



Agiler Werkzeugkasten

Eine technologische Infrastruktur für agile Projekte

Michael Hüttermann

Zwecks zielgerichteter Umsetzung von Aufgaben ist eine technische Infrastruktur notwendig, ein Werkzeugkasten, der effektiv und effizient orchestriert werden muss. Dieser Artikel startet mit einer kontext-sensitiven Einordnung agiler Entwicklung, zeigt auf, welche Bereiche werkzeug-technisch abgedeckt werden sollten, und gibt einen Überblick über eine mögliche agile Infrastruktur für Ihre Projekte.

Das agile Manifest [Man] verdeutlicht explizit die Wichtigkeit von Tooling in einem agilen Projekt. Auch wenn Sie nicht umfassend auf eine „agile Methodik“ wie Extreme Programming oder auf das Management-Rahmenwerk Scrum setzen, können einzelne „agile Patterns“ aus dem agilen Werkzeugkasten jederzeit aufgenommen und angewendet werden. Dies gilt sogar für das als Paradebeispiel eines schwergewichtigen, starren Entwicklungsmodells geltende Wasserfallmodell [Rov70]. Aus dessen vermeintlicher Definition sind bereits agile Grundzüge herauszulesen, nämlich iterative, inkrementelle Entwicklung und die notwendige Nähe zum Kunden.

Infrastruktur

Wie kann eine agile Infrastruktur aussehen? Die hier vorgestellten Werkzeuge und Methodikansätze sollen Impulse und einen Bausteinkasten liefern. Die Werkzeuge zeichnen sich dadurch aus, dass sie in ihrer Domäne einen De-facto-Standard darstellen oder auf dem Weg sind, zu einem solchen zu werden. Deren Nutzung erlaubt agiles Vorgehen, minimiert Einarbeitungszeiten und vereinfacht Problemlösungen und entspannt das Projektbudget, sodass es noch mehr Spielraum hat zur Durchführung von Teambuilding-Maßnahmen.

Software-Configuration-Management

Software-Configuration-Management (SCM) ist eine optimierte Synthese der Disziplinen Release-, Build-, Deployment- und Konfigurationsmanagement hin zu einer Integrationsinfrastruktur. Das Release-Management bezieht sich nicht nur auf das fachliche Release-Management, wozu insbesondere die Identifikation der Konfigurationselemente im SCM-Container und eine Erstellung und Fortschreibung eines Release-Kalenders gehören, es bezieht sich darüber hinaus auf die Auf- und Umsetzung der den Kernprozess begleitenden Maßnahmen, wie beispielsweise das „Branching“ der Entwicklungslinien. Ferner muss das SCM die in diesem Artikel ausgeführten Aktivitäten respektive Werkzeuge definieren und bündelnd vereinen. Insbesondere in einer durch Komplexität und Heterogenität ausgezeichneten Systemlandschaft sollte eine technische Integrationsinfrastruktur aufgesetzt werden, die über den kompletten Lebenszyklus der Applikation von der ursprünglichen Vision der Anwendung bis zur Inbetriebnahme und Wartung alle klassischen Entwicklungsphasen begleitet.



Es werden also auch Artefakte der Anforderungserhebung (Use Cases, User Stories u.ä.) und Testfälle (technische und fachliche) in die atomare Produktklammer eingebunden, um reproduzierbar, automatisiert und effizient fachlich und technisch konsistente Versionen des hergestellten Gesamtpakets bereitzustellen und den Ansprüchen einer Revisionsicherheit zu genügen. Diese beispielsweise nächtlich erstellten Versionen können durch Quality Gates geschleust und eine Staging-Leiter weiter hoch geschoben werden, ohne dass das eigentliche Build (durch Kompilierung, Paketierung) neu erstellt werden muss.

Hinzu kann ein zentrales Vorhalten von QS-gesicherten Komponenten in jeweils gültigen Versionen mit deren (transitiven) Abhängigkeiten in einem Projekt- oder Unternehmens-Repository Transparenz und Wiederverwendbarkeit fördern. Komplementär dazu hilft die Integrationsinfrastruktur eine IDE- und eine Buildsicht kongruent zu verknüpfen. Der Entwickler in seinem an seinem individuellen Entwicklerprofil austarierten, schlanken Arbeitsplatz sieht und arbeitet mit exakt den Komponenten, die ebenso im Buildsystem erstellt und eingebunden werden. Fremdkomponenten werden beispielsweise elegant und entkoppelt als Abhängigkeit sauber eingebunden.

