace* wähle ich eine Kombination aus Variante 2 und 3. Die Installationspfade der JUnit- und commons-logging-Bibliotheken werden von jedem Entwickler in einer persönlichen Property-Datei eingetragen (siehe Listing 6). Den Dateinamen und den Pfad dieser Property-Datei hinterlegt jeder Entwickler in einer Umgebungsvariablen mit dem Namen `E2E_PERSONAL_PROPERTIES`.

```
# Entwicklerspezifische Build-Properties für
# das e2etrace-Build-Skript

# commons.logging-Installationspfad
loggingInstall=D:/apache/commons-logging-1.0.4

# Junit-Installationspfad
junitInstall=D:/java/junit4.3.1
```

Listing 6
Entwicklerspezifische Property-Datei mit den Installationspfaden externer Bibliotheken

Im Build-Skript wird die persönliche Property-Datei mit den folgenden Zeilen geladen:

```
<property environment="os"/>
<property file="${os.E2E_PERSONAL_PROPERTIES}" />
```

Listing 7
Laden der persönlichen Property-Datei im Skript

Die erste Zeile definiert ein Präfix für die Umgebungsvariablen des Betriebssystems. Anschließend wird die über die Umgebungsvariable

%E2E_PERSONAL_PROPERTIES% festgelegte Property-Datei geladen. Die eigentliche Einbindung der externen Bibliotheken in das Skript erfolgt über die Definition eines <path>-Elements:

Listing 8
Einbindung der externen Bibliotheken in das Skript

```
<!-- ==== Externe Bibliotheken einbinden ==== -->
<path id="ExternalLibs">
  <pathelement location="${loggingInstall}/commons-
                         logging.jar"/>
  <pathelement location="${junitInstall}/junit-4.3.1.jar"/>
</path>
```

Ein <path>-Element umfasst, ähnlich wie ein Fileset, eine beliebige Anzahl von Dateien. Zusätzlich zu den in Listing 8 gezeigten <pathelement>-Subelementen kann man auch Filesets zur Auswahl der Dateien verwenden. Wir werden den <path> aus dem Listing später als *Java-CLASSPATH* verwenden.

Weiterhin offene Punkte

Prinzipiell könnte man das Problem externer Bibliotheken damit als erledigt betrachten, doch bei genauer Betrachtung bleiben einige Fragen ungelöst. So muss beispielsweise jedes Teammitglied manuell die richtigen Versionen der externen Bibliotheken installieren. Erfolgt ein Update einer externen Bibliothek, muss dieses wiederum auf allen Entwicklerrechnern nachvollzogen werden. Das Skript hat zudem keine Möglichkeit, falsche Bibliotheksversionen zu erkennen und durch einen Fehler zu quittieren. Mit entsprechendem Aufwand könnte man ein Build-Skript sicherlich um derartige Funktionalität erweitern. Doch eigentlich würde man sich ein Build-Werkzeug wünschen, das mit externen Bibliotheken von vornherein umgehen kann. Dies ist übrigens einer der Gründe, warum Maven entwickelt wurde.

Produkt erstellen

Laut der Beschreibung des Konfigurationselements *Java-Quelltext* sind zwei Schritte zur Erstellung von *e2etrace* notwendig. Einmal muss der gesamte Quelltext der *e2etrace*-Bibliothek mit dem Java-Compiler übersetzt werden. Darüber hinaus wird aus dem Quelltext auch die Dokumentation von *e2etrace* mit Hilfe des JavaDoc-Generators erzeugt. In der Praxis wird die Übersetzung des Quelltextes vermutlich wesentlich häufiger durchgeführt als die Generierung der JavaDoc-Dokumentation. Damit die beiden Schritte separat ausgeführt werden

können, realisieren wir sie im Skript als eigenständige Ziele. Bevor wir mit dem Ziel zum Aufruf des Java-Compilers beginnen, müssen zunächst die noch fehlenden Verzeichnisse für die generierten Artefakte erzeugt werden.

Erzeugung der Zielverzeichnisse

Alle generierten Dateien werden vom Build-Skript unterhalb des Verzeichnisses `target` abgelegt. Dieses Verzeichnis hat rein temporären Charakter, d. h., es wird durch das Build-Skript erstellt und bei Bedarf auch wieder gelöscht.

Die Namen der Unterverzeichnisse von `target` haben wir bereits als Properties definiert. Wir müssen dem Skript nun lediglich ein Ziel hinzufügen, das diese Verzeichnisse anlegt:

Listing 9
Ziel zum Anlegen der Zielverzeichnisse

```
<!-- ==== Vorbereitung ==== -->
<target name="prepare"
        description="Vorbereitung und Initialisierung">
  <mkdir dir="${target.java}"/>
  <mkdir dir="${target.junit}"/>
  <mkdir dir="${target.javadoc}"/>
</target>
```

Über das Attribut `name` wird die Bezeichnung des `<target>`-Elements festgelegt. Wenn wir das Ziel ausführen wollen, übergeben wir Ant diesen Namen als Parameter. Mit dem Attribut `description` kann optional eine Beschreibung des Ziels erfolgen. Ant gibt diese Beschreibung aus, wenn beim Aufruf die Option `–projecthelp` angegeben wird.

Zur Implementierung des Ziels verwenden wir die Funktion `<mkdir>`. Diese erwartet als Parameter einen Verzeichnisnamen. Existiert dieses Verzeichnis nicht, wird es von `<mkdir>` neu angelegt. Die Funktion kann verschachtelte Verzeichnisse direkt erzeugen, es ist also nicht notwendig, den Ordner `target` vor den Unterverzeichnissen separat anzulegen.

Test des prepare-Build-Ziels

Um das neue Ziel zu testen, wechseln wir im Arbeitsbereich in das Verzeichnis `ant` und führen das Skript erstmals aus:

```
> cd ant
> ant prepare
Buildfile: build.xml

prepare:
    [mkdir] Created dir:
            D:\Projekte\e2etrace\trunk\target\java
    [mkdir] Created dir:
            D:\Projekte\e2etrace\trunk\target\junit
```

```
    [mkdir] Created dir:
            D:\Projekte\e2etrace\trunk\target\javadoc

BUILD SUCCESSFUL
Total time: 0 seconds
```

Da wir den Standardnamen `build.xml` für das Build-Skript verwendet haben, können wir beim Aufruf von Ant auf die Angabe eines Dateinamens verzichten. Wer einen anderen Dateinamen bevorzugt, kann diesen über die Option `–f` festlegen.

Aufruf des Java-Compilers

Zum Aufruf des Java-Compilers verwenden wir die `<javac>`-Funktion. Die Beschreibung aller Optionen und Varianten dieser Funktion füllt in der Ant-Dokumentation mehrere Seiten. Eine sinnvolle Minimalversion der Funktion kommt allerdings mit nur zwei Attributen aus:

```
<javac srcdir=«Quelle« destdir=«Ziel« />
```

Inkrementeller Build

Ant sucht daraufhin alle Java-Quellcodedateien in der Verzeichnisstruktur unterhalb von `Quelle`. Vor dem Aufruf des Compilers prüft Ant für jede Quellcodedatei, ob schon eine korrespondierende `.class`-Datei in der Struktur unterhalb von `Ziel` existiert. Kompiliert werden dann nur die Quellen, für die keine oder nur veraltete Zieldateien existieren. Man bezeichnet diesen Mechanismus auch als *inkrementellen Build*.

Für das *e2etrace*-Skript ist diese einfache Variante von `<javac>` im Prinzip ausreichend. Allerdings müssen wir zusätzlich die externen Bibliotheken in den `CLASSPATH` des Java-Compilers aufnehmen. Listing 10 zeigt den vollständigen Aufruf der `<javac>`-Funktion innerhalb des neu angelegten Build-Ziels `compile`.

Listing 10
Ziel zum Kompilieren des Java-Quelltextes

```
<!-- ==== Java-Compiler aufrufen ==== -->
<target name="compile"
        depends="prepare"
        description="Kompiliert den Java-Quelltext und
                     die JUnit-Tests">
  <!-- Java-Quelltext kompilieren -->
  <javac srcdir="${src.java}" destdir="${target.java}">
    <classpath refid="ExternalLibs" />
  </javac>

  <!-- JUnit-Tests kompilieren -->
  <javac srcdir="${src.junit}" destdir="${target.junit}">
    <classpath>
      <pathelement location="${target.java}" />
      <path refid="ExternalLibs"/>
    </classpath>
```

```
    </javac>
  </target>
```

Reihenfolge der Build-Ziele im Skript festlegen

Im Listing habe ich bei der Definition des `<target>`-Elements das Attribut `depends` angegeben. Mit Hilfe von `depends` werden Abhängigkeiten zwischen den einzelnen Zielen eines Build-Skriptes festgelegt. Ant wertet diese Abhängigkeiten aus und ermittelt daraus die Reihenfolge, in der die Ziele abgearbeitet werden. So wird beispielsweise in unserem Skript in Zukunft immer das Ziel `prepare` vor `compile` ausgeführt. Auf diese Weise ist sichergestellt, dass die Zielverzeichnisse für den Java-Compiler bei der Ausführung von `compile` bereits angelegt sind.

Separate Kompilierung von Quellcode und Modultests

Aus dem Listing ist auch ersichtlich, dass der Java-Compiler zur Erstellung von *e2etrace* zwei Mal aufgerufen werden muss. Im ersten Durchgang wird der eigentliche *e2etrace*-Quelltext übersetzt. Neben den Quell- und Zielverzeichnissen wird der Funktion `<javac>` über das geschachtelte Element `<classpath>` eine Referenz auf die externen Bibliotheken übergeben. Hierzu wird dem Attribut `refid` der `id`-Bezeichner des in Abschnitt definierten `<path>`-Elements zugewiesen.

Der zweite Durchlauf kompiliert den Quelltext der JUnit-Tests. In diesem Fall müssen zusätzlich zu den externen Bibliotheken auch die eben erzeugten *e2etrace*-Klassen in den `<classpath>` aufgenommen werden. Dies ist notwendig, da die JUnit-Tests direkt auf den *e2etrace*-Klassen aufsetzen.

Ausführung des compile-Build-Zieles

Der folgende Aufruf von Ant testet das neu erstellte Ziel. In der Ausgabe ist zu erkennen, dass das Ziel `prepare` vor `compile` ausgeführt wird. Da die Zielverzeichnisse schon beim vorherigen Aufruf von Ant erstellt wurden, passiert dieses Mal nichts. Die Funktion `javac` ruft dann, wie erwartet, den Java-Compiler für den *e2etrace*-Quelltext und für die JUnit-Tests aus.

```
> ant compile
Buildfile: build.xml

prepare:

compile:
    [javac] Compiling 14 source files to
            D:\Projekte\e2etrace\trunk\target\java
    [javac] Compiling 5 source files to
            D:\Projekte\e2etrace\trunk\target\junit

BUILD SUCCESSFUL
Total time: 2 seconds
```

Generierung der JavaDoc-Dokumentation

Ant stellt auch für die Erzeugung der JavaDoc-Dokumentation eine spezielle Funktion bereit. Die Implementierung eines entsprechenden Ziels im Skript ist daher keine große Sache:

Listing 11
Build-Ziel zur Generierung der JavaDoc-Dokumentation

```
<!-- ==== JavaDoc generieren ==== -->
<target name="doc"
        depends="prepare"
        description="Erstellt die JavaDoc-Dokumentation">
  <javadoc destdir="${target.javadoc}">
    <fileset refid="JavaQuelltext"/>
    <classpath>
      <path refid="ExternalLibs" />
    </classpath>
  </javadoc>

</target>
```

Die Quelldateien werden der `<javadoc>`-Funktion in Form eines Filesets übergeben. Da wir bereits ein globales Fileset für den Java-Quelltext definiert haben, wird dieses im Beispiel mit Hilfe des `refid`-Attributes referenziert.

Erzeugte Dateien löschen

Ein Build-Skript sollte in der Lage sein, alle von ihm verursachten Veränderungen an der Projektstruktur auch wieder rückgängig zu machen. In unserem Beispiel muss hierzu das Verzeichnis `target` samt aller enthaltenen Unterverzeichnisse und Dateien gelöscht werden. Diese Aufgabe übernimmt das Build-Ziel `clean`:

Listing 12
Löschen aller generierten Dateien mit dem Build-Ziel clean

```
<!-- ==== Generierte Dateien entfernen ==== -->
<target name="clean"
        description="Entfernt alle generierten Dateien
                     und Verzeichnisse">
  <delete dir="${target}" />

</target>
```

Wie das Beispiel zeigt, kann die Funktion `<delete>` neben einzelnen Dateien auch komplette Verzeichnisbäume löschen. Sie sollte daher mit der bei Löschbefehlen üblichen Umsicht eingesetzt werden.

Produkt prüfen

Für den Anfang verlassen wir uns zur Prüfung von *e2etrace* ausschließlich auf die JUnit-Modultests. In Abschnitt führen wir dann zusätzlich ein Werkzeug zur statischen Quelltextanalyse ein. Ant unterstützt die

Durchführung von JUnit-Tests mit einer entsprechenden Funktion. Bevor wir diese einsetzen können, muss allerdings die Ant-Installation angepasst werden.

Einbindung von JUnit in das Skript

Kernfunktionen und optionale Funktionen

Wer schon einen Blick in die Ant-Dokumentation geworfen hat, dem ist sicherlich die Unterteilung in *Kernfunktionen* (Core Tasks) und *optionale Funktionen* (Optional Tasks) aufgefallen. Funktionen aus der zuletzt genannten Gruppe verwenden in der Regel zusätzliche externe Bibliotheken, die nicht Bestandteil von Ant sind. Die `<junit>`-Funktion fällt in die Kategorie der optionalen Funktionen und benötigt zur Ausführung Zugriff auf die JUnit-Bibliothek (`junit-4.3.1.jar`).

Bei früheren Ant-Versionen war letztlich eine Anpassung der lokalen Umgebung notwendig, um Ant diesen Zugriff zu ermöglichen[7]. Dieses Manko wurde in Version 1.7 (endlich!) behoben. Jetzt reicht es aus, wenn die JUnit-Bibliothek im `<classpath>`-Element der `<junit>`-Funktion referenziert wird. Dies ist in unserem Beispiel gewährleistet, da wir `junit-4.3.1.jar` schon als externe Bibliothek in das Skript aufgenommen haben.

Durchführung der Modultests

Das Build-Ziel zur Durchführung der JUnit-Tests ist in Listing 13 dargestellt.

Listing 13
Build-Ziel zur Durchführung der JUnit-Tests

```
<!-- ==== Modultests ausführen ==== -->
<target name="test"
        depends="compile"
        description="Führt alle Modultests aus">
  <delete dir="${target.testreports}" />
  <mkdir dir="${target.testreports}" />

  <junit printsummary="yes">
    <classpath>
      <pathelement location="${target.java}" />
      <pathelement location="${target.junit}" />
      <path refid="ExternalLibs" />
    </classpath>

    <batchtest todir="${target.testreports}">
      <fileset refid="JUnitTests" />
    </batchtest>
    <formatter type="plain" />
```

7. Konkret musste `junit-4.3.1.jar` entweder in den `CLASSPATH` der Shell aufgenommen oder in das Unterverzeichnis `lib` des Ant-Installationsverzeichnisses kopiert werden.

```
    </junit>
  </target>
```

Über das depends-Attribut stellen wir zunächst sicher, dass vor der Ausführung der Tests wirklich alle Quellen kompiliert werden. In den nächsten beiden Zeilen wird das Verzeichnis für die Testreporte bei jedem Aufruf des Ziels neu erstellt. Die entsprechende Property target.testreports haben wir noch nicht definiert. In der Datei build.properties muss daher die folgende Zeile eingefügt werden:

```
target.testreports=${target}/testreports
```

Der eigentliche Aufruf von JUnit erfolgt über die <junit>-Funktion. Mit dem Attribut printsummary fordern wir die Ausgabe einer kurzen Zusammenfassung nach der Ausführung der Tests an. Das geschachtelte <classpath>-Element legt den *CLASSPATH* für die Ausführung der Tests fest. Im Beispiel besteht dieser aus den *e2etrace*-Klassen, den eigentlichen JUnit-Testfällen und den externen Bibliotheken, die auch eine Referenz auf junit-4.3.1.jar beinhalten.

