mehr zum thema:
www.andrena.de/netweaver

TESTGETRIEBENE ENTWICKLUNG MIT SAP NETWEAVER™

Moderne Geschäftsanwendungen sind einem ständigen Wandel an Funktionalität unterworfen. SAP® erklärt die Adaptivität von Prozessen, Anwendungen und Services in der heutigen Zeit zum Schlüsselfaktor für die Zukunft eines Unternehmens. Die Veränderbarkeit von Software erfordert sowohl Änderungsfreundlichkeit von Architektur und Vorgehen als auch schnelle Änderungszyklen, wie es agile Methoden ermöglichen. Ein wichtiger Bestandteil agilen Vorgehens ist die testgetriebene Entwicklung, die uns kontrollierte und abgesicherte Änderungen erlaubt. In diesem Artikel wird die testgetriebene Entwicklung mit SAP NetWeaver™ vorgestellt.

„SAP NetWeaver macht Ihre IT-Landschaft flexibel und agil!", steht auf der deutschen Startseite des Walldorfer Softwarehauses (vgl. [SAP-b]) zu SAP NetWeaver™. Der Erfolg von agilen Entwicklungsmethoden wie eXtreme Programming (vgl. [XP]) und Scrum (vgl. [Scrum]) ist entscheidend dafür verantwortlich, dass das Wort *agil* auch zum Schlagwort des Marketings wurde. Agile Methoden stehen dabei für zielgerichtete und flexible Anpassungen an die sich ständig ändernden Situationen in der IT-Landschaft.

Die Zeiten von „Never Change A Running System" sind laut SAP® nun endgültig vorbei. Wichtig bei Änderungen an der Funktionalität ist, dass sie kontrolliert erfolgen und Seiteneffekte vermieden werden. Hierfür eignen sich vor allem Tests, die wiederholt und automatisiert ausgeführt werden können. Die Tests selbst werden quasi zur Dokumentation und haben gegenüber Pflichtenheften und Anforderungsdokumenten einen entscheidenden Vorteil: Sie sind immer aktuell und spiegeln die tatsächlich implementierten Anforderungen wider.

Tests werden auf unterschiedlichen Granularitätsebenen ausgeführt:

- *Unit-Tests* validieren die Funktionalität einzelner Komponenten und Klassen.
- *Integrationstests* prüfen das Zusammenspiel mehrer Komponenten.
- Bei *Systemtests* handelt es sich um Tests des Systems gegen seine Anforderungen.

*) SAP®, SAP NetWeaver™ sind eingetragene Markenzeichen der SAP AG.

In diesem Artikel konzentrieren wir uns auf die testgetriebene Entwicklung (*Test Driven Development, TDD)* mit Unit-Tests und der Einbindung in den SAP-Entwicklungsprozess.

Testgetriebene Entwicklung

Der Kern der testgetriebenen Entwicklung (vgl. [Lin05]) besteht aus einem einfachen, aber folgenreichen Miniprozess (siehe auch Abb. 1):

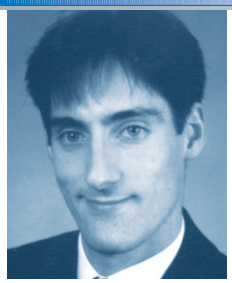
1. Zerlege die nächste umzusetzende Programmieraufgabe in kleine funktionale Happen.
2. Wähle einen Happen und schreibe dafür einen automatisierten Testfall (z.B. mit JUnit, vgl. [JUn]), der noch nicht funktioniert.
3. Schreibe gerade so viel Applikationscode, um den neuen Test (und alle bereits existierenden Testfälle) zum Laufen zu bringen.
4. Räume den Code auf und führe sinnvolle Refaktorisierungen (vgl. [Fow00]) durch.
5. Wiederhole die Schritte 2 bis 4 so lange, bis es keine für die gegenwärtige Anforderung sinnvollen Testfälle mehr gibt.

Die dabei durchgeführten „Mikro-Iterationen" (Schritte 2 bis 4) sollten nur wenige Minuten dauern; so kleine Schritte geben uns maximale Kontrolle über den Programmierfortschritt.

Die konsequente Umsetzung dieses Vorgehens hat zahlreiche positive Effekte. Zum einen erhält man automatisch Code mit einer hohen Testabdeckung. Diese Tests können als Regressionstests verwendet werden und so den Erhalt bestehender Funktionalität im Zuge der Weiterentwicklung sicherstellen; nur mit diesem Sicherheitsnetz lassen sich ständige Refaktorisierungen und Designevolution tatsächlich ohne all zu großes Risiko durchführen. Zum anderen beeinflusst die starke Ausrichtung auf Testbarkeit maßgeblich den Entwurf, da – quasi automatisch – stark entkoppelte Klassen entstehen: ein wesentliches Merkmal guten objektorientierten Designs. Im Gegensatz hierzu gestaltet sich nachträgliches Testen oft als schwierig, da sich sowohl der Code als auch unser innerer Schweinehund gegen diese scheinbar überflüssige Tätigkeit („Das Programm funktioniert doch schon!") wehrt.