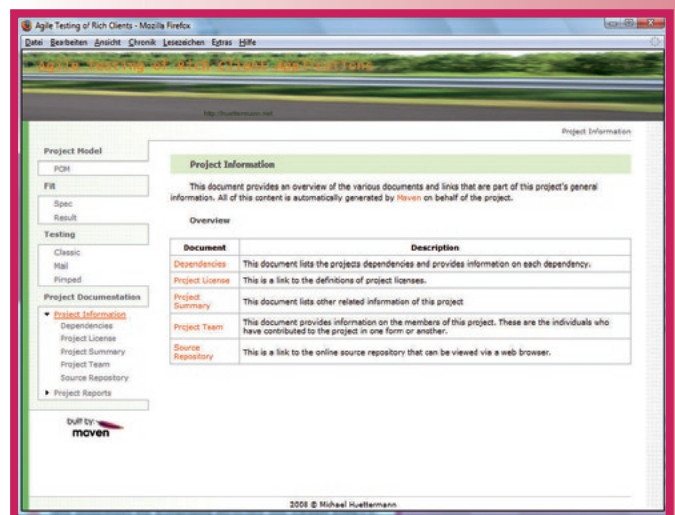


Abb. 1: Die Maven-Seite



SCHWERPUNKTTHEMA

Integrationsinfrastruktur: Maven

Obwohl beispielsweise auch mit Ant [Ant] möglich, hat sich beim Aufbau und Betrieb einer Integrationsinfrastruktur Maven [Maven] etabliert, da es eine Vielzahl von Anforderungen out-of-the-box bedient. Auch eine Symbiose aus Ant und Maven ist nicht unüblich und sogar bei einzelnen Aktivitäten anzuraten.

Maven kann die Subsysteme und Komponenten kompilieren, aggregieren und paketieren oder Zutaten ausschließlich abholen oder referenzieren. Die Basiskonfiguration ist dabei in einem ausführbaren Projektobjekt-Modell verankert, dem Maven-POM. Diese deklarativen Beschreibungen geben vor, „was“ gemacht werden soll (und nicht „wie“, im Vergleich beispielsweise zu Ant). Maven hilft, lange, manuell zu bearbeitende Checklisten respektive manuelle Arbeitsschritte zu minimieren. Auch die Konfiguration (durch Maven-Profile), das Deployment (durch automatische Verteilung) und das eigentliche Releasing (durch das Release-Plug-In) werden unterstützt. Der an Plug-Ins orientierte Maven-Charakter erlaubt es, Integrationsbuilds gezielt durch Einhängen weiterer Funktionalität zu erweitern respektive anzupassen. Funktionalität und Bestandteile (wie Testreihen, Audits) lassen sich mit der generierten Maven-Site umfassend dokumentieren. Abbildung 1 zeigt eine Maven-Site, die über ein eigens entwickeltes „Look and Feel“ verfügt und selbst als Maven-Plug-In bereitgestellt und eingebunden wurde.

Entwicklungsumgebung

Eine Maven-Integration ist für jede gängige IDE verfügbar. Das Maven-Plug-In für Eclipse [m2eclipse] bietet einen formularbasierten POM-Editor. Besonders interessant dabei ist die Darstellung der Abhängigkeiten im Abhängigkeitsgraphen (s. Abb. 2) sowie das Finden und Einbinden von Komponenten über einen Komponenten-Index. Es lassen sich so noch nicht im Klassenpfad existierende Komponenten finden, aus dem Internet oder einem Unternehmens-Proxy herunterladen bzw. im (Komponenten-)Repository bereitstellen. Dies geschieht transparent: Im Workspace werden die Abhängigkeiten über den „Maven Classpath Container“ entkoppelt eingebunden. Die Entkopplung unterstützt die Kongruenz zwischen den lokalen Entwicklersichten und einer zentralen Integrationsicht.

Versionskontrolle mit Subversion

Das Rückgrat Ihrer Entwicklung ist ein Versionskontrollsystem (VCS), das Ihre Artefakte zentral verwaltet, alte Stände verfügbar macht und zeitgleiches Arbeiten mehrerer Entwickler und Versionierung ermöglicht. In der Integrationsinfrastruktur werden die Skripte die wohldefinierten Versionen der Artefakte aus dem VCS holen und weiterverarbeiten. Subversion [SVN] hat sich als leichtgewichtige Wahl etabliert.