Auswahl und Ausführung der Testfälle

Mit dem Subelement <batchtest> werden die Tests dann schließlich ausgeführt. Das Element sammelt zunächst alle über das Fileset ermittelten Testfälle ein, führt einen Test nach dem anderen aus und erzeugt pro Testfall einen Testreport. Die Reporte werden in das mit dem Attribut todir angegebene Verzeichnis geschrieben. Wenn ein Test fehlschlägt, sind die Details über Ort und Ursache des Fehlers im Report zu finden.

Formatierung der Testreporte

Über das Subelement <formatter> kann man das Format der Testreporte festlegen. Im Beispiel wird der Formatter plain verwendet. Dieser erzeugt Reporte in Form einfacher ASCII-Dateien. Für Entwickler sind diese Reporte ausreichend, zur Präsentation auf einer Projekt-Homepage eignen sie sich aber eher nicht. Etwas später werden wir daher mit Hilfe eines XML-Formatters optisch ansprechende Testreporte erzeugen.

Umgang mit fehlerhaften Tests

Die Ausführung des Ziels test könnte beispielsweise folgendes Ergebnis liefern:[8]

```
> ant test
prepare:
compile:
test:
 [delete] Deleting directory (...)\testreports
```

8. Um Platz zu sparen, habe ich die Originalausgabe von Ant etwas gekürzt.

```
    [mkdir] Created dir: (...)\testreports
    [junit] Running e2etrace.timer.DefaultTimerTest
    [junit] Tests run: 1, Failures: 0, Errors: 0
    [junit] Running e2etrace.trace.DefaultTraceSessionTest
    [junit] Tests run: 3, Failures: 1, Errors: 0
    [junit] Test DefaultTraceSessionTest FAILED

BUILD SUCCESSFUL
Total time: 3 seconds
```

Die Ausgabe zeigt einen offenkundigen Widerspruch. Ant meldet `BUILD SUCCESSFUL`, obwohl die Funktion `junit` einen Test als `FAILED` kennzeichnet. Wie ist dieses Ergebnis zu bewerten?

Ignorieren fehlerhafter Testfälle

Aus der Sicht von Ant ist die Ausführung eines Build-Skriptes dann erfolgreich, wenn keines der Ziele einen Fehler meldet. Im Beispiel ist demzufolge im Build-Ziel `test` kein Fehler aufgetreten. Die Funktion `<junit>` hat den fehlgeschlagenen Test schlicht ignoriert. Im Projektalltag hat dies zur Konsequenz, dass das Produkt *e2etrace* trotz fehlerhafter Unit-Tests erstellt werden kann. Dieses Verhalten ist gewollt und entspricht der Standardeinstellung von Ant. Selbstverständlich kann `<junit>` das Skript bei fehlerhaften Tests auch abbrechen. Hierzu muss lediglich das Attribut `haltonfailure=yes` angegeben werden.

Nun stellt sich die Frage, ob wir für das *e2etrace*-Skript mit dem Standardverhalten einverstanden sind oder eher die strikte Variante bevorzugen, die das Produkt nur bei 100% fehlerfreien Tests erstellt. Für Entwickler ist die zuletzt genannte Variante meiner Erfahrung nach nicht praktikabel. Ein Beharren auf jederzeit fehlerfreie Unit-Tests führt oft dazu, dass einzelne Tests »mal kurz« auskommentiert werden – und später komplett in Vergessenheit geraten. Meines Erachtens empfiehlt sich daher die folgende Regelung im Projekt: Solange die Entwickler lokal am Quelltext arbeiten, dürfen Modultests auch fehlschlagen. Das Build-Skript verwendet in diesem Fall die Standardeinstellungen für die `<junit>`-Funktion. Sobald Quelltext jedoch ins Repository geschrieben wird, müssen alle korrespondierenden Tests zu 100% fehlerfrei sein. Natürlich sollte man die Einhaltung dieser Regelung prüfen. Hierzu benötigen wir eine striktere Variante des Build-Skriptes, die bei fehlerhaften Tests sofort abbricht. Mit der entsprechenden Erweiterung des Skriptes beschäftigen wir uns in Abschnitt .

Produkt ausliefern

Listing 14
Erstellung der e2etrace-Bibliotheksdatei

Die Auslieferung von *e2etrace* erfolgt in Form einer jar-Datei. Erzeugt wird diese Datei mit der Funktion `<jar>`:

```
<!-- ==== Java-Archivdatei erstellen ==== -->
```

```
<target name="package"
        depends="test"
        description="Erstellt das Java-Archiv">
  <jar destfile="${target.jar}/e2etrace.jar">
    <fileset dir="${target.java}">
      <include name="**/*.class"/>
    </fileset>

  </jar>

</target>
```

Den Pfad und Dateinamen der zu erstellenden jar-Datei erwartet die Funktion im Attribut `destfile`. Im Beispiel wird der Pfad über eine Property `${target.jar}` angegeben. Diese Property muss wie gewohnt in den `build.properties` definiert werden:

```
target.jar=${target}/jar
```

Damit das neue Zielverzeichnis für jar-Dateien auch wirklich erstellt wird, ist außerdem eine entsprechende Erweiterung des Ziels `prepare` notwendig. Auf eine Parametrierung des Dateinamens der Archivdatei habe ich im Listing verzichtet.

Aus welchen Dateien sich die *e2etrace*-Bibliothek zusammensetzt, legen wir über das geschachtelte Fileset-Element fest. Im Beispiel sind dies ausschließlich die kompilierten Java-Quelltextdateien. Die JUnit-Tests sind hingegen nicht Bestandteil der Bibliothek.

Installationsdatei erzeugen

Als Format für die Installationsdatei verwende ich ein zip-Archiv. Ant stellt zur Erstellung von zip-Dateien die Funktion `<zip>` zur Verfügung.

Listing 15
Erstellung der Installationsdatei als zip-Archiv

```
<!-- ==== Installationsdatei erstellen ==== -->
<target name="install"
        depends="package, doc"
        description="Erstellt die Installationsdatei">
  <zip destfile="${target.install}/e2etrace.zip">
    <!-- e2etrace-Bibliothek -->
    <zipfileset dir="${target.jar}"
                includes="e2etrace.jar" />

    <!-- Readme und Lizenzinfos -->
    <zipfileset dir="${src.doc}" includes="*.txt" />

    <!—JavaDoc-Dokumentation -->
    <zipfileset dir="${target.javadoc}" prefix="doc" />

    <!-- Java-Quelltext -->
    <zipfileset dir="${src.java}" prefix="src" />
```

```
    </zip>
  </target>
```

Über das depends-Attribut stelle ich sicher, dass vor der Ausführung von install die *e2etrace*-Bibliothek und die JavaDoc-Dokumentation erstellt werden. Die Installationsdatei wird im Zielverzeichnis ${target.install} unter dem Namen e2etrace.zip erzeugt. Die Selektion der Dateien für das zip-Archiv erfolgt mit Hilfe von <zipfileset>-Elementen. Diese erweitern das normale Fileset um eine Reihe von zusätzlichen Attributen. Im Beispiel verwende ich hiervon nur das Attribut prefix. Dieses legt fest, in welchem Verzeichnis des zip-Archivs die vom Fileset selektierten Dateien abgelegt werden. So wird beispielsweise die JavaDoc-Dokumentation im Archiv unter dem Verzeichnis /doc abgelegt.

Im Listing habe ich zwei neue Properties verwendet, die in den build.properties wie folgt definiert sind:

```
src.doc=${basedir}/doc
target.install=${target}/install
```

Ergebnisse dokumentieren

Um die Ergebnisse des Build-Prozesses festzuhalten, müssen alle Ausgaben in eine Logdatei geschrieben werden. Ant unterstützt die Erstellung von Logdateien durch die Funktion <record>. Nach dem erstmaligen Aufruf der Funktion schreibt Ant alle Bildschirmausgaben zusätzlich in die angegebene Logdatei. Es empfiehlt sich daher, <record> möglichst frühzeitig in den Build-Prozess einzubinden. In unserem Beispiel eignet sich hierfür das Ziel prepare sehr gut. Listing 16 zeigt die entsprechend erweiterte Version des Ziels.

Listing 16
Aktivierung des Loggings im Ziel prepare

```
<!-- ==== Vorbereitung ==== -->
<target name="prepare"
        description="Vorbereitung und Initialisierung">
  <mkdir dir="${target.java}" />
  <mkdir dir="${target.junit}" />
  <mkdir dir="${target.javadoc}" />
  <mkdir dir="${target.jar}" />
  <mkdir dir="${target.install}" />
  <mkdir dir="${target.log}" />

  <record name="${target.log}/build.log"/>

</target>
```

Die Logdatei build.log wird im Verzeichnis ${target.log} angelegt. Eine entsprechende Property habe ich zuvor in den build.properties definiert. Im Beispiel wird die Logdatei nun bei jedem Aufruf des

Build-Skriptes neu erzeugt, d. h., ältere Dateien werden überschrieben. Ich persönlich bevorzuge jedoch eine Variante, die für jeden Aufruf des Skriptes eine neue Logdatei anlegt. Dies erreicht man am einfachsten, indem Datum und Uhrzeit des Build-Laufes in den Namen der Logdatei aufgenommen werden. Um dies in unserem Skript umzusetzen, müssen wir lediglich den Aufruf von `<record>` durch die folgenden beiden Zeilen ersetzen:

Listing 17
Erweiterung des Logdateinamens um Datum und Uhrzeit

```
<tstamp/>
<record name="${target.log}/
                build-${DSTAMP}-${TSTAMP}.log"/>
```

Die Funktion `<tstamp>` initialisiert die Properties `${DSTAMP}` und `${TSTAMP}` mit dem aktuellen Datum und der aktuellen Uhrzeit. Diese beiden Properties verwende ich in der `<record>`-Funktion als Bestandteil des Dateinamens. Führt man das Build-Skript dann beispielsweise am 23. Februar 2006 um 11:51 Uhr aus, wird eine Logdatei mit dem Namen `build-20060223-1151.log` erstellt. Wem die Standardformate für Datum und Uhrzeit nicht gefallen, kann diese beim Aufruf von `<tstamp>` auch ändern (Details hierzu sind in der Ant-Dokumentation zu finden).

Zusammenfassung und Ausblick

Das bis jetzt erstellte, relativ einfache Build-Skript würde als Listing im Buch bereits mehrere Seiten in Anspruch nehmen. Da XML-Dokumente ohne geeignete Unterstützung durch einen Editor nur schwer lesbar sind, erspare ich Ihnen diesen »Zeichensalat«. Sie finden das komplette Build-Skript und die zugehörige Property-Datei zum Download auf der Webseite zum Buch.

Das Skript bildet in der jetzigen Form eine solide Basis für einen Build-Prozess. Wir können das Produkt *e2etrace* aus den Konfigurationselementen erstellen, überprüfen und ausliefern. Allerdings habe ich eine Reihe von typischen Problemstellungen in einem KM-Prozess vernachlässigt. Beispielsweise fehlt bisher jegliche Unterstützung für die Vorbereitung und Erstellung eines neuen Release. In den folgenden Kapiteln werden wir das Basisskript daher so ausbauen, dass es auch den Anforderungen eines KM-Prozesses genügt.

Einführung von Build-Varianten

Das Skript für *e2etrace* implementiert bisher eine Art universellen Build-Prozess. Es macht keinen Unterschied zwischen den lokalen

Builds eines Entwicklers und der »offiziellen« Auslieferung des Produktes. Um dies zu ändern, führen wir nun drei verschiedene Varianten des Build-Prozesses ein: den Entwickler-, den Integrations- und den Release-Build.

Prinzipielle Vorgehensweise

Bevor wir uns in die Implementierung der Build-Varianten stürzen, sollten wir uns ein paar Gedanken über die prinzipielle Vorgehensweise machen. Zur Umsetzung der Varianten mit Ant stehen uns mehrere Alternativen zur Verfügung:

1. Wir erstellen ein großes »Monster-Skript«, in dem die einzelnen Varianten über Properties unterschieden werden.
2. Wie Variante 1, nur dass für die Varianten jeweils unterschiedliche Build-Ziele implementiert werden.
3. Wir erstellen pro Variante ein separates Build-Skript.

Gegen Alternative 1 sprechen die schlechte Wartbarkeit und die ungenügende Unterstützung von Ant für Kontrollstrukturen. Die Ant-Skriptsprache ist spezialisiert auf die Implementierung von Build-Prozessen. Der Kontrollfluss, also die Reihenfolge, in der die einzelnen Build-Ziele und Funktionen abgearbeitet werden, wird statisch bei der Skripterstellung über das `depends`-Attribut festgelegt. Zwar kann der Kontrollfluss durch die bedingte Ausführung von Build-Zielen beeinflusst werden, dies ist jedoch kein vollwertiger Ersatz für die Möglichkeiten, die generische Skriptsprachen, wie beispielsweise *Perl* oder *Ruby,* bieten. Mit etwas Motivation wäre es sicherlich machbar, ein Ant-Skript so zu »verbiegen«, dass auch die bedingte Abarbeitung von Build-Varianten möglich ist, aber warum sollten wir uns das antun?

Alternative 2 ist schon besser, allerdings besteht auch hier die Gefahr, ein »Monsterskript« zu produzieren. Daher bevorzuge ich Alternative 3. Wir erstellen also pro Build-Variante ein eigenes Skript

(siehe Abb. 2). Um Redundanzen zu vermeiden, lagern wir zusätzlich die gemeinsam genutzte Funktionalität in ein Skript `common.xml` aus.

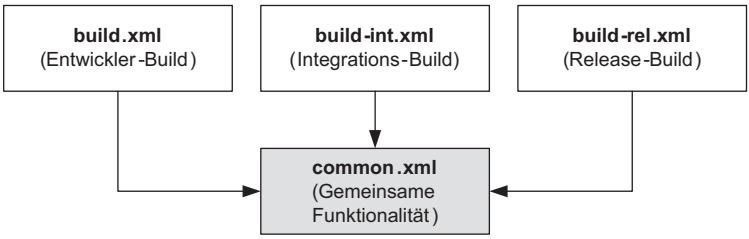


Abb. 2
Aufteilung der Build-Varianten auf mehrere Skripte

Aufteilung der Funktionalität auf die Skripte

Damit die Modularisierung unseres Build-Prozesses nicht schiefgeht, müssen wir uns weiterhin einige Gedanken über die Aufteilung der Funktionalität machen. In `common.xml` soll die *gemeinsame Funktionalität* enthalten sein, doch was genau bedeutet dies im Detail?

Öffentliche Build-Ziele

Genau wie in einem normalen Programm auch empfiehlt es sich, auch beim Entwurf eines etwas komplexeren Build-Prozesses zwischen *öffentlicher* und *privater Funktionalität* zu unterscheiden. Die öffentliche Funktionalität in unserem Prozess wird durch die Build-Ziele in den drei Skripten `build.xml`, `build-int.xml` und `build-rel.xml` repräsentiert. Nur die in diesen drei Dateien definierten Ziele werden über die Kommandozeile aufgerufen.