Damit testgetriebene Entwicklung auch in einem größeren Team funktionieren kann, bedarf es jedoch eines weiteren Bausteins: der *Continuous Integration* (fortlaufende Integration) (vgl. [Fow]). Dieser Begriff impliziert, dass nach jeder beendeten Aufgabe der neue und veränderte Code ins Gesamtsystem integriert wird und dabei sowohl die formale Kompatibilität (lässt sich der gesamte Quellcode miteinander kompilieren?) als auch die

die autoren



Marco Klemm (E-Mail: marco.klemm@andrena.de) ist SAP NetWeaver Development Consultant und Projektleiter bei der andrena objects ag, wo er für den Bereich SAP verantwortlich ist.

Johannes Link (E-Mail: johannes.link@andrena.de) ist Softwareentwickler und Projektleiter bei der andrena objects ag. Er ist Autor der Bücher „Softwaretests mit JUnit" und „Unit Testing in Java".

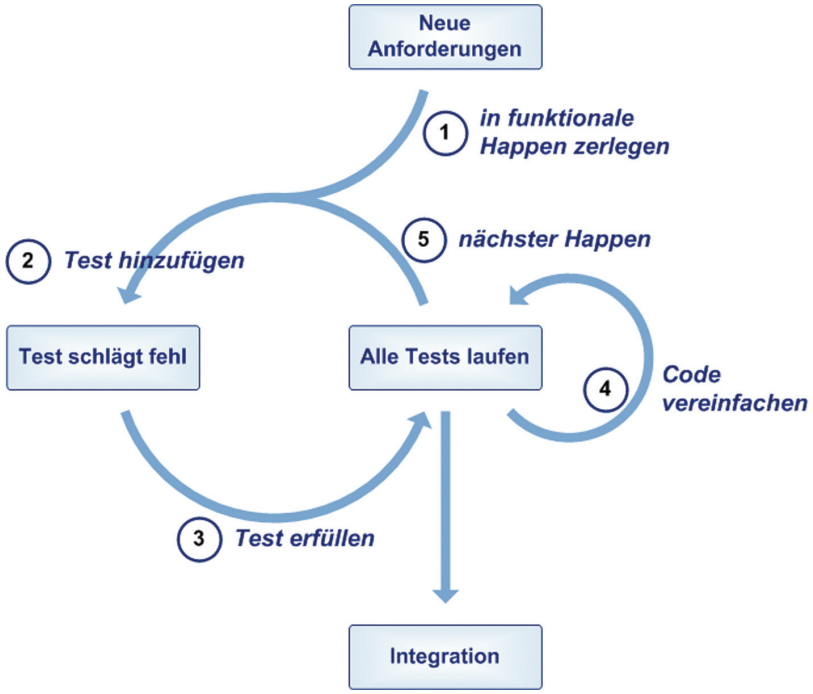


Abb. 1: Testgetriebener Miniprozess

funktionale Integrität (laufen alle automatisierten Tests erfolgreich?) sichergestellt werden. Typischerweise findet die Integration ins Gesamtsystem mehrmals – jedoch mindestens einmal – am Tag statt; je häufiger man integriert, desto kleiner gestalten sich die Integrationsprobleme. *Continuous Integration* kann sowohl durch ein standardisiertes und sequenzielles Vorgehen erfolgen als auch durch spezialisierte Werkzeuge (z. B. „CruiseControl", vgl. [Cru]) unterstützt werden.

Ein weiterer wesentlicher Punkt bei der testgetriebenen Entwicklung betrifft die Granularität unserer Testfälle. Es handelt sich hierbei ausschließlich um echte Unit-Tests, d. h. Testfälle, die unabhängig von externen Komponenten und Ressourcen (z.B. Dateisystem und Datenbank) sind. Dies ist notwendig, um die automatisierten Tests sowohl wiederholbar und schnell zu machen – schließlich werden sie alle paar Minuten ausgeführt – und um ihre Aussagekraft möglichst spezifisch zu halten (wo ist ein Fehler versteckt, wenn ein bestimmter Test fehlschlägt?). Im SAP NetWeaver™-Kontext bedeutet dies beispielsweise, dass solche Unit-Tests nicht über die Grenzen der zu testenden Komponente hinausgehen und auch nicht auf die sie umgebende SAP-Systemlandschaft zugreifen. *Dummy*- und *Mock*-Objekte sind ein geeignetes Mittel, eine Unit für die Dauer eines Tests von diesen Abhängigkeiten zu isolieren.