Der CVS-Nachfolger Subversion verwaltet auch Binärdateien und Meta-Daten sehr effizient und erlaubt kostengünstiges Tagging/Branching durch Kopieren von Verzeichnissen im Repository. Subversion erlaubt atomare Commits, lässt sich via Apache-Webserver betreiben und skalieren und basiert auf Konventionen, was dem „agilen Gedanken“ Rechnung trägt. Dass auch die Bearbeitungshistorie beim Verschieben erhalten bleibt, ermöglicht erst eine vollständige Revisionsicherheit (im Gegensatz zu CVS). Subversion kann sowohl in alle gängigen IDEs als auch automatisiert eingebunden werden. TortoiseSVN [TorSVN] erlaubt den komfortablen Zugriff aus dem herkömmlichen Explorer heraus.

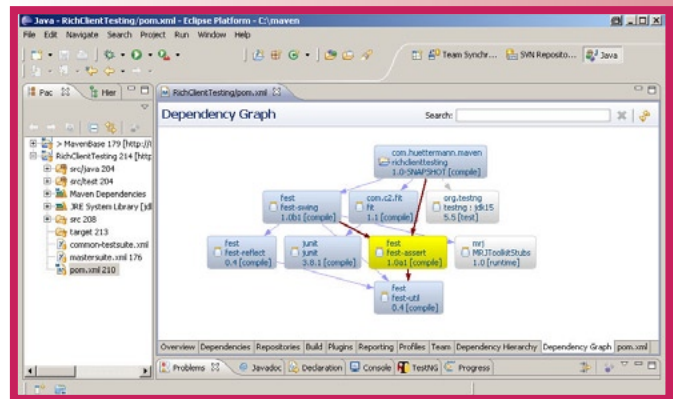


Abb. 2: Maven innerhalb von Eclipse

Akzeptanz- und Komponententests

Eine weitere wichtige Facette in Ihrem technologischen Ökosystem ist das Testen. Testen ist Software Engineering und Test-Artefakte werden verzahnt mit den eigentlichen Klassen entwickelt, fortgeschrieben und optimiert.

Funktionale Tests spezifizieren das Verhalten der Anwendung aus Nutzersicht. Diese Akzeptanztests beschreiben die Abnahmekriterien, wann also das entwickelte Feature als angenommen charakterisiert werden kann. Es lassen sich Anforderungen an Ihre Anwendung mit Fit [Fit] in einer für den Kunden verständlichen HTML-Notation formulieren, via Winword (oder Texteditor) fortschreiben und mit FEST [FEST] in Form eines intuitiv lesbaren „Fluent Interface“ (analog dem Ansatz der domänenspezifischen Sprachen, DSL) beispielsweise auf Ihre Java Swing-Anwendung anwenden (s. Listing 1 und [Huet08a]). Auch ein Fit-Einsatz im Kontext von Tests auf Webanwendungen (Rich Internet Applications) in der Kombination mit Selenium [Sel] und/oder WebTest [WebTest] ist denkbar [Huet08b].

Vom Entwickler geschriebene Komponententests überprüfen Design und Funktionalität des Codes, ob also die richtigen Features auch richtig umgesetzt wurden. Hier ist TestNG [TestNG] (s. Listing 2) eine gute Wahl. Es erlaubt die deklarative Beschreibung von Testsequenzen, die Durchführung von nebenläufigen oder datengetriebenen Tests und eine Definition von Testgruppen und -abhängigkeiten.