Private Build-Ziele

In `common.xml` befinden sich hingegen ausschließlich private Ziele, die von den öffentlichen Zielen der anderen Skripte aufgerufen werden. In Ant-Skripten werden private Ziele in der Regel durch ein Minuszeichen am Anfang des Namens gekennzeichnet. Lässt man zusätzlich das Attribut `description` bei der Definition des `<target>`-Elements weg, unterdrückt Ant private Ziele auch bei der Ausgabe der Projekthilfe über die Option `–projecthelp`.

Erstellung der ersten Version von common.xml

Die erste Version von `common.xml` besteht im Prinzip aus dem in Abschnitt erstellten Skript `build.xml`. Da die gemeinsame Funktionalität jedoch nur aus privaten Zielen bestehen soll, erhalten alle Build-Zielnamen ein Minuszeichen als Präfix. Zusätzlich werden sämtliche

description- und depends-Attribute aus den <target>-Elementen entfernt. Im Endeffekt entsteht dadurch eine Art Bibliothek mit wiederverwendbaren Build-Zielen. Diese haben keinerlei Abhän-gigkeiten untereinander, der Kontrollfluss des Build-Prozesses wird später von den übergeordneten Skripten festgelegt.

Listing 18 zeigt einen Ausschnitt aus der so entstandenen Datei common.xml. Die Implementierung der <target>-Elemente hat sich, mit der Ausnahme von –prepare, gegenüber Abschnitt nicht geändert. Im Listing habe ich sie daher nicht wiederholt. Im Ziel –prepare fallen lediglich die Funktionen <tstamp> und <record> zur Aktivierung des Loggings weg. Mehr dazu im nächsten Abschnitt.

Listing 18
Ausschnitt aus common.xml

```
<?xml version="1.0"?>
<project name="common">
  <!-- ==== Definition der Properties ==== -->
  <property environment="os" />
  <property file="${os.E2E_PERSONAL_PROPERTIES}" />
  <property file="ant/build.properties" />

  <!-- ==== Selektion der KM-Elemente ==== -->
  <!-- Java-Quelltext -->
  <fileset id="JavaQuelltext" dir="${src.java}">
    <include name="**/*.java" />
  </fileset>

  <!-- JUnit-Tests -->
  <fileset id="JUnitTests" dir="${src.junit}">
    <include name="**/*Test.java" />
  </fileset>

  <!-- ==== Externe Bibliotheken einbinden ==== -->
  <path id="ExternalLibs">
    <pathelement location="${loggingInstall}/
                           commons-logging.jar" />
    <pathelement location="${junitInstall}/junit-4.3.1.jar" />
  </path>

  <!-- ==== Vorbereitung ==== -->
  <target name="-prepare">
    <mkdir dir="${target.java}" />
    <mkdir dir="${target.junit}" />
    <mkdir dir="${target.javadoc}" />
    <mkdir dir="${target.jar}" />
    <mkdir dir="${target.install}" />
    <mkdir dir="${target.log}" />
  </target>

  <!-- ==== Java-Compiler aufrufen ==== -->
  <target name="-compile">
    ...
```

```
  </target>

  ... (Alle restlichen Build-Ziele)
</project>
```

Entwickler-Build

Der Entwickler-Build wird im Rahmen der täglichen Projektarbeit auf den lokalen Rechnern des Entwicklerteams ausgeführt. Er muss einfach zu bedienen und vor allem schnell sein. Ein erster Schritt in diese Richtung ist die Wahl des Dateinamens. Für den Entwickler-Build verwenden wir den Standardnamen `build.xml`. Beim Aufruf von Ant kann dadurch auf die Angabe des zu verwendenden Skriptes verzichtet werden.

Build-Ziele im Entwicklerskript

Das Skript für den Entwickler-Build besteht aus den in Tabelle 1 aufgezählten öffentlichen Zielen. Jedes Ziel ruft intern das entsprechende private Ziel aus `common.xml` auf. Der Kontrollfluss wird über den in der Spalte »Abhängig von« angegebenen Inhalt des `depends`-Attributes festgelegt.

Tab. 1
Ziele des Entwickler-Build-Skriptes

Build-Ziel	Abhängig von	Beschreibung
prepare	–	Vorbereitung und Initialisierung
compile	prepare	Kompiliert den Java-Quelltext und die JUnit-Tests. Hierbei sollen alle Debug-Informationen in die .class-Dateien eingebunden werden.
doc	prepare	Generiert die JavaDoc-Dokumentation.
test	compile	Führt alle Modultests aus.
package	compile	Erstellt das Java-Archiv.
clean	–	Entfernt alle generierten Dateien und Verzeichnisse.

Unterschiede zum Skript aus Abschnitt

Die Funktionalität des Entwickler-Build-Skriptes unterscheidet sich in drei Punkten von dem Skript aus Abschnitt :

1. Die Installationsdatei für *e2etrace* kann durch den Entwickler-Build nicht erstellt werden. Das Ziel `install` existiert daher im Skript nicht.
2. Für das Ziel `package` wird auf die automatische Ausführung der JUnit-Tests vor der Erstellung der Bibliotheksdatei verzichtet. Das Ziel `package` ist also lediglich von `compile` abhängig.
3. Bisher haben wir den Quelltext mit den Standardeinstellungen der Funktion `<javac>` und damit ohne Debug-Informationen kompi-

liert. Im Entwickler-Build sollen jedoch explizit alle verfügbaren Debug-Informationen in die .class-Dateien eingebunden werden.

Kompilieren mit Debug-Informationen

Da das Ziel compile im Entwicklerskript auf dem gemeinsam genutzten Ziel –compile aufsetzt, ist zur Umsetzung des zuletzt genannten Punktes eine Änderung in common.xml notwendig. Ob Debug-Informationen eingebunden werden oder nicht, wird beim Aufruf der Funktion <javac> über die Attribute debug und debuglevel festgelegt. Da –compile ein wiederverwendbares Ziel bleiben soll, können wir den Wert dieser Attribute in common.xml nicht statisch festlegen. Wir führen daher zwei Properties ${compile.debug} und ${compile.debug.level} ein, die vom Aufrufer des Ziels als Parameter übergeben werden müssen. Listing 19 zeigt die entsprechend erweiterte Version des Ziels –compile.

Listing 19
Optionale Einbindung von Debug-Informationen

```
<!-- ==== Java-Compiler aufrufen ==== -->
<!-- Parameter:
     compile.debug = true/off
     compile.level.debug = lines | vars | source
-->
<target name="-compile">
  <!-- Java-Quelltext kompilieren -->
  <javac srcdir="${src.java}"
         destdir="${target.java}"
         debug="${compile.debug}"
         debuglevel="${compile.debug.level}">
    <classpath refid="ExternalLibs" />
  </javac>

  <!-- JUnit-Tests kompilieren -->
  <javac srcdir="${src.junit}"
         destdir="${target.junit}"
         debug="${compile.debug}"
         debuglevel="${compile.debug.level}">
    <classpath>
      <pathelement location="${target.java}" />
      <path refid="ExternalLibs" />
    </classpath>
  </javac>
</target>
```

Implementierung des Entwicklerskriptes

Listing 20
Implementierung des Entwickler-Build-Skriptes

Die vollständige Implementierung des Entwicklerskriptes ist in Listing 20 dargestellt:

```
<?xml version="1.0"?>
<project name="e2etrace-dev" basedir="..">
  <description>
```

```
    Entwickler-Build-Skript für e2etrace
  </description>

  <import file="common.xml" />

  <!-- ==== Vorbereitung ==== -->
  <target name="prepare"
          description="Vorbereitung und Initialisierung">
    <antcall target="-prepare" />
    <tstamp />
    <record name="${target.log}/
            build-${DSTAMP}-${TSTAMP}.log" />
  </target>

  <!-- ==== Java-Compiler aufrufen ==== -->
  <target name="compile" depends="prepare"
          description="Kompiliert den Java-Quelltext und
                       die JUnit-Tests">
    <antcall target="-compile">
      <param name="compile.debug" value="true" />
      <param name="compile.debug.level"
             value="lines,vars,source" />
    </antcall>
  </target>

  <!-- ==== JavaDoc generieren ==== -->
  <target name="doc" depends="prepare"
          description="Generiert die JavaDoc-
                       Dokumentation">
    <antcall target="-doc" />
  </target>

  <!-- ==== Modultests ausführen ==== -->
  <target name="test" depends="compile"
          description="Führt alle Modultests aus">
    <antcall target="-test" />
  </target>

  <!-- ==== Java-Archivdatei erstellen ==== -->
  <target name="package" depends="compile"
          description="Erstellt Java-Archiv">
    <antcall target="-package" />
  </target>

  <!-- ==== Generierte Dateien entfernen ==== -->
  <target name="clean"
          description="Entfernt alle generierten Dateien
                       und Verzeichnisse">
     <antcall target="-clean" />
  </target>
</project>
```

Am Anfang des Skriptes wird über die Funktion <import> die Datei common.xml eingebunden. Anschließend können alle Build-Ziele aus common.xml im Skript verwendet werden. Die öffentlichen Ziele des Entwicklerskriptes rufen dann via <antcall> die Basisziele aus common.xml auf. Zusätzlich wird der Kontrollfluss des Entwicklerskriptes über die depends-Attribute festgelegt.

Im Ziel prepare wird nach dem Aufruf des Basisziels das Logging aktiviert. Dies kann nur hier und nicht im Build-Ziel –prepare aus common.xml erfolgen, da <record> nicht in einer durch <antcall> aufgerufenen Funktion verwendet werden kann. In diesem Fall würde <record> lediglich den Ablauf dieser einen Funktion protokollieren.

Bei der Implementierung von compile müssen zusätzlich die Parameter compile.debug und compile.debug.level an das Basisziel –compile übergeben werden. Dies erfolgt über die geschachtelten <param>-Elemente der <antcall>-Funktion:

Integrations-Build

Über den Integrations-Build stellen wir fest, in welcher Verfassung sich das Projekt befindet. In dieser Build-Variante laufen, zumindest vom Standpunkt des Entwicklungsteams aus betrachtet, alle Fäden zusammen. Die mehr oder weniger isoliert durchgeführten Änderungen der einzelnen Entwickler werden jetzt zusammengeführt und getestet. Wenn der Integrations-Build inklusive aller Tests durchläuft, hat das Team gute Arbeit geleistet. Wenn nicht, hat jemand geschlafen oder schlicht gepfuscht. Es ist meines Erachtens wichtig, dem Integrations-Build eine zentrale Rolle im Projekt einzuräumen und fehlerhafte Builds nicht auf die leichte Schulter zu nehmen.[9]

Erweiterungen im Integrations-Build

Im Vergleich zu den bisher vorgestellten Skripten werden im Integrations-Build folgende Erweiterungen realisiert:

- *Abgleich mit dem Repository*
  Das Skript verwendet automatisch den jeweils neuesten Stand aus dem Repository.

9. In [Clark04] werden diverse, teilweise recht amüsante Möglichkeiten beschrieben, wie ein fehlerhafter Integrations-Build allen Teammitgliedern eindrucksvoll mitgeteilt werden kann. In einem der beschriebenen Szenarien werden beispielsweise zwei per USB angesteuerte Lava-Lampen (eine grüne und eine rote) verwendet. Ich werde etwas später noch eine vergleichsweise banale Alternative in Form einer E-Mail-Benachrichtigung beschreiben, finde die Lava-Lampen-Idee aber sehr cool.

- *Kennzeichnung und Auslieferung des Produktes*
  Als Ergebnis des Integrations-Builds wird eine Installationsdatei von *e2etrace* erstellt. Jede Auslieferung von *e2etrace* wird eindeutig gekennzeichnet und kann auch im Nachhinein einer Revision des Repositorys zugeordnet werden.
- *Benachrichtigung der Teammitglieder*
  Die Teammitglieder werden per E-Mail über das Ergebnis des Integrations-Builds benachrichtigt.
- *Automatischer, zeitgesteuerter Ablauf des Skriptes*
  Der Integrations-Build soll vollständig automatisch durchgeführt werden. Hierzu wird im Projekt am besten ein separater Rechner eingerichtet, auf dem das Skript zeitgesteuert gestartet wird.

Einsatz spezialisierter Werkzeuge

Wie wir gleich sehen werden, hält sich der Aufwand zur Umsetzung der genannten Punkte in Grenzen. Dies ändert sich jedoch schlagartig, wenn mehr Komfort und Flexibilität gewünscht sind. Ein Beispiel hierfür sind optisch ansprechende Berichte, die abhängig vom Build-Ergebnis an unterschiedliche Mail-Verteiler geschickt werden. Bevor man an dieser Stelle zusätzlichen Aufwand investiert, empfehle ich dringend, einen Blick auf *CruiseControl*[10] oder *Luntbuild*[11] zu werfen. CruiseControl und Luntbuild sind Open-Source-Werkzeuge speziell zur Umsetzung von automatisierten Integrations-Builds. Die Einarbeitung in eines dieser Tools lohnt sich meines Erachtens dann, wenn die Anforderungen an den Integrations-Build deutlich über die oben genannten Punkte hinausgehen.

Build-Ziele im Integrationsskript

Das Integrationsskript wird automatisch ausgeführt und erstellt als Ergebnis immer die Installationsdatei von *e2etrace*. Im Gegensatz zum Entwicklerskript ist es daher nicht notwendig, die einzelnen Schritte des Build-Prozesses als separate Ziele zu definieren. Wir benötigen demzufolge lediglich die drei in Tabelle 2 aufgezählten Build-Ziele.

10. *http://cruisecontrol.sourceforge.net*
11. http://sourceforge.net/projects/luntbuild

Build-Ziel	Abhängig von	Beschreibung
prepare	–	Vorbereitung und Initialisierung. In diesem Ziel wird die Protokollierung des Integrations-Builds gestartet. Im Gegensatz zum Entwickler-Build heißt die Logdatei jedoch immer `build.log`. Da die Ergebnisse des Skriptes per Mail verteilt werden, kann auf die Datumsangaben im Namen der Logdatei verzichtet werden.
install	clean, prepare	Erstellt die Installationsdatei von e2etrace. Die Datei wird nur erstellt, wenn alle Modultests erfolgreich durchlaufen wurden. Um die Fehlersuche zu erleichtern, werden vom Compiler die Zeilennummern als Debug-Informationen eingebunden.
clean	–	Entfernt alle generierten Dateien und Verzeichnisse.

Tab. 2
Ziele des Integrations-Build-Skriptes

Vermeidung der inkrementellen Kompilierung

Die Abhängigkeit des Ziels `install` von `clean` und `prepare` garantiert, dass jedes Mal ein vollständiger Build durchgeführt wird. Dies dauert zwar deutlich länger als der inkrementelle Build des Entwicklerskriptes, für den Integrations-Build spielt dies allerdings keine Rolle.[12] Vollständige Builds haben den Vorteil, dass sie auch bei strukturellen Veränderungen am Quelltext zuverlässig funktionieren. Ein inkrementeller Build kommt hingegen ins Stolpern, wenn beispielsweise eine Quelltextdatei verschoben wird. Die `<javac>`-Funktion erkennt in diesem Fall zwar die »neue« Datei am Zielort der Verschiebe-Operation und kompiliert diese. Die »alte« `.class`-Datei am Ursprungsort wird jedoch nicht aus dem `target`-Verzeichnis entfernt. Im Endeffekt landet die Klasse dann in zwei Versionen im Java-Archiv und kann allerlei seltsame Probleme auslösen.