Unit-Tests in SAP NetWeaver™

Wie ist es nun möglich, testgetrieben, iterativ und vor allem einfach mit dem SAP NetWeaver™ Developer Studio (im Folgenden kurz „Developer Studio") zu entwickeln? Der TDD-Prozess ist werkzeugunabhängig und kann letztendlich mit jeder Programmiersprache durchgeführt werden. Das Developer Studio basiert auf der Plattform Eclipse (vgl. [Ecl]). Was dort sehr gut funktioniert, wird auch hier funktionieren.

Zunächst ist es wichtig, eine geeignete Strukturierung für die Unit-Tests zu finden, die mit dem SAP-Komponentenmodell (vgl. [SAP-a]) zusammenpasst. Dieses besteht aus folgenden Elementen:

- Produkt
- Softwarekomponente (*Software Component, SC*)
- Entwicklungskomponente (*Development Component, DC*)
- Entwicklungsobjekt (*Development Object, DO*)

Produkte sind eine Auswahl von Softwarekomponenten. Dabei können SCs in mehreren Produkten enthalten sein. Ein Produkt beschreibt dabei typischerweise die Menge der Funktionen, die dem Kunden zur Verfügung gestellt wird.

Softwarekomponenten definieren eine modular abgeschlossene Funktionalität und sind die Auslieferungseinheit für Installationen und Upgrades.

Bestandteil der SCs sind die *Entwicklungskomponenten*, die die elementaren Einheiten für einen *Build* und das *Deployment* darstellen. Jede DC hat einen eindeutigen Typ, der den Verwendungszweck widerspiegelt (z.B. WebDynpro-, J2EE-, Enterprise-Portal-, Java-Komponente). Für unsere Betrachtungen ist dies insofern wichtig, als dass der hier beschriebene testgetriebene Entwicklungsprozess für alle gängigen Komponententypen funktioniert.

Das SAP-Komponentenmodell kennt Sichtbarkeitsregeln für DCs, die durch Komponentenhierarchien (Eltern/Kind-Beziehung) und die Relation *Usage-Dependency* definiert sind. So sind nach Voreinstellung alle DOs und geschachtelte DCs außerhalb ihrer umschließenden DC nicht sichtbar, es sei denn, sie werden in einem *Public Part* als Komponentenschnittstelle explizit freigegeben.

Die DCs umfassen eine Menge von *Entwicklungsobjekten*, die als einzelne Dateien in der Versionsverwaltung enthalten sind. Beispiele für DOs sind Java-Quelldateien, Konfigurationsdateien und Web-Ressourcen.

Unit-Tests sind dafür gedacht, die elementaren Einheiten isoliert zu testen – dies entspricht unseren DOs, d. h. einzelnen Klassen oder wenigen, sehr eng zusammen arbeitenden Klassen.

Unit-Test-Strukturierung

Es stellt sich die Frage, wie man Unit-Tests in dieses Komponentenmodell geeignet integriert. Für die Strukturierung von Applikations- und Testcode gibt es mehrere Möglichkeiten, die wir zunächst aufzählen und anschließend für die SAP-Entwicklung bewerten:

- Die Unit-Tests werden direkt bei der zu testenden Klasse im gleichen Java-*Package* abgelegt.
- Die Unit-Tests sind im gleichen Java-*Package*, jedoch in einem separaten Quellcode-Verzeichnis der gleichen Entwicklungskomponente organisiert.
- Die Unit-Tests werden modular in einer eigenen Entwicklungskomponente separiert.