Testabdeckung

Alle Tests beider Testkategorien (Komponententests, funktionale Tests) sollten automatisiert durchgeführt werden, um einen hohen Entwicklungsfluss zu ermöglichen und kontinuierlich Regressionstests fahren zu können. Die Testergebnisse

```
public void insertRow5Column2(int param) {
    window.table(TABLE_NAME).cell(row(4).
        column(1)).enterValue(String.valueOf(param));
}

public String checkRow1Column1() {
    return window.table(TABLE_NAME).cell(row(0).column(0)).value();
}
```

Listing 1: FEST: Steuerung und Validierung der visuellen Controls via Fluent Interface, verankert in einer Fit-Fixture



```

package com.huettermann.prio;

import org.testng.annotations.DataProvider;
import org.testng.annotations.Test;

/**
 * @author Michael Huettermann
 */
@Priority(1)
public class PriorityTest {

    @Priority(3)
    @Test(groups = {"backend"})
    public void PriorityPlus3() {
        System.out.println("3");
    }

    @DataProvider(name = "provide")
    public Object[][] createData() {
        return new Object[][] {
            { "A", new Integer(36) },
            { "B", new Integer(37) } };
    }

    @Test(groups={"backend"},dataProvider = "provide",
        threadPoolSize = 3, invocationCount = 9, timeOut = 10000)
    public void verifyData(String n1, Integer n2) {
        //sophisticated tests
        System.out.println("d");
    }
}

```

Listing 2: TestNG: Ein Testintegrationswerkzeug. Priorisierte, datengetriebene, nebenläufige Tests

```

public class TestThisThat_Integration {
    private IConnector con;

    @BeforeTest(groups = {"goodtest"})
    public void setUp() throws Exception {
        con = EasyMock.createMock(IConnector.class);
    }

    @Test(groups={"goodtest"})
    public void testGood(){
        final Map <String,String>myMap = new HashMap<String,String>();
        myMap.put(„KEY“, „EINS“);
        EasyMock.expect(con.getDC()).andReturn(myMap);
        EasyMock.replay(con);
        String value = new MyServiceImpl().getSomething();
        Assert.assertTrue(value.equals(„EINS“));
        EasyMock.verify(con);
    }

    @Test(groups={"badtest"})
    public void testBad(){
        String value = new ServiceImpl().getSomething();
        Assert.assertTrue(value.equals(„EINS“));
    }

    private class MyServiceImpl extends ServiceImpl {
        @Override
        protected IConnector getConnector() {
            return con;
        }
    }
}

```

Listing 3: Mocking eines Datenbankzugriffs mit EasyMock, verheiratet mit TestNG

werden dabei regelmäßig publiziert. Eine interessante Frage hierbei ist, welcher Code eigentlich tatsächlich läuft bzw. getestet wird? Was bringt es, wenn 100 % der Tests erfolgreich laufen, Sie aber nur 50 % der Anwendung testen?

Eine Untersuchung der Testabdeckung mit EMMA [EMMA] kann hier Aufschluss geben, welche Stellen Ihrer Anwendung nicht durchlaufen werden oder für welche keine Tests existieren. Insbesondere dann, wenn während der Testreihe eine Klasse gar nicht angefasst wurde, sollten Sie genauer hinschauen. Das auf Bytecode-Instrumentalisierung basierende EMMA glänzt durch die Untersuchung der sogenannten Block-Abdeckung und unterschiedliche Laufmodi („on-the-fly“, offline). Ein überzeugendes Reporting stellt alle Java-Anwendungspakete dar mit der Möglichkeit, bis auf Klassen und Methoden weiterzunavigieren. Jeweils werden die gewonnenen Mess-

zahlen der (konfigurierbaren) Abdeckungseinheiten dargestellt und kontext-abhängig aggregiert. Auf Klassenebene werden um Durchlaufinformationen angereicherte, farblich markierte Programmzeilen des untersuchten Artefakts im unteren Bereich der Ergebnisseite angehängt. Abbildung 3 zeigt einen exemplarischen EMMA-Report.

Versuchen Sie gar nicht erst 100 % Testabdeckung für Komponententests anzustreben. Der Nutzen rechtfertigt nicht den Aufwand, der gewöhnlich ab einer Abdeckung von ca. 80 % exponentiell wächst oder per se nicht nötig ist (wie das Testen von settern und gettern). Berücksichtigen Sie Fallstricke, dass leere Testmethoden (Testmethoden mit einem leeren Rumpf) zur Folge haben, dass diese „Tests“ als erfolgreich markiert werden. EMMA lässt sich elegant in die IDE (vgl. [Well08] in dieser Ausgabe) und Ant-Skripte integrieren und in Maven-POMs reinhängen.