Erster Entwurf des Integrationsskriptes

Der in Listing 21 dargestellte, erste Entwurf des Integrationsskriptes besteht, analog zum Entwicklerskript, im Wesentlichen aus Aufrufen der Ziele in `common.xml`. Diesmal wird der Kontrollfluss allerdings maß-

12. Dies gilt zumindest für »normal große« Projekte. In extrem großen Projekten mit über 1.000.000 Zeilen Code ist ein vollständiger Integrations-Build eventuell nicht mehr machbar.

geblich durch die Reihenfolge der <antcall>-Funktionen im Build-Ziel install bestimmt.

Listing 21
Erster Entwurf des Integrationsskriptes

```
<?xml version="1.0"?>
<project name="e2etrace-int" basedir="..">
  <description>
    Integrations-Build-Skript für e2etrace
  </description>

  <import file="common.xml" />

  <!-- ==== Vorbereitung ==== -->
  <target name="prepare"
          description="Vorbereitung und Initialisierung">
    <antcall target="-prepare" />
    <record name="${target.log}/build.log" />
  </target>

  <!-- ==== Installationsdatei erstellen ==== -->
  <target name="install"
          depends="clean, prepare"
          description="Erstellt die Installationsdatei">
    <!—Java-Compiler aufrufen -->
    <antcall target="-compile">
      <param name="compile.debug" value="true" />
      <param name="compile.debug.level" value="lines" />
    </antcall>

    <!-- Modultests ausführen -->
    <antcall target="-test">
      <param name="test.haltonfailure" value="true" />
    </antcall>

    <!-- JavaDoc generieren -->
    <antcall target="-doc" />

    <!-- Java-Archivdatei erstellen -->
    <antcall target="-package" />

    <!-- Installationsdatei erstellen -->
    <antcall target="-install" />
  </target>

  <!-- ==== Generierte Dateien entfernen ==== -->
  <target name="clean"
          description="Entfernt alle generierten Dateien
                       und Verzeichnisse">
    <antcall target="-clean" />
  </target>
</project>
```

Beim Aufruf des Basisziels –compile wird im Listing der Parameter compile.debug.level auf lines gesetzt. Dies veranlasst den Compiler, nur

die Zeilennummern als Debug-Informationen in die .class-Dateien aufzunehmen.

Abbruch des Skriptes bei fehlgeschlagenen Testfällen

Neu hinzugekommen ist der Parameter test.haltonfailure beim Aufruf des Basisziels –test. Bisher haben wir aus fehlgeschlagenen Tests keine Konsequenzen gezogen. Für den Integrations-Build ist eine striktere Gangart angesagt, hier müssen fehlerhafte JUnit-Tests zum Abbruch des Skriptes führen. Genau dies bewirkt der oben genannte Parameter. Die entsprechende Erweiterung des Ziels –test in common.xml zeigt Listing 22.

Listing 22
Erweiterung des Ziels –test in common.xml um den Parameter test.haltonfailure

```
<!-- ==== Modultests ausführen ==== -->
<!-- Parameter:
     test.haltonfailure = true | false
-->
<target name="-test">
  <delete dir="${target.testreports}" />
  <mkdir dir="${target.testreports}" />

  <!-- Test ausführen -->
  <junit printsummary="yes"
         failureproperty="-test.failure">
    <classpath>
      <pathelement location="${target.java}" />
      <pathelement location="${target.junit}" />
      <path refid="ExternalLibs" />
    </classpath>

    <formatter type="plain" />
    <formatter type="xml" />

    <batchtest todir="${target.testreports}">
      <fileset refid="JUnitTests" />
    </batchtest>

  </junit>

  <junitreport todir="${target.testreports}">
    <fileset dir="${target.testreports}">
      <include name="TEST-*.xml"/>
    </fileset>
    <report format="frames"
            todir="${target.testreports}/html"/>
  </junitreport>

  <!-- Skript ggf. abbrechen -->
  <condition property="-test.callfail">
    <and>
      <istrue value="${test.haltonfailure}" />
      <isset property="-test.failure" />
    </and>
  </condition>
```

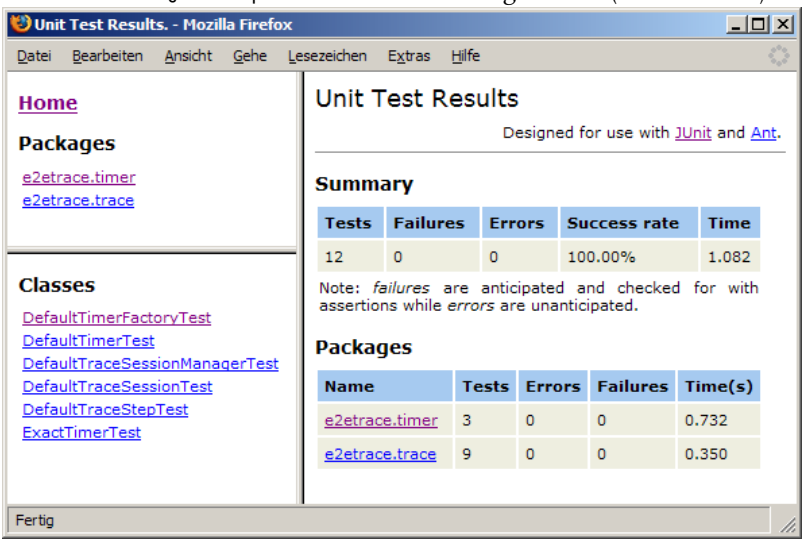
```
        <fail message="Modultests sind fehlgeschlagen!"
              if="-test.callfail" />

    </target>
```

Erzeugung von Testreporten im HTML-Format

Die einfache Lösung zur Anpassung des Ziels –test wäre die Verwendung des bereits in Abschnitt erwähnten Attributes haltonfailure gewesen. Ich habe mich allerdings für eine auf den ersten Blick etwas umständlichere Implementierung entschieden und auf das Attribut verzichtet. Das Problem mit haltonfailure ist, dass es das Skript im Fehlerfall sofort nach der Abarbeitung der <junit>-Funktion beendet. Dies ist jedoch aufgrund einer weiteren Erweiterung des Build-Ziels nicht erwünscht. Im Listing ist zu erkennen, dass die Testreporte jetzt nicht mehr nur im ASCII-Format, sondern zusätzlich in Form von XML-Dateien erzeugt werden. Aus diesen XML-Dateien werden mit Hilfe der Funktion <junitreport> HTML-Dateien generiert (siehe Abb. 3).

Abb. 3
Testreporte im HTML-Format



Damit die Aufbereitung der Reporte auch bei fehlerhaften Tests durchgeführt wird, darf das Skript natürlich nicht bereits in der <junit>-Funktion abgebrochen werden. Statt haltonfailure verwende ich daher das Attribut failureproperty. Dieses veranlasst die <junit>-Funktion, im Fehlerfall die übergebene Property –test.failure neu anzulegen.

Nachdem die HTML-Reporte erzeugt wurden, prüfe ich dann mit der <condition>-Funktion, ob die Property –test.failure existiert *und* ob der Parameter test.haltonfailure den Wert true hat. Nur wenn beide Bedingungen erfüllt sind, setzt die <condition>-Funktion eine weitere, temporäre Property –test.callfail auf true. In der letzten

Zeile des Build-Ziels wird schließlich die Funktion `<fail>`-ausgeführt, wenn diese temporäre Property existiert. `<fail>` bewirkt einen sofortigen Abbruch des Build-Skripts.

Synchronisation mit dem Repository

Der Integrations-Build konsolidiert alle Änderungen und Erweiterungen des Entwicklungsteams in einer neuen Version des Produktes. Daher ist die automatische Synchronisation mit dem Repository Teil des Build-Prozesses. In unserem Fall wird hierzu der `update`-Befehl von Subversion eingesetzt. Wer an dieser Stelle kurz einen Blick in die Ant-Dokumentation wirft, wird zunächst feststellen, dass keinerlei Funktionen zum Zugriff auf ein Subversion-Repository existieren. Zwar werden CVS und eine Reihe von kommerziellen Versionskontrollsystemen unterstützt, aber eben kein Subversion[13]. Dies ist nicht unbedingt ein Problem, denn wir können den Subversion-Client auch mit Hilfe der `<exec>`-Funktion aufrufen. Ein entsprechendes Beispiel bespreche ich etwas später.

Allerdings stellt sich die Frage, ob wir überhaupt aus dem Integrationsskript selbst auf das Repository zugreifen wollen. Da das Skript Bestandteil des Repositorys ist, könnte es durch die abgerufenen Changesets verändert werden. Für den Integrations-Build müsste dann diese neue Version des Skriptes verwendet werden. Wir haben es also offensichtlich mit einem Henne-Ei-Problem zu tun.

Man könnte an dieser Stelle argumentieren, dass Änderungen am Integrationsskript relativ selten sind und das Problem daher eher theoretischer Natur ist. Meiner Erfahrung nach stimmt die Aussage, dass Änderungen am Integrationsskript selten sind. *Wenn* es allerdings geändert wird, dann im denkbar ungünstigsten Moment. *Murphy's Law*[14] ist in der Softwareentwicklung nun mal eine Tatsache.

Auslagerung des Subversion-Kommandos in ein Shell-Skript

Ich bevorzuge daher eine sehr simple Alternative. Statt den `update`-Befehl in das Integrationsskript selbst einzubauen, führe ich ihn in einem minimalen Wrapper-Shell-Skript aus. Dieses ist, wie das Integrationsskript, Teil des Repositorys. Die Implementierung des Wrappers ist derart trivial, dass es nach der Neuerstellung wirklich nicht mehr geändert werden muss. Listing 23 zeigt die Windows-Variante des Wrapper-Skripts `build-int.cmd`. Es wird, zusammen mit `build-int.xml`, im Verzeichnis `ant` abgelegt.

13. Es gibt allerdings unter [URL:AntSvnLib] eine Ant-Erweiterung zum Download, die zumindest einige einfache Ant-Funktionen zum Zugriff auf Subversion bietet. Einen wesentlichen Fortschritt gegenüber den manuellen Aufrufen mit Hilfe von `<exec>` stellt sie jedoch meiner Ansicht nach nicht dar.
14. »If anything can go wrong, it will«, *http://www.murphys-laws.com*

Listing 23
Synchronisation mit dem Repository und Aufruf des Integrationsskriptes über ein Shell-Skript

```
@echo off
cd ..
svn update > ant\~svnupdate.log
cd ant
call ant -f build-int.xml install
del ~svnupdate.log
```

Das Shell-Skript leitet die Ausgabe des `update`-Befehls in eine Datei `~svnupdate.log` um. Wir werden diese Datei später in die per Mail verschickte Benachrichtigung über Erfolg oder Misserfolg des Integrations-Builds einbinden.

In Zukunft darf der Integrations-Build nur noch über das neue Shell-Skript `build-int.cmd` gestartet werden. Da wir die Ausführung sowieso automatisieren wollen, können wir dies problemlos gewährleisten. Eine weitere Voraussetzung ist natürlich, dass ein separater Arbeitsbereich für den Integrations-Build angelegt wird.

Parallele Entwicklungspfade

Wenn mehrere Entwicklungspfade im Projekt aktiv sind, beispielsweise der *trunk* und ein Release-Branch, wird pro Pfad ein eigener Integrations-Build durchgeführt. Hierzu erzeugt man mit dem `checkout`-Befehl zwei Integrationsarbeitsbereiche:

```
svn checkout svn://localhost/e2etrace/trunk \
             integrate-trunk
svn checkout svn://localhost/e2etrace/branches/RB-1.0.0 \
             integrate-RB-1.0.0
```

In beiden Arbeitsbereichen kann dann mit `build-int.cmd` der jeweilige Integrations-Build gestartet werden.

Verwendung von Build-Nummern

Integrations-Builds dienen nicht nur als regelmäßiges Feedback für die Entwickler, sondern bilden auch die Grundlage für die Arbeit des restlichen Projektteams. So werden beispielsweise in fortgeschritteneren Projektphasen die vom Skript erstellten Installationsdateien in regelmäßigen Abständen an Testanwender ausgeliefert. Diese finden unweigerlich Fehler und melden diese über den Fehlermanagementprozess an die Entwickler zurück. Für das Entwicklerteam ist es nun wichtig, die genaue Produktversion zu kennen, auf die sich eine Fehlermeldung bezieht. Da für interne Auslieferungen keine Release-Nummern verwendet werden, benötigen wir ein alternatives Identifikationsschema.

Gut geeignet sind meiner Erfahrung nach die sogenannten *Build-Nummern*. Hierbei wird für jede durch den Integrations-Build erstellte Produktversion eine eindeutige Nummer vergeben. Werden später beispielsweise Fehlerberichte für eine interne Auslieferung erfasst, beziehen sich diese immer auf eine bestimmte Build-Nummer. Das Entwick-

lerteam kann damit den Bezug zum Repository herstellen und für die Fehlersuche die richtige Revision des Quelltextes verwenden.

Ant unterstützt die Generierung von Build-Nummern durch eine eigene Funktion `<buildnumber>`. Diese stellt mit Hilfe einer Datei die Eindeutigkeit der Build-Nummern sicher. Zusätzlich müssten wir jedoch den Bezug zum Repository herstellen, beispielsweise mit Hilfe eigens erstellter Tags für jeden Build.

Ermittlung der Build-Nummer aus dem Repository

Beim Einsatz von Subversion als Repository gibt es jedoch eine meiner Ansicht nach elegantere Alternative. Wir können direkt die Repository-Revision als Build-Nummer verwenden. Dies garantiert zum einen die Eindeutigkeit der Nummern und stellt zudem ohne weiteren Aufwand eine Verbindung zwischen Produktversion und Repository her.

Changeset-Revision als Build-Nummer

Da der Integrations-Build unter Umständen parallel in mehreren Entwicklungspfaden durchgeführt wird, dürfen wir jedoch nicht die globale Repository-Revision als Grundlage für die Build-Nummer verwenden. Diese wäre in allen Zweigen identisch. Relevant ist vielmehr die Changeset-Revision des aktuellen Entwicklungszweigs.

Ermittlung der Changeset-Revision

Zur Ermittlung der Changeset-Revision ist das Subversion-Kommando `info` gut geeignet. Für den Arbeitsbereich `integrate-trunk` könnte das Ergebnis beispielsweise wie folgt aussehen:

```
>svn info
Pfad: .
URL: svn://localhost/e2etrace/trunk
Basis des Projektarchivs: svn://localhost/e2etrace
UUID des Projektarchivs: 011d8a56-4c44-7149-85a0-246079c8a55b
Revision: 32
Knotentyp: Verzeichnis
Plan: normal
Letzter Autor: POPPGU
Letzte geänderte Rev: 32
Letztes Änderungsdatum: 2006-02-28 09:38:56
```

Ausgabe der Arbeitsbereich-Infos im XML-Format

Uns interessiert die Revision aus der Zeile »Letzte geänderte Rev«, im Beispiel also 32. Die Auswertung der oben gezeigten Rückgabe im Build-Skript wäre allerdings recht mühsam und zudem abhängig von der installierten Sprachversion von Subversion. Glücklicherweise unterstützt der `info`-Befehl auch die Option `--xml`. Gibt man diese an, liefert Subversion als Ergebnis ein XML-Dokument:

```
>svn info --xml
<?xml version="1.0" encoding="utf-8"?>
```

```
<info>
  <entry kind="dir" path="." revision="32">
     <url>svn://localhost/e2etrace/trunk</url>
     <repository>
       <root>svn://localhost/e2etrace</root>
       <uuid>011d8a56-4c44-7149-85a0-246079c8a55b</uuid>
     </repository>
     <wc-info>
       <schedule>normal</schedule>
     </wc-info>
     <commit revision="32">
        <author>POPPGU</author>
        <date>2006-02-28T08:38:56.787235Z</date>
     </commit>
  </entry>
</info>
```

Diese Rückgabe können wir in eine Datei umleiten und im Skript auswerten. Der von uns gesuchte Wert befindet sich im Attribut `revision` des `<commit>`-Elements.