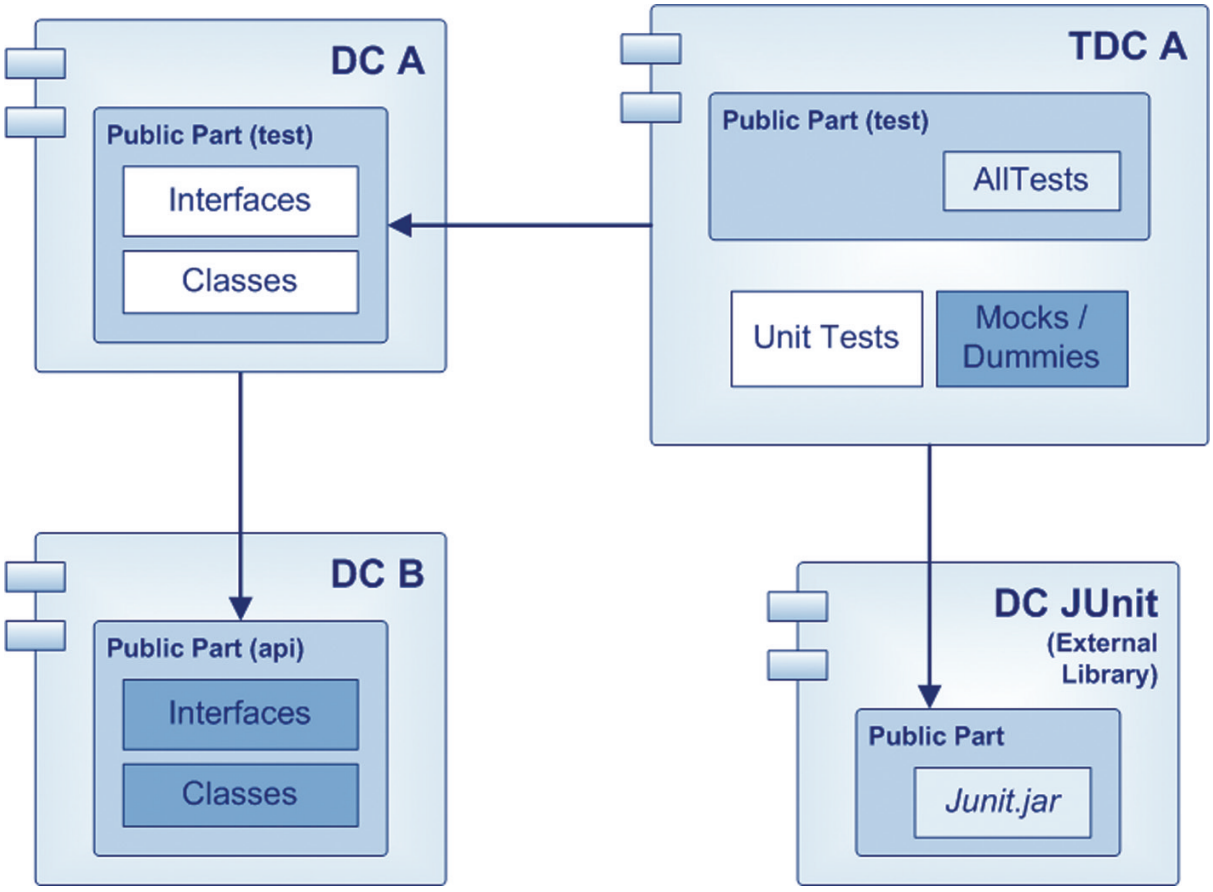


Abb. 2: Teststrukturierung einer Development-Component (DC A) mit Abhängigkeiten (DC B)

Die erste Variante hat den Vorteil, dass die Testklassen nahe bei den zu testenden Klassen liegen, was dem Entwickler die tägliche Arbeit erleichtert. Im zweiten Fall erreicht man eine stärkere Trennung von Test- und Applikationscode. Allerdings ist bei beiden Varianten nicht sichergestellt, dass unerwünschte Abhängigkeiten vom Applikationscode zu den Testklassen entstehen, da in Java jeder als public deklarierte Typ von jedem anderen Objekt verwendet werden kann. Dies kann verhindert werden, wenn – wie in der dritten Variante – die Unit-Tests samt Seilschaft (z.B. *Mocks*, *Dummies*) in einer eigenen Entwicklungskomponente gekapselt werden.

Test-Development-Components

Den dritten Fall betrachten wir nun genauer. Wir bezeichnen im Folgenden eine DC, die die Unit-Tests und ihre Abhängigkeiten kapselt, als *TDC* (Testentwicklungskomponente, *Test-Development-Component*). Eine DC und ihre dazugehörige TDC müssen folgenden Anforderungen genügen:

a) Die TDC muss vollen Zugriff auf alle DOs einer DC haben, um alle funktionalen Anforderungen testen zu können.
b) Keine DC darf Zugriff auf TDCs haben, um die Separierung von Test- und Applikationscode zu sichern.
c) Alle Abhängigkeiten einer DC zu anderen DCs müssen von ihrer TDC isoliert werden, um die lokale Ausführung der Tests zu gewährleisten.

Die ersten beiden Anforderungen werden umgesetzt, indem wir für die DC einen *Public Part* mit allen zu testenden Klassen/Schnittstellen und eine *Usage-Dependency* von der TDC zu der DC definieren. Dies ist in Abbildung 2 dargestellt. Damit hat TDC *A* Zugriff auf alle Klassen und Schnittstellen von DC *A*. Wenn Komponentenhierarchien über geschachtelte DCs gebildet werden, müssen wir zusätzliche Schritte ausführen, weil eine geschachtelte DC nicht außerhalb ihrer besitzenden DC sichtbar ist. Das Komponentenmodell stellt für dieses Problem ein Werkzeug bereit: mit einer *Public Part Entity Reference* (vgl. [SAP-a]) wird der *Public Part* der geschachtelten DC an die besitzende DC weitergereicht, womit das beschriebene Verfahren greift.