EMMA Coverage Report (generated Sat Sep 20 21:23:08 CEST 2008)				
[all classes]				
OVERALL COVERAGE SUMMARY				
name	class, %	method, %	block, %	line, %
all classes	80% (7/8)	64% (16/25)	70% (425/547)	80% (82.5/103)
OVERALL STATS SUMMARY				
total packages:	3			
total executable files:	5			
total classes:	8			
total methods:	25			
total executable lines:	103			
COVERAGE BREAKDOWN BY PACKAGE				
name	class, %	method, %	block, %	line, %
com.huettermann.fit.util	67% (2/3)	50% (6/12)	60% (113/189)	66% (24.5/37)
com.huettermann.fit.framework	100% (1/1)	100% (2/2)	81% (136/167)	96% (23/24)
com.huettermann.fit.application	100% (4/4)	73% (8/11)	92% (176/191)	83% (35/42)
[all classes]				
EMMA 2.0.5312 (C) Vladimir Koubtsov				

Abb. 3: EMMA-Report

Testisolation und -integration

Denken Sie immer daran, Ihre Anwendung architektonisch zuzuschneiden und isolierte Tests anzusetzen. Sie brauchen nicht zwingend einen Service oder eine Datenbankkommunikation über die grafische Benutzeroberfläche zu testen.

Nutzen Sie im Zusammenspiel mit TestNG das Mocking-Werkzeug EasyMock [EasyMock], um Um- und Teilsysteme, Ressourcen und sonstige in diesem Kontext nicht „unter Test“ stehende Komponenten intelligent zu simulieren. Diese Mocks im Workspace angewendet, erlauben schnelle Testzyklen und Rückkopp-



SCHWERPUNKTTHEMA

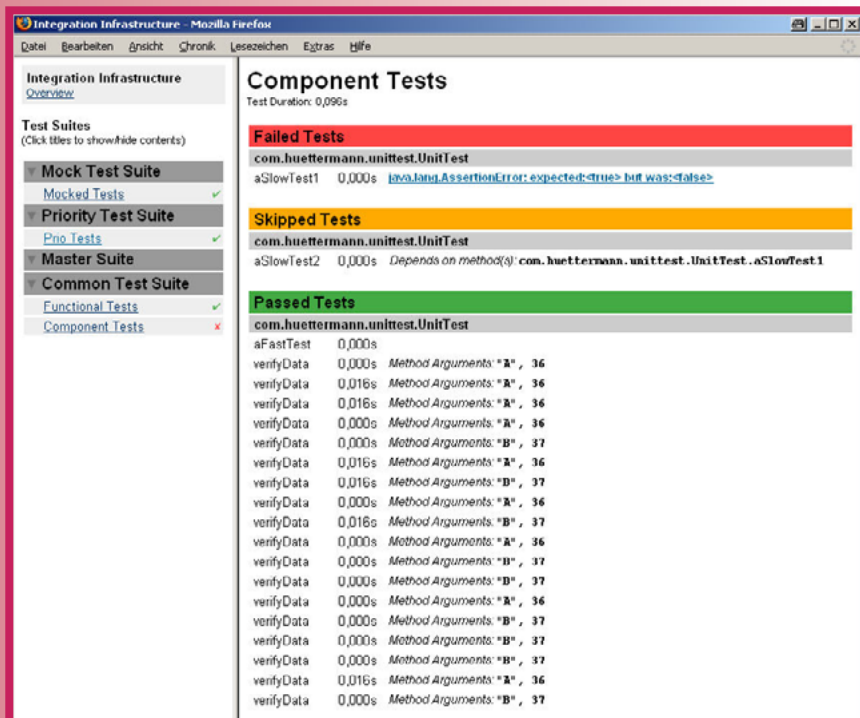


Abb. 4: TestNG: Aggregiertes Reporting über Testkategorien hinweg

lungen über Ihre Arbeiten. Deaktivieren Sie dann zentral und datengetrieben diese Mocks für Testläufe auf zentralen Integrationsmaschinen. Listing 3 zeigt das intelligente Wegmocken einer ressourcenintensiven, externen Schnittstelle. Achtung: Mocks sind keine einfachen Dummies oder Stubs, vergleiche dazu auch [JaNo09] in einer der kommenden Ausgaben.