Ant-Makros

Bisher haben wir die Funktionalität des Skripts mit Hilfe von Build-Zielen implementiert. Für die Ermittlung der Build-Nummer verwende ich stattdessen ein *Makro*. Der Grund hierfür ist, dass Build-Ziele keine Ergebnisse an einen Aufrufer zurückliefern können. Würde beispielsweise in einem der gemeinsam genutzten Ziele in `common.xml` eine neue Property definiert werden, wäre diese nach dem Aufruf des Ziels mit `<antcall>` in `build-int.xml` nicht sichtbar. Aus technischer Sicht liegt dies daran, dass `<antcall>` für den Aufruf des Ziels einen eigenen, separaten Namensraum für Properties verwendet.

Ein Makro wird hingegen nicht aufgerufen, sondern eingebettet. Der Mechanismus ist im Prinzip vergleichbar mit der *Suchen & Ersetzen*-Funktionalität eines Texteditors. Das Makro zur Ermittlung der Build-Nummer in Listing 24 kann daher problemlos eine neue Property erzeugen, die auch für den Aufrufer sichtbar ist.

Listing 24
Makro zur Ermittlung der Build-Nummer aus der Changeset-Revision

```
<!-- Build-Nummer aus Changeset-Revision ermitteln -->
<macrodef name="getbuildnr">
  <attribute name="buildnrproperty" />
  <sequential>
    <exec executable="svn" output="~getbuildnr.xml">
      <arg value="info" />
      <arg value="--xml" />
    </exec>
    <xmlproperty file="~getbuildnr.xml"
                 collapseAttributes="true" />
    <property name="@{buildnrproperty}"
              value="${info.entry.commit.revision}" />
    <echo message="Verwende Build-Nummer:
```

```
                    ${info.entry.commit.revision}" />
      <delete file="~getbuildnr.xml" />
    </sequential>
  </macrodef>
```

Über das Element <attribute> wird ein Parameter buildnrproperty für das Makro festgelegt. Beim Aufruf des Makros kann über diesen Parameter der Name einer Property übergeben werden. Dieser wird in der Zeile <property name=«@{buildnrproperty} …/> anstelle des mit @ markierten Parameters eingesetzt. Das Makro definiert dadurch die Property neu und initialisiert sie mit der Build-Nummer (siehe unten).

Die eigentliche Implementierung des Makros befindet sich innerhalb des <sequential>-Blockes. Als Erstes wird der Subversion-Client mit Hilfe der Funktion <exec> aufgerufen. Über das Attribut output werden alle Ausgaben des Clients in die Datei ~getbuildnr.xml umgeleitet. Das Kommando info und die Option –-xml übergeben wir mit den geschachtelten <arg>-Elementen. Ant bastelt aus diesen Angaben dann einen für das verwendete Betriebssystem passenden Aufruf zusammen.

Parsen einer XML-Datei

Nach der Ausführung des Kommandos muss die Changeset-Revision aus der XML-Datei gelesen werden. Ant bietet hierfür die sehr praktische Funktion <xmlproperty> an. Diese erzeugt aus allen Elementen und Attributen einer XML-Datei Ant-Properties. Geschachtelte Elemente werden durch Punkte getrennt. Da ich im Beispiel zudem den Parameter collapseAttributes auf true gesetzt habe, werden Attribute in der XML-Datei wie geschachtelte Elemente behandelt.

Die Changeset-Revision aus der Datei ~getbuildnr.xml wird demnach in die Property ${info.entry.commit.revision} eingelesen. Wie oben beschrieben, wird diese Revision nun als Build-Nummer in einer neuen, per Parameter festgelegten Property abgelegt. Zusätzlich gibt das Makro die Build-Nummer mit der <echo>-Funktion auf der Konsole aus. Am Ende des Makros wird schließlich die temporäre XML-Datei wieder gelöscht.

Genau wie die gemeinsam genutzten Ziele wird auch das neue Makro in der Datei common.xml angelegt. Mit der Verwendung des Makros im Integrationsskript beschäftigen wir uns im nächsten Abschnitt.

Kennzeichnung des Produktes mit der Build-Nummer

Die ermittelte Build-Nummer wird zur eindeutigen Kennzeichnung der internen *e2etrace*-Auslieferungen verwendet. Üblicherweise markiere ich ein ausgeliefertes Produkt auf zwei Arten. Zum einen integriere ich die Build-Nummer in den Namen der Installationsdatei. Oft werden die Ergebnisse eines Integrations-Builds in einem temporären Archiv

abgelegt oder an Testanwender ausgeliefert. Die Build-Nummer im Namen hilft, hier den Überblick zu bewahren.

Die Anwender selbst bekommen die Installationsdatei jedoch meist gar nicht zu Gesicht, daher sollte die Build-Nummer zusätzlich in das Produkt selbst eingebettet werden. In klassischen Anwendungen mit einer Benutzeroberfläche kann die Build-Nummer beispielsweise in einem Info-Dialog angezeigt werden. Da *e2etrace* als Bibliothek keine GUI hat, integriere ich die Build-Nummer in eine Datei `RELEASE-NOTES.txt` mit allgemeinen Informationen zum aktuellen Release von *e2etrace*.

Beginnen wir mit dem einfacheren Teil und ergänzen den Namen der Installationsdatei mit der Build-Nummer. Hierzu ist eine kleine Erweiterung des `install`-Ziels im Skript notwendig:

Listing 25
Build-Nummer als Teil des Namens der Installationsdatei verwenden

```
...
<!-- Build-Nummer ermitteln -->
<getbuildnr buildnrproperty="build.number"/>

<!-- Installationsdatei erstellen -->
<antcall target="-install" />

<!-- Installationsdatei ausliefern -->
<copy file="${target.install}\e2etrace.zip"
      tofile="${install.archive}\
        e2etrace-${svn.branch}-build#${build.number}.zip" />
...
```

In der bisherigen Version wurde an dieser Stelle lediglich das private Ziel `–install` aus `common.xml` aufgerufen. Jetzt ermitteln wir zusätzlich über das Makro `<getbuildnr>` die Build-Nummer. Wie Sie sehen, werden Makros wie »normale« Ant-Funktionen verwendet. Der Parameter `buildnrproperty` wird im Beispiel mit dem Wert `build.number` belegt. Nach der Ausführung des Makros ist die Build-Nummer daher in der Property `${build.number}` hinterlegt. Diese Property verwenden wir im nächsten Abschnitt als Teil des Namens der Installationsdatei.

Auslieferung der Installationsdatei

Im Falle von *e2etrace* ist für die Auslieferung der Installationsdatei die `<copy>`-Funktion ausreichend. Am Ende von Listing 25 wird die vom Ziel `–install` erzeugte Datei `e2etrace.zip` in das Auslieferungsarchiv `${install.archive}` kopiert und gleichzeitig umbenannt. Ein Teil des Namens bildet hierbei die oben erwähnte Build-Nummer. Zusätzlich füge ich den Namen des aktuellen Entwicklungspfades mit der Property `${svn.branch}` in den Dateinamen ein. Zwingend notwendig ist dies nicht, da die Build-Nummern auch über die einzelnen Entwick-

lungspfade hinweg eindeutig sind. Es erleichtert jedoch den Umgang mit dem Auslieferungsarchiv, wenn direkt aus den Dateinamen auf den jeweils neuesten Build in einem bestimmten Branch geschlossen werden kann.

Die neuen Properties `${install.archive}` und `${svn.branch}` habe ich in den `build.properties` wie folgt definiert:

```
# Verzeichnis des Auslieferungsarchivs
install.archive=D:/Projekte/e2etrace/install-archive
# Branch-Name
svn.branch=trunk
```

Im Beispiel verwende ich für das Auslieferungsarchiv ein lokales Verzeichnis. Normalerweise wird man im Projekt stattdessen ein allgemein zugängliches Netzlaufwerk verwenden. Der Branch-Name lautet im Listing `trunk`. Sobald ein neuer Release-Branch angelegt wird, muss man in diesem Branch die Property ändern, z. B. in `RB-1.1.0`. Bei der späteren Zusammenführung des Release-Branches mit dem *trunk* ist dann darauf zu achten, dass die Änderung in den `build.properties` nicht zurück in den *trunk* übertragen wird.

Build-Nummer in eine Textdatei einbetten

Um die Build-Nummer zusätzlich in die Datei `RELEASE-NOTES.txt` einzubetten, sind etwas umfangreichere Änderungen an den Skripten notwendig. Bisher wurden die ASCII-Textdateien mit den Release- und Lizenzinformationen vom Build-Ziel `–install` direkt aus dem Verzeichnis `doc` in die Installationsdatei übernommen. Dies gilt es nun zu ändern, da zumindest die Datei mit den Release-Infos während der Erstellung der Installationsdatei verändert wird. Wir führen daher ein neues Verzeichnis `target\doc` ein. In den `build.properties` und in `common.xml` sind folgende Änderungen notwendig, um das neue Verzeichnis in den bestehenden Build-Prozess einzubinden:

- Definition einer neuen Property `${target.doc}` mit dem Wert »`${target}/doc`« in den `build.properties`.
- Erweiterung des Build-Ziels `–prepare` in `common.xml` um einen neuen `<mkdir>`-Aufruf für das neue Zielverzeichnis.
- Anpassung des Build-Ziels `–install` in `common.xml` an das neue Verzeichnis. Die Textdateien werden jetzt aus dem Verzeichnis `${target.doc}` statt wie bisher aus `${src.doc}` gelesen.

Nachdem die Erweiterungen im Basisskript durchgeführt wurden, können wir das Ziel `install` im Integrationsskript nochmals ausbauen. Da die Textdateien jetzt im Verzeichnis `target\doc` erwartet werden,

müssen wir sie vor der Erstellung der Installationsdatei dorthin kopieren. Praktischerweise unterstützt die <copy>-Funktion von Ant gleichzeitig das Ersetzen von Schlüsselwörtern durch beliebige Werte. Diese Möglichkeit nutzen wir, um während des Kopiervorgangs das Schlüsselwort @BUILDNR@ in den Textdateien durch die ermittelte Build-Nummer zu ersetzen. Listing 26 zeigt das entsprechend erweiterte Build-Ziel install im Integrationsskript.

Listing 26
Build-Ziel zur Erstellung der Installationdatei im Integrationsskript

```
<!-- ==== Installationsdatei erstellen ==== -->
<target name="install"
        depends="clean, prepare"
        description="Erstellt die Installationsdatei">
  (...)

  <!-- Build-Nummer ermitteln -->
  <getbuildnr buildnrproperty="build.number"/>

  <!-- Build-Nummer in Textdateien einbetten -->
  <copy todir="${target.doc}">
    <fileset dir="${src.doc}">
      <include name="*.txt"/>
    </fileset>
    <filterset>
       <filter token="BUILDNR" value="${build.number}"/>
    </filterset>
  </copy>

  <!-- Installationsdatei erstellen -->
  <antcall target="-install" />

  (...)

</target>
```

Die Ersetzung des Schlüsselwortes wird über das <filterset>-Element der <copy>-Funktion gesteuert. Mit dem Attribut token gibt man das Schlüsselwort ohne die umrahmenden @-Zeichen an, mit value den einzusetzenden Wert.

Benachrichtigung der Teammitglieder per E-Mail

Da der Integrations-Build automatisch ausgeführt wird, sollten wir das Team in geeigneter Weise über das Ergebnis des Skriptes informieren. Eine Möglichkeit hierfür stellt die Veröffentlichung des Build-Status auf der Projekt-Homepage dar. Noch besser finde ich persönlich jedoch die Benachrichtigung des Teams mit einer Status-Mail. Denn im Gegensatz zur Projekt-Homepage wird die Mailbox unter Garantie von jedem Teammitglied regelmäßig abgerufen.

Versenden der Mails über einen Ant-Logger

Ant stellt zwei unterschiedliche Mechanismen zum Verschicken des Build-Status per Mail bereit. Von Haus aus mitgeliefert werden die

sogenannten *Logger*, welche die gesamte Ausgabe eines Build-Skripts festhalten. Standardmäßig ist der *DefaultLogger* aktiv, der schlicht alles auf der Konsole ausgibt. Ersetzt man diesen mit dem ebenfalls mitgelieferten *MailLogger*, werden alle Ausgaben zusätzlich gesammelt und am Ende des Build-Skriptes per Mail verschickt. Mit dem *MailLogger* könnten wir die Status-Mails im Prinzip umsetzen, ich habe mich jedoch trotzdem für eine andere Art der Implementierung entschieden. Ausschlaggebend sind hierfür zum einen didaktische Gründe. Am Beispiel der Status-Mail lässt sich die bisher von mir vernachlässigte *bedingte Ausführung* von Build-Zielen sehr schön erklären. Zum anderen sollen die Status-Mails eine Reihe von Anforderungen umsetzen, mit denen der *MailLogger* überfordert wäre:

Anforderungen an den E-Mail-Versand

- Abhängig vom Build-Status sollen unterschiedliche Mail-Inhalte und -Empfänger verwendet werden.
- Wenn Modultests fehlschlagen, sollen die Reporte der fehlgeschlagenen Testfälle im Text der Mail verschickt werden. Die Reporte erfolgreicher Testfälle werden hingegen nicht in die Mail aufgenommen.
- Aus dem Anhang der Mail muss der komplette Build-Status ersichtlich werden. Der Build-Status besteht aus den Logdateien mit den Ausgaben von Ant und Subversion.

Ich verwende statt des *MailLoggers* die zweite von Ant bereitgestellte Alternative zum Versenden von E-Mails: die Funktion `<mail>`. Um die obigen Anforderungen vollständig umzusetzen, sind allerdings noch einige zusätzliche Erweiterungen im Skript notwendig.

Prinzipielle Vorgehensweise zum Versenden der Status-Mail

Zunächst müssen wir uns über ein grundlegendes Problem Gedanken machen. Wie unterscheiden wir einen erfolgreichen von einem fehlgeschlagenen Build? Da Builds aus allen möglichen Gründen scheitern können, ist eine sichere Erkennung des Status erst nach der vollständigen Abarbeitung des Skriptes möglich. Schlägt ein Build fehl, liefert Ant einen Exit-Code ungleich null. Diesen Exit-Code können wir theoretisch in unserem Wrapper-Skript `build-int.cmd` abfragen. In der Praxis funktioniert dies jedoch leider erst ab Ant V1.7. Alle früheren Ant-Versionen liefern auf Grund eines Fehlers im Ant-Start-Skript zumindest unter Windows immer eine Null als Exit-Code .

Ermittlung des Build-Status aus der Protokolldatei

Wir haben jedoch noch andere Möglichkeiten, den Build-Status zu ermitteln, der für alle Ant-Versionen gleichermaßen funktioniert. Zudem ist er aus didaktischer Sicht wertvoller, da er mir Gelegenheit bietet, einige weitere interessante Ant-Funktionen vorzustellen.