Im Beispiel hat die Entwicklungskomponente *A* eine *Usage-Dependency* zur Komponente *B* und damit Zugriff auf ihren *Public Part*. Um Anforderung c) zu erfüllen, müssen die Schnittstellen und Klassen der Entwicklungskomponente *B* durch *Mocks* oder *Dummies* isoliert werden. Diese können ebenfalls Bestandteil der TDC sein, womit die Separierung von Applikations- und Testcode weiterhin erfüllt ist. Das Schreiben von *Mocks* ist mit den verfügbaren Java-Mock-Frameworks (z. B. „EasyMock", vgl. [Eas]) sehr einfach. Möchte man *Mocks* wieder verwenden, bietet es sich an, generische TDCs zu extrahieren.

JUnit-Einbindung

Damit die Unit-Tests tatsächlich ausgeführt werden können, fehlt noch ein wichtiger Bestandteil – das Test-Framework „JUnit". Um JUnit nicht in jede TDC einbinden zu müssen, separieren wir das Test-Framework in einer eigenen DC. Seit SAP NetWeaver™ Developer Studio SP11 ist es möglich, eine DC vom Typ External Library zu erstellen. Damit ist uns der Großteil der Arbeit schon abgenommen: wir fügen das Archiv JUnit.jar der External Library-Komponente hinzu und

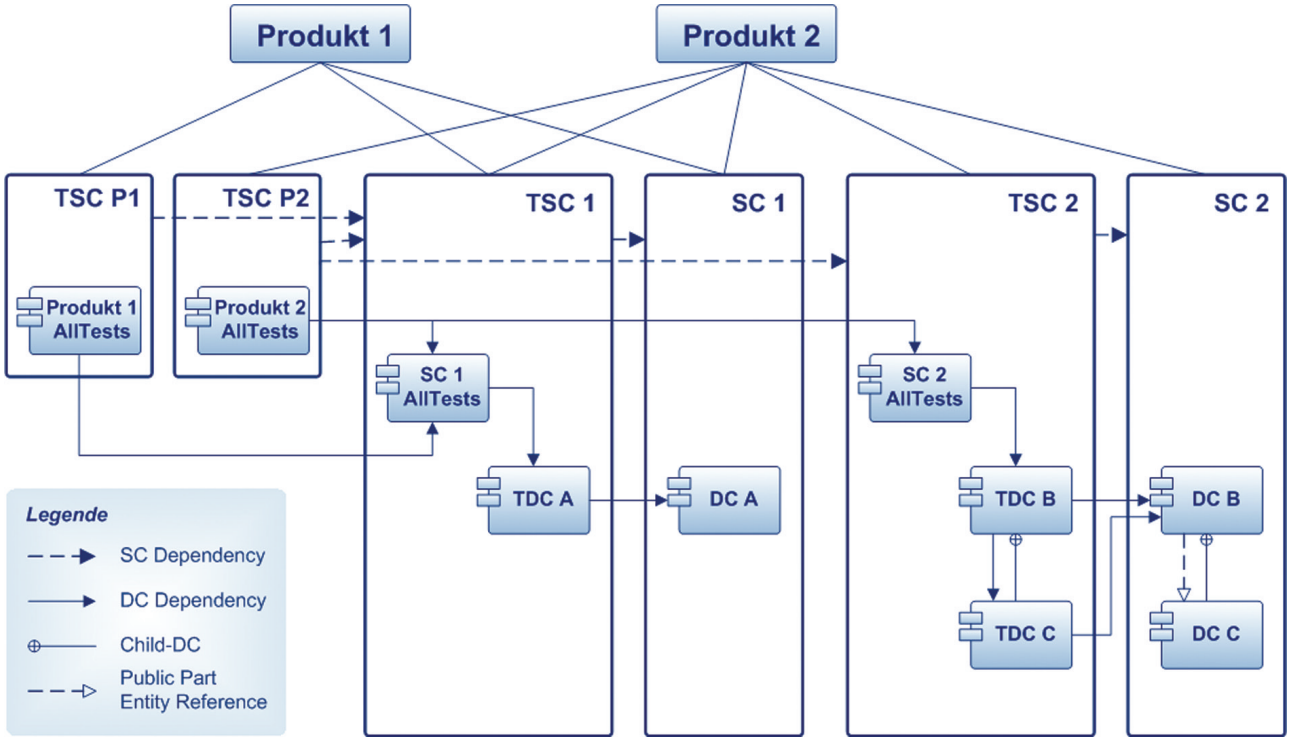


Abb. 3: Komponentenarchitektur mit der dazugehörigen Testhierarchie

definieren einen *Public Part*, um das JUnit-Framework für die TDCs sichtbar zu machen (siehe Abb. 2).