Nicht nur zur Umsetzung von Komponententests, auch zwecks Integration Ihrer unterschiedlichen Testkategorien ist TestNG prädestiniert. Kombinieren Sie Komponenten- und funktionale Tests via TestNG und überbrücken Sie somit die technische Testbarriere. Das resultierende Ergebnisdokument bietet einen zentralen Einstieg in Ihren Testkosmos und eröffnet effizientes, aggregiertes Reporting (s. Abb. 4). TestNG kann sowohl in Ihre IDE inkludiert, als auch in das Buildsystem aufgenommen werden.

Standards, Audit

Standards beschreiben insbesondere die für das Projekt aufgesetzten Vorgaben an Design und Code. Konventionen in Word-Dokumenten sind hilfreich, dort schaut aber kaum jemand hinein. Ferner sind sie nicht effizient und ohne Medienbruch auf die Anwendung anwendbar.

Zur Validierung von Standards haben sich Werkzeuge wie Checkstyle [Checkstyle], PMD [PMD] und Findbugs [FindBugs] etabliert. Diese Konventionen sind ausführbar und lassen sich dem VCS hinzufügen. Check-

style hilft beispielsweise bei der Orientierung an den „Sun Java Coding Guidelines“. Findbugs unterstützt das Auffinden von Design Flaws und fehleranfälligen, irreführendem Code. So werden alle Negativmuster des Buchs [BIGa] identifiziert. Ferner unterstützt Findbugs den Java Specification Request 305: „Annotations for Software Defect Detection“. PMD basiert auf Regeln, die beispielsweise ungenutzte Variablen finden oder auf zu komplexen Code aufmerksam machen. Die Werkzeuge werden mit mitgelieferten Konfigurationen ausgeliefert, die als Vorlage dienen können. Sie sollten diese aber immer an Ihre eigenen Bedürfnisse anpassen.

Die Überprüfungen sollten in den Workspace eingebunden werden, um die diagnostische Lücke zu minimieren. Je später Sie Fehler finden, desto höher die Wahrscheinlichkeit von Folgefehlern und desto teurer die Behebung. So gibt es für Eclipse jeweils Plug-Ins um Checkstyle, PMD oder Findbugs einzubinden und bereits während der Entwicklung sehr agil auf Defekte aufmerksam gemacht zu werden. Diese Werkzeuge nur in den zentralen Buildprozess zu integrieren, ist also nur der halbe Weg, der aber in dieser Form dennoch zielführender wäre, als ausschließlich die Arbeitsplätze mit IDE-Plug-Ins auszustatten. Nur zu gern ignoriert ein Entwickler eine IDE-Warnung oder konfiguriert eine Regel lokal schlichtweg um.

Kommunikation, Wissensaustausch

Tooling kann selbstverständlich keine Kommunikation ersetzen. Dennoch muss die Infrastruktur den Informationsfluss unterstützen und die Kommunikation anregen. Es hat sich bewährt, hierzu ein Wiki einzusetzen, auf dem für das Team interessante Informationen fortgeschrieben werden können. Hier können die aus dem Integrationsbuild erstellten Artefakte automatisch hinterlegt und die aktuelle Verfassung der Software illustriert werden. Dieses Blutbild der Software setzt sich zusammen aus dem Ergebnis der Tests, Testabdeckung und Metriken.