Wer die Ausgaben von Ant aufmerksam beobachtet, wird feststellen, dass Ant fehlgeschlagene Builds immer mit der Meldung »BUILD FAILED« abschließt. Dieser String findet sich demnach auch in der Protokolldatei `build.log`. Zur Ermittlung des Build-Status müssen wir diese Datei daher lediglich auf das Vorkommen der obigen Meldung untersuchen.

Kurz zusammengefasst ergibt sich aus dem bisher Gesagten das folgende Konzept zur Implementierung der Status-Mails: Wir ermitteln den Build-Status erst nach der Ausführung des eigentlichen Integrations-Builds anhand der von Ant erstellten Logdatei. Die Status-Mail wird im Anschluss über ein neu zu erstellendes Build-Ziel `mailresult` verschickt. Im Wrapper-Skript `build-int.cmd` muss Ant also insgesamt zwei Mal ausgeführt werden. Der interne Ablauf von `mailresult` ist in Abbildung 4 dargestellt. Zunächst ermittelt das Ziel `–checkforfailure` das Ergebnis des Build-Laufes aus der Protokolldatei `build.log`. War der Build erfolgreich, wird das Ziel `–mailsuccess` ausgeführt und eine entsprechende Mail verschickt. Ist der Build hingegen fehlgeschlagen, verzweigt die Ausführung zu `–mailfailure`.

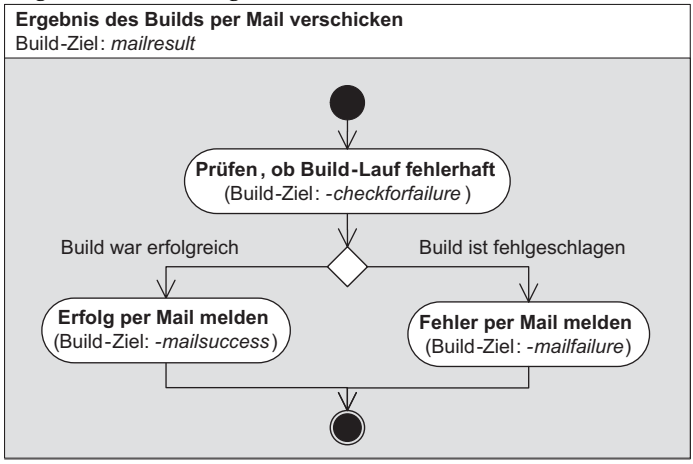


Abb. 4
Ablauf des Build-Ziels zum Verschicken des Ergebnisses per Mail

Ermittlung des Build-Status aus der Protokolldatei

Listing 27 zeigt die Implementierung des Ziels `–checkforfailure`. Durch das führende Minuszeichen wird es als privates Ziel im Skript

build-int.xml gekennzeichnet. Es kann also nicht direkt von der Kommandozeile aus aufgerufen werden.

Listing 27 *Implementierung des Ziels zur Ermittlung des Build-Status*

```
<!-- Prüfen, ob der Build fehlgeschlagen ist -->
<!-- Wenn ja, wird die Property ${mail.buildfailed}
     gesetzt -->
<target name="-checkforfailure">
  <condition property="mail.buildfailed">
    <isfileselected file="${target.log}/build.log">
      <contains text="BUILD FAILED"
                casesensitive="true"/>
    </isfileselected>
  </condition>
</target>
```

Mit der Funktion <condition> wird eine Property neu definiert, wenn die angegebenen Bedingungen erfüllt sind. Im Beispiel wird die Property ${mail.buildfailed} nur dann initialisiert, wenn in der Datei ${target.log}\build.log die Textfolge »BUILD FAILED« vorkommt. Zur Formulierung dieser Bedingung verwende ich die Kondition <isfileselected>. Diese überprüft, ob die angegebene Datei von dem als Subelement angegebenen Selektor ausgewählt wird. Der Selektor <contains> wiederum wählt eine Datei nur dann aus, wenn sie die angegebene Textfolge enthält.

Erfolgreichen Build per Mail melden

Das Verschicken der Erfolgs-Mail übernimmt das Build-Ziel -mailsuccess:

Listing 28 *Implementierung des Build-Ziels zum Verschicken der Erfolgs-Mail*

```
<!-- Erfolgreichen Build per Mail melden -->
<target name="-mailsuccess" unless="mail.buildfailed">
  <mail mailhost="localhost"
        mailport="25"
        subject="Integrations-Build war erfolgreich!">
    <from address="build-int@localhost" />
    <replyto address="poppgu@localhost" />
    <to address="poppgu@localhost" />

    <message>
    Der Integrations-Build wurde erfolgreich
    durchgeführt!
    </message>
    <fileset dir="${target.log}">
      <include name="*.*" />
    </fileset>
    <fileset dir="ant">
      <include name="~svnupdate.log" />
    </fileset>

  </mail>
```

```
</target>
```

Bei der Definition des <target>-Elements habe ich das bisher nicht verwendete Attribut unless angegeben. unless steuert, zusammen mit dem im nächsten Abschnitt vorgestellten Attribut if, die bedingte Ausführung von Build-Zielen. Im Beispiel bewirkt das Attribut, dass das Build-Ziel –mailsuccess nur dann ausgeführt wird, wenn die Property ${mail.buildfailed} *nicht* existiert. Da diese Property vom Ziel –checkforfailure nur dann initialisiert wird, wenn der Build fehlgeschlagen ist, wird –mailsuccess bei einem erfolgreichen Build ausgeführt.

Einsatz der mail-Funktion

Zum Versenden der Mail verwende ich die Funktion <mail>. Die Funktion benötigt zusätzliche Java-Bibliotheken, die ins lib-Verzeichnis der Ant-Installation kopiert werden müssen. Hierbei handelt es sich um mail.jar aus der *JavaMail-API*[15] und activation.jar aus dem *JavaBeans Activation Framework*.[16]

Über die Attribute mailhost und mailport wird der Funktion die Adresse eines SMTP-Servers mitgeteilt. Das Attribut subject legt die Betreffzeile der Mail fest. Die geschachtelten Elemente sind selbsterklärend, sie bestimmten Empfänger und Inhalt der Mail. Interessant ist hierbei die auch im Beispiel verwendete Möglichkeit, beliebige Dateien über <fileset>-Elemente im Anhang der Mail zu verschicken.

Fehlerhaften Build per Mail melden

E-Mails für fehlerhafte Integrations-Builds werden mit dem Ziel –mailfailure erzeugt. Zusätzlich soll dieses Ziel nach Testreporten von fehlgeschlagenen JUnit-Tests suchen und diese, sofern vorhanden, direkt in den Text der Nachricht einbetten. Das Kalkül hinter diesem Vorgehen ist, dass auf diese Weise jeder Empfänger der Mail unmissverständlich auf fehlerhafte Modultests hingewiesen wird. Damit hoffe ich, die Motivation für sorgfältigen Testentwurf und -erstellung im Team dauerhaft aufrechtzuerhalten. Listing 29 zeigt die vollständige Implementierung des Build-Ziels.

Listing 29
Implementierung des Ziels zum Melden fehlerhafter Integrations-Builds

```
<!-- Fehlgeschlagenen Build-Lauf melden -->
<target name="-mailfailure" if="mail.buildfailed">
  <concat destfile="~failuremail">
    <header>
       Fehlgeschlagene Unit-Tests (wenn keine Tests
       folgen, bitte build.log prüfen):
    </header>
    <fileset dir="${target.testreports}">
      <or>
```

15. *http://java.sun.com/products/javamail*
16. *http://java.sun.com/products/javabeans/glasgow/jaf.html*

```
        <contains text="FAILED" casesensitive="yes"/>
        <contains text="ERROR" casesensitive="yes"/>
      </or>
    </fileset>
  </concat>

  <mail mailhost="localhost"
        mailport="25"
        subject="!!! Integrations-Build war NICHT
                 erfolgreich !!!">
    <from address="build-int@localhost" />
    <replyto address="poppgu@localhost" />
    <to address="poppgu@localhost" />
    <to address="mhr@localhost" />

    <message src="~failuremail"/>

    <fileset dir="${target.log}">
      <include name="*.*" />
    </fileset>
    <fileset dir="ant">
      <include name="~svnupdate.log" />
    </fileset>

  </mail>

  <delete file="~failuremail"/>

</target>
```

Das Ziel –mailfailure wird nur dann ausgeführt, wenn die Property ${mail.buildfailed} existiert (siehe Attribut if im Listing). Es stellt damit das Gegenstück zu –mailsuccess dar.

Gleich zu Beginn der Implementierung wird mit der Funktion <concat> der Text der Fehler-Mail generiert. Dieser soll aus den Testreporten aller fehlgeschlagenen JUnit-Tests bestehen. <concat> macht nichts anderes, als die Inhalte aller durch das geschachtelte <fileset>-Element ausgewählten Dateien aneinanderzuhängen und in einer Zieldatei zu speichern. Im Beispiel ist dies die mit dem Attribut destfile angegebene, temporäre Datei ~failuremail.

Logische Verknüpfung von Selektoren

Das <fileset>-Element verwendet zwei mittels logischem *ODER* verknüpfte Selektoren zur Auswahl der Testreporte. Die Selektoren prüfen, ob in einer Datei die Textfolge FAILED oder ERROR vorkommt. Ist dies der Fall, handelt es sich um den Report eines fehlgeschlagenen JUnit-Tests, und die Datei wird selektiert. Letzten Endes erstellt diese Kombination aus <concat>, <fileset>, <or> und <contains> eine Datei ~failuremail, in der die Reporte aller fehlerhaften JUnit-Tests zusammengefasst sind. Mit dem <header>-Element wird zusätzlich an den Anfang der Datei ein kurzer Erläuterungstext geschrieben.

Die Fehler-Mail wird wieder mit der Funktion `<mail>` gesendet. Im Beispiel entspricht der Code weitgehend dem aus `–mailsuccess`. Als Nachrichtentext wird diesmal jedoch der Inhalt von `~failuremail` verwendet. Zudem verwende ich unterschiedliche Empfänger und Betreffzeilen für Erfolgs- und Fehler-Mails.

Fertigstellung des Integrationsskriptes

Zur endgültigen Fertigstellung des Integrationsskriptes fehlt nur noch das übergeordnete Build-Ziel `mailresult`:

Listing 30
Übergeordnetes Ziel zum Versenden des Build-Ergebnisses per Mail

```
<!-- ==== Ergebnis per Mail verschicken ==== -->
<target name="mailresult"
        depends="-checkforfailure,
                 -mailsuccess,
                 -mailfailure"
        description="Versendet eine Mail mit dem
                     Ergebnis des Builds">

</target>
```

Dieses Ziel hat keinerlei Implementierung. Vielmehr werden über das `depends`-Attribut lediglich die Abhängigkeiten zu den untergeordneten Zielen `-checkforfailure`, `-mailsuccess` und `-mailfailure` hergestellt. Ant ruft die Ziele in der angegebenen Reihenfolge auf und berücksichtigt hierbei die mittels `if` und `unless` festgelegten Ablaufbedingungen.

Zeitgesteuerte Ausführung des Skriptes

Der Integrations-Build wird unter Windows über das Wrapper-Shell-Skript `build-int.cmd` gestartet[17]. In Listing 31 ist die aktualisierte Fassung dargestellt, die nach der Ausführung des eigentlichen Builds zusätzlich das Ergebnis per Mail versendet.

Listing 31
Aktualisierte Fassung des Wrapper-Shell-Skriptes für Windows

```
@echo off
cd ..
svn update > ant\~svnupdate.log
cd ant
call ant -f build-int.xml install
call ant -f build-int.xml mailresult
del ~svnupdate.log
```

Die automatische, zeitgesteuerte Ausführung von `build-int.cmd` übernimmt nun ein *Task-Scheduler*. Unter einem aktuellen Windows-System kann hierfür beispielsweise das Kommandozeilen-Tool `schtasks` eingesetzt werden:

17. Eine entsprechende Variante für Unix finden Sie auf der Webseite zum Buch.

Automatische Ausführung mit einem Task-Scheduler

```
> schtasks /create /tn Integrations-Build   \
      /tr d:\projekte\e2etrace\trunk\ant\build-int.cmd \
      /sc TÄGLICH /st 19:00:00
ERFOLGREICH: Der geplante Task "Integrations-Build" wurde
erfolgreich erstellt.
```

Der obige Befehl sorgt dafür, dass das Skript `build-int.cmd` täglich um 19:00 Uhr gestartet wird. Unter Unix steht mit `crontab` ein vergleichbares Werkzeug zur Verfügung.

Release-Build

Mit dieser Build-Variante wird ein »offizielles« Release des Produktes erstellt. Theoretisch könnte hierfür, analog zum Integrations-Build, ein aktueller Stand aus dem Repository abgerufen und daraus die Installationsdatei für das Release generiert werden. Meiner Ansicht nach ist diese Vorgehensweise aber zu riskant. Trotz aller Sorgfalt kann nicht ausgeschlossen werden, dass quasi kurz vor Schluss ungeprüfte Änderungen in das Repository und damit auch in das Release wandern. Es ist daher sicherer, eine durch das Integrationsskript erstellte und bereits geprüfte Installationsdatei als Grundlage für das neue Release zu verwenden.

Prinzipieller Ablauf zur Erstellung eines Release

Abbildung 5 auf Seite 54 zeigt den kompletten Ablauf zur Erstellung eines Release, unterteilt in fünf Abschnitte. Die Beispiele in der Abbildung beziehen sich auf ein neues Release 1.1.0 von *e2etrace*. Zur Vorbereitung des Release werden in Abschnitt eins ein Tag *PREP-1.1.0* und der Release-Branch erzeugt[18]. In Abschnitt zwei wird der Integrations-Build im Release-Branch durchgeführt. Sobald die erzeugte Installationsdatei alle Qualitätskriterien erfüllt, wird die entsprechende Build-Nummer in Block drei zur Auslieferung freigegeben. Erst jetzt, im vierten Block, kommt das noch zu erstellende Release-Skript zum Einsatz. Dieses erstellt auf Basis der freigegebenen Build-Nummer das neue Release von *e2etrace*. Der letzte, fünfte Block des Prozesses wird wieder manuell ausgeführt.