Hierarchische Testsuiten

Unsere bisherigen Betrachtungen befassten sich mit der Teststrukturierung für eine DC und ihre Abhängigkeiten. In der testgetriebenen Java-Entwicklung hat es sich bewährt, Testsuiten zu organisieren, die die logische Modularisierung widerspiegeln. Auf diese Weise ist es möglich, Tests auf jeder Aggregationsstufe (einzelne Klasse, *Package*, Komponente, Produkt) auszuführen. Gleiches wollen wir auch hier unter Berücksichtigung des SAP-Komponentenmodells erreichen. Folgende Anforderungen sollen zusätzlich zu den oben bereits genannten erfüllt sein:

d) Alle Tests einer DC müssen gemeinsam ausgeführt werden können.
e) Alle Tests aller DCs einer SC müssen gemeinsam ausgeführt werden können.
f) Alle Tests aller zu einem Produkt gehörenden SCs müssen gemeinsam ausgeführt werden können.

Um alle Tests einer DC auszuführen, reicht es, eine Testsuite-Klasse zu erstellen, die alle Unit-Tests der DC aufruft – nennen wir diese *DC-AllTests*. Gewöhnlich berücksichtigt man dabei die Java-*Package*-Struktur, d. h. in jedem Java-*Package* mit Unit-Tests wird eine eigene Testsuite erzeugt, die die dortigen Unit-Tests zusammenfasst. Die Testsuite der DC sammelt die Testsuiten der Java-*Packages* lediglich ein.

Für die Ausführung aller Tests einer Softwarekomponente erzeugen wir eine eigene TDC mit einer Testsuite – wir nennen sie *SC-AllTests* –, deren einzige Aufgabe es ist, die Testsuiten aller anderen TDCs dieser SC aufzurufen.

Als Letztes benötigen wir noch eine Test-Softwarekomponente (*Test Software Component, TSC*), deren einziger Zweck es ist, die Unit-Tests aller SCs aufzurufen. Da Softwarekomponenten Bestandteil mehrerer Produkte sein können, benötigen wir genau eine TSC für jedes Produkt, um einerseits alle Tests eines Produkts ausführen zu können und andererseits die Abhängigkeiten des Produkts zu ihren SCs einzuhalten. Damit sind die Anforderungen d) bis f) erfüllt, was in der Beispiel-Komponentenarchitektur mit der dazugehörigen Testhierarchie in Abbildung 3 dargestellt ist. Die Testkomponenten sind in eigenen TSCs separiert, was die prinzipielle Auslieferung des Produkts ohne Testcode ermöglicht (wenn man dies denn möchte).

Testgetriebener SAP-Entwicklungsprozess

Kommen wir nun zur eigentlichen Umsetzung. Die Aufgabe besteht darin, das oben beschriebene testgetriebene Vorgehen möglichst nahtlos so in den SAP-Entwicklungsprozess zu integrieren, dass die gewünschten Testkomponenten und ihre Abhängigkeiten bei Bedarf entstehen und nicht vorab definiert werden müssen. Dabei sind die Schritte

- Importieren/Aktualisieren der notwendigen DCs,
- Hinzufügen/Ändern der DCs und
- Freigeben der aktualisierten DCs

wiederholt, iterativ auszuführen. Mit anderen Worten: Testgetriebene Entwicklung und *Continuous Integration* werden hier auf den SAP-Entwicklungsprozess übertragen.

Das SAP-Komponentenmodell bildet die Grundlage für den automatisierten *Build*-Prozess, das *Deployment* und die Freigabe von Releases (vgl. [Kes05]). Für den Entwicklungsprozess sind außer Developer Studio drei weitere Komponenten wichtig, die Bestandteil der „Java Development Infrastructure" (JDI)[1]) sind:

[1]) *neuerdings von SAP als NetWeaver Development Infrastructure (NWDI) bezeichnet.*

- *Design Time Repository (DTR)*,
- *Component Build Service (CBS)* und
- *Software Landscape Directory (SLD)*.

Das DTR ist die Versionsverwaltung von SAP NetWeaver™ und bietet in etwa die Funktionalität, die wir von CVS und Subversion kennen. Sie ermöglicht sowohl die Versionierung auf Dateiebene, das Ein- und Auschecken und Konfliktlösung, als auch die Verwaltung mehrerer Entwicklungszweige.