18. Die genauen Gründe für die Erstellung des Subversion-Tags erläutere ich in Kapitel 4 des gedruckten Buches.

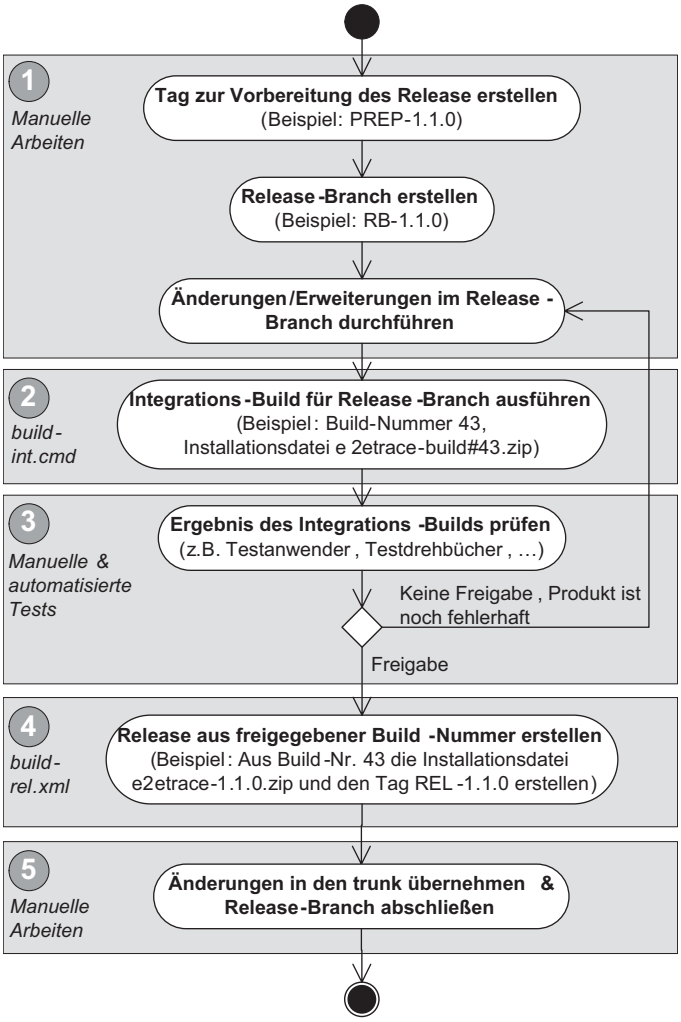


Abb. 5
Prozess zur Erstellung eines Release

Implementierung des Release-Skriptes

Das Release-Skript setzt auf den Ergebnissen eines zuvor ausgeführten Integrations-Builds auf. Welcher Integrations-Build die Grundlage für das neue Release bildet, wird dem Skript über die Build-Nummer mitgeteilt. Für *e2etrace* nehmen wir an, dass das neue Release 1.1.0 auf Build-Nummer 43 basieren soll. Das Skript muss nun den Release-Tag *REL-1.1.0* im Repository erzeugen – und zwar auf Basis der zur Build-

Nummer passenden Revision. Weiterhin muss das Skript sicherstellen, dass der Release-Tag im Repository exakt zur ausgelieferten Installationsdatei passt. Diese Installationsdatei ist nichts anderes als eine Kopie der Datei von Build 43 aus dem Auslieferungsarchiv. Der etwas gekürzte Code des Release-Skriptes ist in Listing 32 dargestellt.

Listing 32
Implementierung des Release-Skriptes

```
<?xml version="1.0"?>
<project name="e2etrace-rel" basedir=".."
         default="install">
  <description>
    Release-Build-Skript für e2etrace
  </description>

  <import file="common.xml" />

  <!-- ==== Parameter überprüfen ==== -->
  <fail message="Beim Aufruf des Skripts muss via
                 -Dinstall.build.number=xxxx die
                 Build-Nummer des Release-Builds
                 angegeben werden!"
        unless="install.build.number"/>

  ... entsprechende Prüfungen für release.number,
      admin.name und admin.password ...

  <!-- ==== Release erstellen ==== -->
  <target name="install">
    <exec executable="svn" failonerror="true">
      <arg value="copy" />
      <arg value="--message" />
      <arg value="Erstelle Tag REL-${release.number}"/>
      <arg value="--username"/>
      <arg value="${admin.name}"/>
      <arg value="--password"/>
      <arg value="${admin.password}"/>
      <arg value="--revision"/>
      <arg value="${install.build.number}"/>
      <arg value="${svn.url}/branches/
                  RB-${release.number}"/>
      <arg value="${svn.url}/tags/
                 REL-${release.number}"/>
    </exec>

    <copy file="${install.archive}\
               e2etrace-${svn.branch}-
               build#${install.build.number}.zip"
          tofile="${release.archive}\
                   e2etrace-${release.number}.zip"/>

    <echo>Release ${release.number} wurde erfolgreich
          erstellt.</echo>
    <echo>Tag im Repository :
```

```
            ${svn.url}/tags/REL-${release.number}</echo>
        <echo>Installationsdatei: ${release.archive}/
            e2etrace-${release.number}.zip</echo>
    </target>
</project>
```

Das Skript enthält nur ein einziges Build-Ziel `install`. Im `<project>`-Element geben wir dieses Ziel mit Hilfe des Attributes `default` als Standardziel des Skriptes an. Wir können dann bei der Ausführung auf die Angabe eines Build-Zieles verzichten.

Stattdessen müssen dem Skript `build-rel.xml` allerdings eine Reihe von Parametern übergeben werden (siehe Tab. 3). Dies erfolgt bei der Ausführung von Ant generell mit der folgenden Syntax:

```
ant ... –D<parameter>=<wert>.
```

Tab. 3
Parameter für das Release-Skript

Parameter	Beschreibung
install.build.number	Build-Nummer des Integrations-Builds, der als Basis für das Release verwendet werden soll. Beispiel: -Dinstall.build.number=43
release.number	Release-Nummer in der Form x.x.x. Beispiel: -Drelease.number=1.1.0
admin.name	Name eines Subversion-Anwenders mit Admin-Berechtigung. Dieser Anwender wird verwendet, um den Release-Tag im Repository zu erstellen. Beispiel: -Dadmin.name=root
admin.password	Kennwort des obigen Anwenders Beispiel: -Dadmin.password=root

Im Skript wird zu Beginn überprüft, ob beim Aufruf wirklich alle Parameter angegeben wurden. Hierzu verwende ich die `<fail>`-Funktion zusammen mit dem Attribut `unless`. `<fail>` wird also nur dann ausgeführt, wenn der entsprechende Parameter beim Aufruf nicht übergeben wurde.

Aufruf des Subversion-Clients

Das Ziel `install` erstellt im ersten Schritt den Release-Tag im Repository. Hierzu wird mittels <exec> der Subversion-Client mit einer langen Liste von Parametern ausgeführt. Der Aufruf im Skript entspricht der folgenden Kommandozeile:

```
svn copy --message "Erstelle Tag REL-${release.number}" \
         --username ${admin.name}                       \
         --password ${admin.password}                   \
         --revision ${install.build.number}             \
         --${svn.url}/branches/RB-${release.number}     \
         --${svn.url}/tags/REL-${release.number}
```

Da die Funktion <exec> mit dem Parameter `failonerror=true` ausgeführt wird, bricht das Skript ab, wenn die Erstellung des Tags durch den Subversion-Client scheitert. Wurde der Tag hingegen erfolgreich erzeugt, kopiert das Skript im Anschluss die zur Build-Nummer passende Installationsdatei aus dem Auslieferungsarchiv in das Release-Archiv. Der Name der Datei wird hierbei an die Release-Nummer angepasst. Am Ende des Skriptes werden die wichtigsten Daten des neuen Release mit Hilfe der <echo>-Funktion ausgegeben.

Die im Skript neu eingeführten Properties sind in der Datei `build.properties` wie folgt definiert:

```
# Verzeichnis des Release-Archives
release.archive=D:/Projekte/e2etrace/release-archive
# Subversion URLs
svn.url=svn://localhost/e2etrace
```

Erstellung eines neuen Release mit dem Skript

Das fertige Release-Skript kann nun beispielsweise wie folgt ausgeführt werden:

```
> ant -f build-rel.xml -Dinstall.build.number=43 \
      -Drelease.number=1.1.0 -Dadmin.name=root   \
      -Dadmin.password=root
Buildfile: build-rel.xml

install:

     [exec] Revision 47 übertragen.
     [copy] Copying 1 file to
            D:\Projekte\e2etrace\release-archive
     [echo] Release 1.1.0 wurde erfolgreich erstellt.
     [echo] Tag im Repository :
            svn://localhost/e2etrace/tags/REL-1.1.0
     [echo] Installationsdatei: D:/Projekte/e2etrace/
            release-archive/e2etrace-1.1.0.zip

BUILD SUCCESSFUL
Total time: 1 second
```

Qualitätssicherung durch Audits und Metriken

Zusätzlich zur funktionalen Absicherung von e2etrace durch die Modultests möchte ich eine Reihe von automatisierten Audits mit Hilfe eines Werkzeuges zur statischen Quelltextanalyse durchführen. Hierbei sollen folgende Ziele erreicht werden:

Ziele der Qualitätssicherung durch automatisierte Audits

1. In gedruckten Buch wurden folgende Namenstemplates für das Konfigurationselement *Java-Quelltext* festgelegt: *<Klassenname>.java* für normale Klassen, *I<Interfacename>.java* für Schnittstellen und *Abstract<Klassenname>.java* für abstrakte Klassen. Die Einhaltung dieser Vorgaben soll geprüft werden.
2. Die Dokumentation von e2etrace wird aus den JavaDoc-Kommentaren im Quelltext erzeugt. Daher muss sichergestellt werden, dass wirklich für jede Klasse, Schnittstelle und jede öffentliche Methode ein vollständiger JavaDoc-Kommentar vorliegt.
3. Der Quelltext von e2etrace soll einfach zu verstehen und zu erweitern sein. Dies lässt sich im Detail natürlich nicht durch eine Automatik sicherstellen. Wir können aber zumindest nach Hinweisen suchen, ob übermäßig komplexe Methoden im Quelltext existieren.

Auswahl der geeigneten Checkstyle-Module

Die statische Codeanalyse wird mit dem Werkzeug Checkstyle durchgeführt. Dieses bietet insgesamt 127 verschiedene Audits und Metriken in Form von *Prüfmodulen* an. Aus diesem umfangreichen Angebot wählen wir nun mit Hilfe des GQM-Verfahrens die für e2etrace geeigneten Module aus. Tabelle 4 dokumentiert diese Entscheidungsfindung:

Tab. 4
Auswahl der geeigneten Checkstyle-Prüfungen

Ziel	Fragen	Checkstyle-Modul
Einhaltung der Namenskonventionen	Beginnen die Namen aller abstrakten Klassen mit dem Präfix »Abstract«?	`AbstractClassName`
	Beginnen die Namen aller Schnittstellen mit dem Präfix »I«?	`TypeName`
Vollständige JavaDoc-Kommentare für Klassen, Schnittstellen und öffentliche Methoden	Existiert für jede Klasse und Schnittstelle ein JavaDoc-Kommentar?	`JavadocType`
	Existiert für jedes Package von e2etrace eine Datei `package.html`?	`PackageHtml`
	Existiert für jede öffentliche Methode ein JavaDoc-Kommentar?	`JavadocMethod`
	Sind die JavaDoc-Kommentare vollständig?	`JavadocStyle`

Verständlichkeit und Erweiterbarkeit des Quelltextes	Gibt es außergewöhnlich komplexe Methoden im Quelltext?	`Cyclomatic Complexity` (Maximale CC=10)

Die konkrete Auswahl der geeigneten Prüfmodule habe ich anhand der Checkstyle-Dokumentation durchgeführt. Hier ist genau beschrieben, welche Kriterien des Quelltextes durch die einzelnen Module überprüft werden. Es lohnt sich in jedem Fall, diese Dokumentation auf der Checkstyle-Webseite kurz durchzublättern. Wie ich im gedruckten Buch ausführlich erläutere, sollte man jedoch keinesfalls der Versuchung erliegen und einfach »so viel wie möglich« Prüfmodule verwenden. Die Folge wären mit Sicherheit lange Listen von »Verstößen«, in denen die wirklich wichtigen Hinweise dann untergehen.

Installation und Konfiguration

Um Checkstyle in einem Ant-Skript aufrufen zu können, müssen wir zunächst die Installation von Ant anpassen. Hierzu wird die Datei `checkstyle-all-4.3.jar` aus dem Installationsverzeichnis von Checkstyle in das Unterverzeichnis `lib` des Installationsverzeichnisses von Ant kopiert.

Im nächsten Schritt erstellen wir die in Listing 33 gezeigte Checkstyle-Konfigurationsdatei, in der die für e2etrace geplanten Audits und Metriken festgelegt werden. Die Datei bekommt den Namen `checkstyle-config.xml` und wird in der Projektstruktur im Verzeichnis `ant` abgelegt.

Listing 33 *Checkstyle-Konfiguration für e2etrace*

```
<?xml version="1.0"?>
<!DOCTYPE module PUBLIC
 "-//Puppy Crawl//DTD Check Configuration 1.2//EN"
 "http://www.puppycrawl.com/dtds/configuration_1_2.dtd">
<!-- Checkstyle-Konfiguration für e2etrace -->
<module name="Checker">
  <module name="PackageHtml"/>
  <module name="TreeWalker">
    <module name="AbstractClassName"/>
    <module name="TypeName">
      <property name="format"
                value="^I[A-Z][a-zA-Z0-9]*$"/>
      <property name="tokens"
                value="INTERFACE_DEF"/>
    </module>
    <module name="JavadocMethod"/>
    <module name="JavadocType"/>
    <module name="JavadocStyle"/>
    <module name="CyclomaticComplexity"/>
```

```
    </module>
  </module>
```

Eine Checkstyle-Konfiguration besteht aus einer XML-Datei mit geschachtelten `module`-Elementen. Das oberste Element trägt immer den Namen `Checker`. Unterhalb erfolgt dann die Festlegung, welche Audits und Metriken von Checkstyle ausgeführt werden sollen. Die Namen der Module in der Konfigurationsdatei entsprechen denen aus Tabelle 4.

Hierarchie der Prüfmodule

Neben dem Namen eines Moduls muss man allerdings auch dessen Einordnung in der internen Modulhierarchie kennen. Im Listing wird beispielsweise eine Reihe von Modulen unterhalb des Moduls `TreeWalker` eingeordnet. Die Information, welches Modul wo in der Hierarchie steht, erhält man aus der Checkstyle-Dokumentation. Hier wird für jedes Modul ein *Parent* angegeben. Im Fall von `PackageHtml` ist der Parent das Root-Modul `Checker`, im Fall von `AbstractClassName` ist es eben der `TreeWalker`.[19]

Festlegung von Parametern und Schwellwerten

Die Parameter und Schwellwerte der einzelnen Module können mit Hilfe von *Properties* an die eigenen Bedürfnisse angepasst werden. Welche Properties möglich sind und wie die Standardwerte lauten, ist pro Modul ebenfalls in der Online-Dokumentation beschrieben. Im Listing musste ich lediglich für das Modul `TypeName` eigene Properties festlegen, in allen anderen Fällen waren die Standardwerte ausreichend. Das Modul `TypeName` prüft, ob die Namen von Klassen und Schnittstellen einem vorgegebenen Muster entsprechen. Standardmäßig sind alle Namen zulässig, die mit einem Großbuchstaben beginnen. Unsere Konvention lautet jedoch, dass Schnittstellen immer nach dem Muster *I<Schnittstellenname>* benannt werden müssen. Um das Standardverhalten des Moduls zu ändern, habe ich die beiden Properties `format` und `tokens` definiert. Über `format` gebe ich eine *Regular Expression* an, die das erwünschte Namensmuster beschreibt. Mit `tokens` lege ich fest, dass das Muster nur für Schnittstellennamen anzuwenden ist. Das Muster für Klassennamen musste ich nicht ändern, hier entspricht die Standardeinstellung des Moduls meinen Vorstellungen.

19. Die Hierarchie wird durch die Implementierung der Prüfalgorithmen vorgegeben. Das Modul `TreeWalker` ist in der Lage, Java-Quelltext einzulesen und die einzelnen Programmkonstrukte in einer internen Datenstruktur abzulegen. Die untergeordneten Prüfmodule greifen dann nicht mehr auf den eigentlichen Quelltext, sondern lediglich auf die Datenstruktur zu. Dies erleichtert die Erstellung von neuen Prüfmodulen enorm.

Umgang mit fehlgeschlagenen Audits

Da Audits der Qualitätssicherung im Projekt dienen, muss ein Verstoß gegen eine der definierten Richtlinien spürbare Auswirkungen haben. Werden die Ergebnisse der automatisierten Audits nur als unverbindliche Hinweise kommuniziert, verlieren sie erfahrungsgemäß innerhalb kürzester Zeit jegliche Bedeutung im Projekt. Daher müssen wir Audits letztlich genauso wie Modultests behandeln und im Fehlerfall den Build-Prozess abbrechen.