Der CBS sorgt für die *Builds*, die sowohl lokal als auch zentral ausgeführt werden können, und berücksichtigt dabei die Abhängigkeiten des Komponentenmodells; Verletzungen der Abhängigkeiten führen zum Abbruch des *Build*-Vorgangs.

Im SLD sind die Entwicklungskonfigurationen (*Development Configurations*) gespeichert, die unter anderem Informationen über die Abhängigkeiten von Softwarekomponenten enthalten. Ein *Track* fasst alle Entwicklungskonfigurationen von einer oder mehreren SCs zusammen.

In Abbildung 4 ist der testgetriebene Entwicklungsprozess dargestellt. Zunächst importieren wir eine Entwicklungskonfiguration aus dem SLD, um anschließend die darin konfigurierten SCs und DCs auszuchecken (Schritt 1). Die Komponenten befinden sich im inactive state, was bedeutet, dass eine lokale Kopie zur Bearbeitung freigegeben ist. Dieser Status wird durch einen eigenen *Workspace* im DTR repräsentiert. Das Developer Studio importiert automatisch alle benötigten Bibliotheken, um die Komponenten lokal kompilieren zu können. Auf diese Weise hat der Entwickler eine lokale Entwicklungsumgebung, in der sich Änderungen nicht auf die zentrale Infrastruktur auswirken.

Nun kann der testgetriebene Entwicklungszyklus stattfinden (Schritte 2 bis 6 **in Abb. 4**). Der Unterschied der Entwicklung für SAP NetWeaver™ im Vergleich zur konventionellen Java-Entwicklung besteht darin, dass der *Build* über den CBS integraler Bestandteil des Entwicklungsprozesses ist. Die Komponentenhierarchie und *Usage-Dependency* definieren die Sichtbarkeit von Komponenten. Der lokale *Build* sorgt per Knopfdruck dafür, dass die im Komponentenmodell definierten Abhängigkeiten eingehalten werden, indem er den *Java Build Path* aller Entwicklungskomponenten aktualisiert. (Für Eclipse-