Dieser Ansatz funktioniert allerdings nur dann, wenn wirklich nur die für die Softwarequalität kritischen Prüfungen durchgeführt werden. Checkstyle bietet beispielsweise auch Audits an, welche die Einhaltung von Formatierungsregeln verifizieren. Es ist nun sicherlich nicht angemessen, einen Integrations-Build nur wegen einer falsch positionierten Klammer scheitern zu lassen.

Ähnliches gilt für Audits, die keine eindeutigen Aussagen, sondern nur Hinweise auf potenzielle Probleme liefern. Ein Beispiel hierfür ist das auch in der e2etrace-Konfigurationsdatei verwendete Modul `CyclomaticComplexity`. Standardmäßig schlägt dieser Audit fehl, wenn für eine Methode ein CC-Wert größer als zehn ermittelt wird. Auch wenn dieser Fall eintritt, sollte deswegen nicht der Integrations-Build abgebrochen werden. Eventuell ist der implementierte Algorithmus schlicht und einfach komplex. Eine Aufteilung in mehrere kleinere Methoden vereinfacht dann nicht unbedingt das Verständnis des Quelltextes – und darum geht es uns ja schließlich. Für solche Situationen kann in der Konfigurationsdatei festgelegt werden, welche Auswirkungen ein fehlgeschlagenes Audit hat. Standardmäßig gilt jeder Verstoß als Fehler, der im Zweifelsfall auch zu einem Abbruch des Build-Skriptes führt. Mit der Property `severity` kann man diese Einstellung ändern. Über die folgende Erweiterung in der Konfigurationsdatei lege ich fest, dass ein zu hoher CC-Wert lediglich als Warnung interpretiert wird:

```
<module name="CyclomaticComplexity">
  <property name="severity" value="warning"/>
</module>
```

Listing 34
Anpassung des Schärfegrades eines Audits

Integration in den Build-Prozess

Die Checkstyle-Audits sollen sowohl im Entwickler-Build als auch im Integrations-Build durchgeführt werden. Daher implementieren wir das neue Build-Ziel in der Datei `common.xml`:

```
<!-- ==== Checkstyle ausführen -->
<target name="-checkstyle">
```

Listing 35
Implementierung des Build-Ziels zur Ausführung von Checkstyle

```
  <!-- checkstyle-Funktion definieren -->
  <taskdef resource="checkstyletask.properties"/>

  <!-- Checkstyle ausführen -->
  <checkstyle config="ant/checkstyle-config.xml"
              failureProperty="-cs.failure"
              failOnViolation="false">
    <fileset refid="JavaQuelltext"/>
    <formatter type="xml"
               tofile="${target.checkstyle}/
                       checkstyle_report.xml"/>
  </checkstyle>

 <style in="${target.checkstyle}/checkstyle_report.xml"
        out="${target.checkstyle}/checkstyle_report.html"
        style="${checkstyleInstall}/contrib/
               checkstyle-noframes-sorted.xsl"/>

 <!-- Skript ggf. abbrechen -->
 <condition property="-cs.callfail">
   <isset property="-cs.failure" />
 </condition>

 <fail message="Checkstyle-Audits sind fehlgeschlagen!"
       if="-cs.callfail" />

</target>
```

Da Ant von Haus aus keine Funktion zum Aufruf von Checkstyle kennt, muss diese am Anfang des Listings mit Hilfe des `<taskdef>`-Elements zunächst definiert werden. Die eigentliche Implementierung der `<checkstyle>`-Funktion wird zusammen mit Checkstyle ausgeliefert. `<taskdef>` macht nichts anderes, als diese Implementierung in das Build-Skript einzubinden.

Im nächsten Schritt wird die neue Funktion aufgerufen. Die vollständige Dokumentation aller Parameter von `<checkstyle>` finden Sie auf der Checkstyle-Webseite. Die im Listing verwendeten Parameter sind jedoch für viele Anwendungsfälle ausreichend. Mit dem Parameter `config` übergeben wir der Funktion den Pfad und Dateinamen der zuvor erstellten Konfigurationsdatei. Die Parameter `failureProperty` und `failOnViolation` kennen wir im Prinzip schon von der `<junit>`-Funktion. Über `failOnViolation` legen wir fest, dass fehlgeschlagene Audits nicht zum sofortigen Abbruch des Skriptes führen. Stattdessen wird in diesem Fall die über `failureProperty` angegebene Property `-cs.failure` neu initialisiert. Der eigentliche Abbruch erfolgt dann über die bereits bekannten Funktionen `<condition>` und `<fail>` am Ende des Listings.

Die beiden geschachtelten Parameter der `<checkstyle>`-Funktion legen zum einen über ein Fileset die zu prüfenden Quelldateien fest und

zum anderen das Format für die Audit-Ergebnisse. Im Listing werden die Ergebnisse in eine XML-Datei im Verzeichnis `${target.checkstyle}` geschrieben. Diese neue Property habe ich in den `build.properties` wie folgt definiert:

```
target.checkstyle=${target}/checkstyle
```

Erzeugung eines Berichtes mit den Audit-Ergebnissen

Natürlich ist ein XML-Dokument zur Präsentation der Audit-Ergebnisse denkbar ungeeignet. Daher generiere ich aus den XML-Daten mit Hilfe der `<style>`-Funktion einen hübsch formatierten HTML-Bericht.

`<style>` erzeugt mit Hilfe eines XSLT-Prozessors aus einer XML-Datei mehr oder weniger beliebige Ausgabeformate. Der Inhalt und das Format der Ausgabedatei werden über ein sogenanntes *Stylesheet* in Form einer XSL-Datei[20] festgelegt. Im Listing verwende ich die XSL-Datei `${checkstyleInstall}\contrib\checkstyle-noframes-sorted.xsl`, die zusammen mit Checkstyle ausgeliefert wird. Die Property `${checkstyleInstall}` verweist hierbei auf das Installationsverzeichnis von Checkstyle. Da Checkstyle, genau wie JUnit und commons-logging, eine projektexterne Bibliothek ist, erfolgt die Definition von `${checkstyleInstall}` in der in Abschnitt eingeführten, entwicklerspezifischen Property-Datei. Listing 36 zeigt den neuen Inhalt der Datei für meine Entwicklungsumgebung.

Listing 36
Erweiterte entwicklerspezifische Property-Datei

```
# Entwicklerspezifische Build-Properties für
# das e2etrace-Build-Skript

# commons.logging Installationspfad
loggingInstall=D:/ApacheGroup/commons-logging-1.0.4

# Junit-Installationspfad
junitInstall=D:/java/junit4.3.1

# Checkstyle-Installationspfad
checkstyleInstall=D:/java/checkstyle-4.3
```

20. XSL steht für *eXtensible Stylesheet Language*. Über XSL werden Transformationsregeln für die einzelnen Elemente einer XML-Datei definiert. Um beispielsweise aus einer XML-Datei eine HTML-Datei zu erzeugen, muss via XSL jedem XML-Element ein passender HTML-Tag zugewiesen werden.

Ausführung der Audits

Gestartet werden die Audits entweder vom Entwickler- oder vom Integrationsskript. Beginnen wir mit der Erweiterung des Entwicklerskriptes:

Listing 37
Aufruf von Checkstyle über ein neuen Build-Ziel im Entwicklerskript

```
<!-- ==== Checkstyle ausführen ==== -->
<target name="checkstyle" depends="prepare"
        description="Führt Checkstyle aus">
  <antcall target="-checkstyle"/>
</target>
```

Das neue Build-Ziel `checkstyle` kann von jedem Entwickler bei Bedarf separat aufgerufen werden. Es hat, bis auf das obligatorische `prepare`, keine weiteren Abhängigkeiten und ist daher nicht in den »normalen« Build-Lauf eingebunden. Entwickler sollten die Audits jedoch zumindest vor jedem `commit`-Befehl starten, um sicherzustellen, dass durch das neue Changeset keine Fehler verursacht werden.

Vergisst ein Teammitglied diesen kurzen Check, kommen fehlerhafte Audits spätestens durch den Integrations-Build ans Tageslicht. Listing 38 zeigt die entsprechende Erweiterung des Skriptes `build-int.xml`. In diesem Fall gibt es kein separates Build-Ziel zur Ausführung von Checkstyle. Vielmehr werden die Audits, genau wie die Modultests, automatisch vor der Erstellung der Installationsdatei gestartet.

Listing 38
Erweiterung des Build-Ziels zur Erstellung der Installationsdatei im Integrationsskript

```
<!-- ==== Installationsdatei erstellen ==== -->
<target name="install" depends="clean, prepare"
        description="Erstellt die Installationsdatei">
  <!—Java-Compiler aufrufen -->
  <antcall target="-compile">
    <param name="compile.debug" value="true" />
    <param name="compile.debug.level" value="lines" />
  </antcall>

  <!-- Modultests ausführen -->
  <antcall target="-test">
    <param name="test.haltonfailure" value="true" />
  </antcall>

  <!-- Checkstyle ausführen -->
  <antcall target="-checkstyle"/>

  ... (siehe Listing 26)

</target>
```

Nachdem die Implementierung der beiden Skripte erweitert wurde, können wir die automatisierten Audits erstmalig ausführen. Dies erfolgt am einfachsten mit Hilfe des Entwicklerskriptes:

```
> ant checkstyle
Buildfile: build.xml

prepare:

-prepare:

checkstyle:

-checkstyle:
[checkstyle] Running Checkstyle 4.1 on 18 files

BUILD FAILED
Checkstyle-Audits sind fehlgeschlagen!
```

Das Ergebnis ist ernüchternd, da e2etrace die Audits nicht besteht. Die detaillierten Ursachen hierfür finden wir im Checkstyle-Bericht `target\checkstyle\checkstyle_report.html` (siehe Abb. 6 und 7). Vor der Auslieferung von e2etrace muss offensichtlich die JavaDoc-Dokumentation noch nachgebessert werden.

Einrichtung einer Projekt-Homepage

Unser Build-Prozess produziert eine Reihe von Ergebnissen, die sich sehr gut zur Veröffentlichung auf einer Projekt-Homepage eignen. Dies sind im Einzelnen:

- Die Ergebnisse der während des Integrations-Builds durchgeführten Modultests
- Der ebenfalls im Integrations-Build erstellte Checkstyle-Bericht
- Die während des Integrations-Builds generierte Logdatei

Abb. 6
Zusammenfassung des Checkstyle-Berichtes

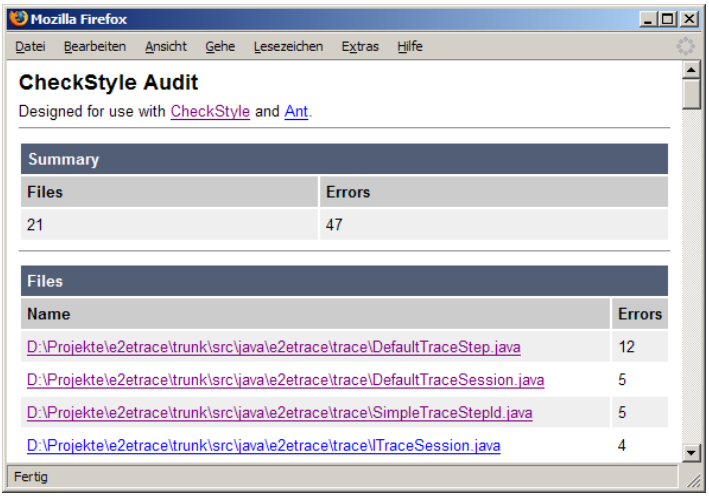
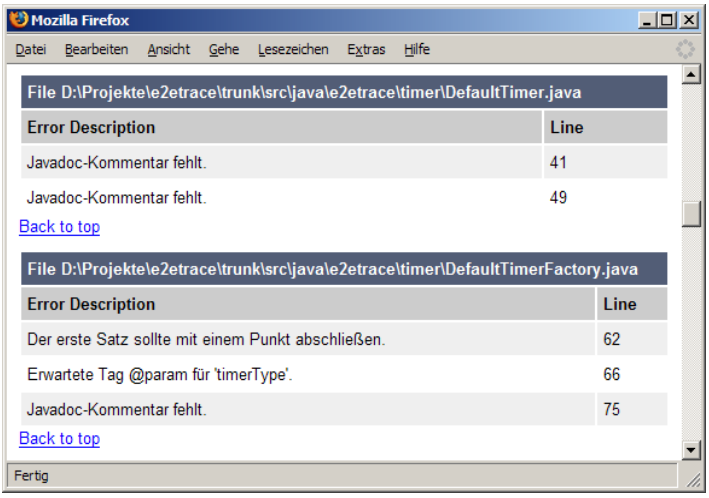


Abb. 7
Detaillierte Audit-Ergebnisse im Checkstyle-Bericht



Eigenes Homepage-Team

Große Projekte leisten sich oft den Luxus, für die Erstellung und Pflege einer Projekt-Homepage ein extra Team abzustellen. In diesem Fall muss man das Integrationsskript lediglich minimal erweitern und die Audit- und Modultest-Berichte regelmäßig in die entsprechenden Verzeichnisse auf dem Webserver kopieren.

Einsatz eines Publishing-Frameworks

Viele Projekte müssen allerdings ohne ein eigenes Homepage-Team auskommen. Trotzdem ist auch in diesen Fällen die Einrichtung einer Projekt-Homepage sinnvoll, nur muss man aus Aufwandsgrün-

den die Sache pragmatisch angehen. Ant bietet leider von Haus aus keine Unterstützung zur Erstellung einer Projekt-Homepage an. Statt sich selbst mit dem Schreiben von HTML-Seiten herumzuschlagen, empfiehlt sich der Einsatz spezieller *Publishing-Frameworks*. Derartige Werkzeuge reduzieren die notwendigen Arbeiten auf das Schreiben der eigentlichen Inhalte. Anhand vorgegebener Templates werden dann aus den Inhalten HTML-Seiten generiert. Das resultierende Design mag nicht in allen Details den eigenen Ansprüchen genügen, aber in der Regel erfüllt die Projekt-Homepage trotzdem ihren Zweck.

Apache Forrest

Ein relativ junges, aber meines Erachtens sehr gut gemachtes Publishing-Framework ist *Apache Forrest*[21]. Selbst ohne jegliche Vorkenntnisse kann man mit Hilfe von Forrest mit maximal einem Tag Aufwand den Rahmen für eine Projekt-Homepage erstellen. Hierbei legt man Menüstruktur und Inhalte in XML-Dateien fest, aus denen Forrest dann HTML-Dateien generiert. Diese werden durch die vom Build-Prozess erstellten Berichte ergänzt und auf einem beliebigen Webserver veröffentlicht. Für das in Abbildung 8 gezeigte Beispiel sind nur eine Hand voll XML-Dateien mit jeweils 40–50 Zeilen Quelltext notwendig. Wer tiefer in Forrest einsteigen möchte, findet auf der Webseite zum Buch den kompletten Quelltext für das in der Abbildung gezeigte Beispiel.

21. Es heißt wirklich *Forrest*, nicht *Forest* (Wald). Apache Forrest war ursprünglich als Schwesterprojekt von Apache Gump geplant, daher der Name. Näheres zum Projekt erfährt man unter *http://forrest.apache.org*. Es lag im Frühjahr 2006 erst in Version 0.7 vor, war aber trotzdem schon stabil und vor allem sehr gut dokumentiert.

Abb. 8
Mit Apache Forrest generierte Projekt-Homepage