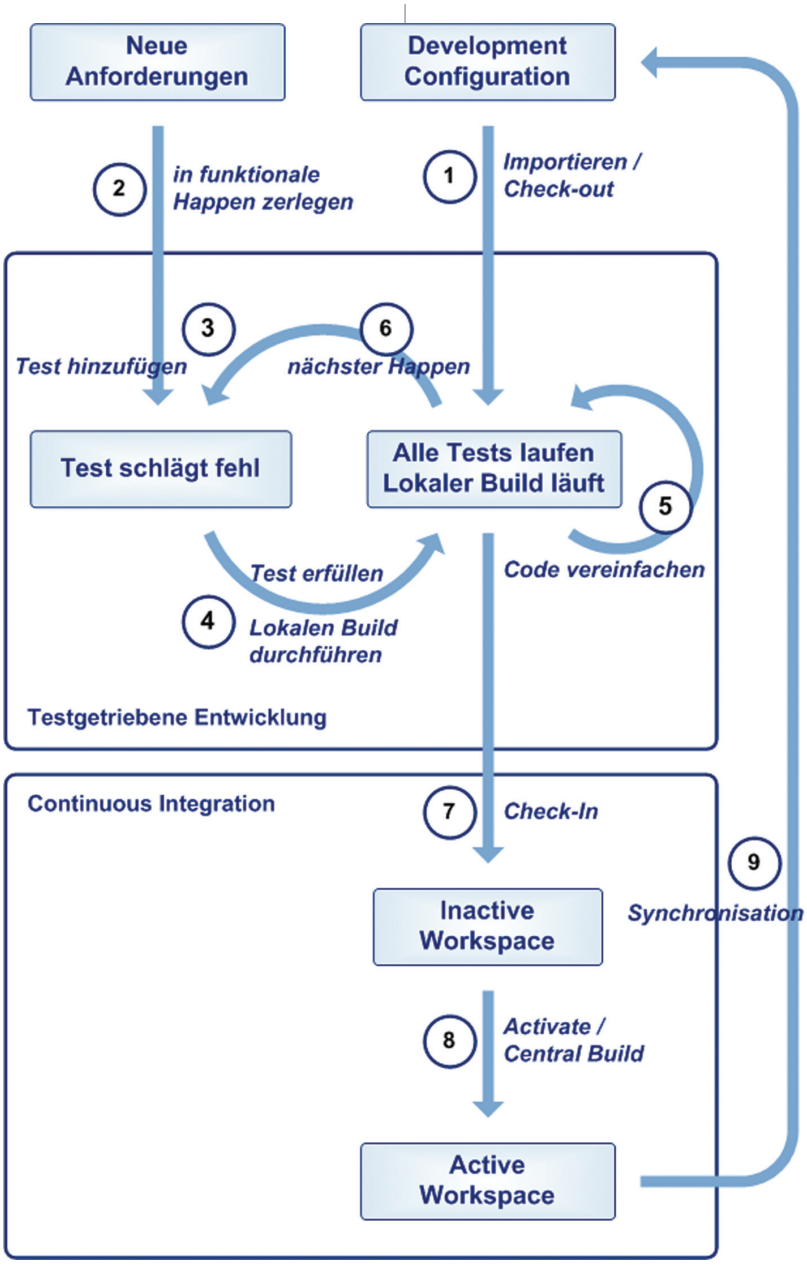


Abb. 4: Einbindung des testgetriebenen Miniprozesses in den JDI-Entwicklungsprozess

gewöhnte Entwickler ist dies ein beschwerliches Unterfangen – erwarten diese doch, dass dies direkt bei der Änderung von Abhängigkeiten zwischen DCs geschieht.) Ist das nicht der Fall, bricht der *Build* ab und das Developer Studio wird uns die entsprechenden Fehlermeldungen des Compilers anzeigen (in diesem Fall sollten auch die Unit-Tests fehlschlagen).

Nachdem eine Aufgabe abgeschlossen ist und alle Unit-Tests in der lokalen Entwicklungsumgebung durchgelaufen sind, werden die geänderten Quellen zurück in den inaktiven *Workspace* des DTR eingecheckt (Schritt 7). Im nächsten Schritt aktiviert man die bereits versionierten Änderungen und macht diese der Systemlandschaft bekannt. Hierzu wird der zentrale *Build* des CBS ausgeführt, der sich vom lokalen *Build* nicht unterscheidet. Es wird geprüft, ob die geänderten DCs mit den auf dem Server liegenden Komponentenversionen kompatibel sind. Im Erfolgsfall werden die geänderten DCs vom inaktiven zum aktiven DTR-Workspace übernommen. Als letztes (Schritt 9) synchronisiert man die neue Entwicklungskonfiguration in den lokalen *Workspace*.

Fazit

Testgetriebene Entwicklung hat sich in den letzten Jahren als qualitätserhöhende Maßnahme bewährt. Sie beeinflusst das

Design positiv und führt zu lose gekoppelten Komponenten. Die dabei entstehenden automatisierten Regressionstests mit hoher Codeabdeckung sichern die Funktionalität bei Änderungen. Integrations- und Systemtests können die hier beschriebenen Unit-Tests sehr gut ergänzen.

Das normale TDD-Vorgehen bedarf weniger zusätzlicher Schritte, um nahtlos in den SAP-Entwicklungsprozess und das SAP-Komponentenmodell integriert werden zu können. Durch die Verwendung von Test-Entwicklungskomponenten und Test-Softwarekomponenten erreichen wir eine gute Separierung von Applikations- und Testcode. Der beschriebene Ansatz eignet sich insbesondere für die Entwicklung von Web-Services, WebDynpro- und Enterprise-Portal-Komponenten.

Testgetriebene Entwicklung ist im SAP-Umfeld eher unbekannt, was die ganzheitliche Einführung in die Softwareerstellung erschwert. Die Erfahrung zeigt jedoch, dass dies durch Schulung und Coaching spürbar erleichtert wird. Auch TDD will gelernt sein. ■

Literatur & Links

[Cru] CruiseControl, Homepage, siehe: cruisecontrol.sf.net/

[Eas] EasyMock, Homepage, siehe: www.easymock.org/

[Ecl] eclipse.org, Homepage, siehe: www.eclipse.org/

[Fow] M. Fowler, M. Foemmel, Continuous Integration, siehe: www.martinfowler.com/articles/continuousIntegration.html

[Fow00] M. Fowler, Refactoring – Improving the Design of Existing Code, Addison-Wesley, 2000

[JUn] JUnit, Testing Resources for Extreme Programming, siehe: www.junit.org/

[Kes05] K. Kessler et al., Java-Programmierung mit dem SAP Web Application Server, Galileo Press, 2005

[Lin05] J. Link, Softwaretests mit JUnit, dpunkt-Verlag, 2005.

[SAP-a] SAP Help Portal, siehe: help.sap.com

[SAP-b] SAP®, SAP NetWeaver™, siehe: www.sap.com/germany/solutions/netweaver

[Scrum] Control Chaos, siehe www.control-chaos.com/

[XP] Extreme Programming: A gentle introduction, siehe: www.extremeprogramming.org/