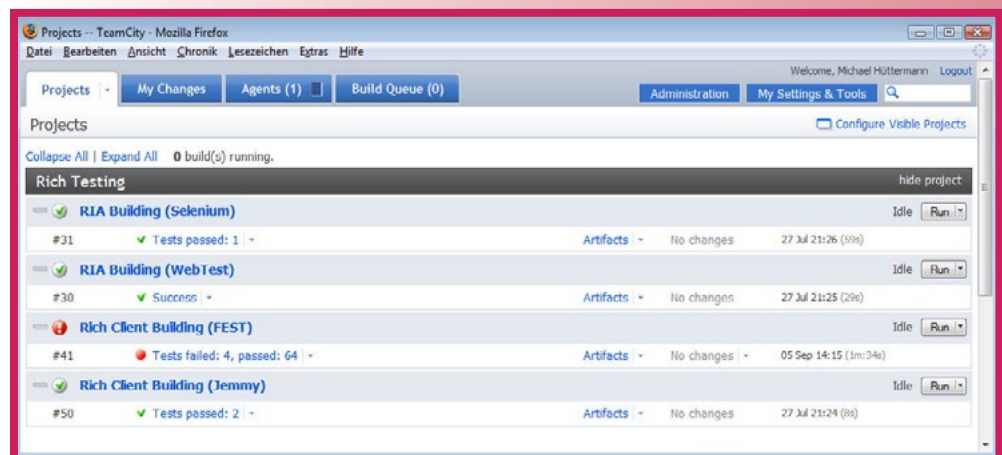


Abb. 5: Continuous Integration mit TeamCity



Wikis wie Trac [Trac] erlauben eine Integration mit Subversion und dienen durch ihre Features wie Roadmap und Ticketing System als leichtgewichtiges Projektmanagement-Werkzeug, in dem auch Anforderungsdokumente eingebettet fortgeschrieben werden könnten. Das kommerzielle Wiki Confluence [Confluence] ist ebenfalls zu empfehlen. Kommt für Sie kein Wiki in Frage, denken Sie über eine Projektseite nach, die Ihnen auch Maven liefern kann. Abbildung 1 zeigt ein Beispiel einer solcher Seite, die neben Audits und Testabdeckungsuntersuchung auch die Ergebnisse von funktionalen (Fit-)Tests und Komponententests beinhaltet.

Continuous Integration

Ihre auf Ant und/oder Maven basierten Builds lassen sich über Continuous Integration [CI] Server wie CruiseControl [CC] oder den in der Professional Edition kostenlosen TeamCity [TC] fortlaufend anstoßen und auswerten. Abbildung 5 zeigt mehrere in TeamCity verankerte Buildkonfigurationen. Charmant dabei ist die funktionsreiche Web 2.0-Ajax-Oberfläche, die über Ereignisse aus den Builds in Echtzeit informiert, Buildstatistiken anbietet und die Möglichkeit besitzt, vermeintlich hängende Builds zu identifizieren. Auch eine enge, aber dennoch entkoppelte Verzahnung mit IDEs wie Eclipse fördert agiles Arbeiten, z. B. durch das Anstoßen von privaten Builds via „Remote-Run“ und der automatische Commit durch „Delayed Commit“. Auf diesem Wege werden tatsächlich die Buildskripte genutzt, der eigene Rechner aber nicht blockiert. Welchen Build-Server Sie auch nutzen, prüfen Sie den Einsatz eines Build-Stagings. Kommen im Workspace noch Mocks und Smoke-Tests zum Einsatz, wird auf einer ersten zentralen Buildinstanz die Software kompiliert und ein Grundsatz von Tests angewendet. Dieser Build könnte bei jedem Commit angestoßen werden. Auf einer zweiten Buildinstanz können dann nachgelagert umfangreichere Tests laufen gelassen werden, beispielsweise das komplette Repertoire an Komponente- und funktionalen Tests. In diesem Kontext können Maven-Profile und TestNG-Gruppen eine elegante Steuerung ermöglichen. Die via Maven generierte Seite können Sie über das Buildportal ergänzend verfügbar machen.

Zusammenfassung

Ob Sie nun einen umfassenden agilen Entwicklungsprozess leben oder einen eher traditionellen: Ein zielgerichtetes Tooling wird helfen, Ihre agilen Entwicklungsmustern zu begleiten. Durch Konzentration auf Werkzeuge wie die hier vorgestellten und deren richtige Nutzung wird es gelingen, Ihren Entwicklungsprozess transparent und effizient zu halten und Artefakte in kurzen Zeitabständen bereitzustellen. Dabei sollte die Qualität immer eine Konstante sein. Die „Building Blocks“ SCM, Testing, VCS, Standards, Continuous Integration und Wissensaustausch können beim Aufbau einer technischen Infrastruktur als Orientierungshilfe dienen [Huet08].

Literatur und Links

[Ant] Buildwerkzeug Ant, <http://ant.apache.org/>
 [BIGa] J. Bloch, N. Gafter, Java Puzzlers – Traps, Pitfalls, Corner Cases, Addison-Wesley, 2005
 [CC] CI-Werkzeug CruiseControl, <http://cruisecontrol.sourceforge.net/>

[Checkstyle] Code Smell Detector Checkstyle, <http://checkstyle.sourceforge.net/>
 [CI] M. Fowler, Continuous Integration, 2006, <http://martinfowler.com/articles/continuousIntegration.html>
 [Confluence] Confluence-Enterprise Wiki, <http://www.atlassian.com/software/confluence/>
 [DDD] E. Evans, Domain-Driven Design, Addison-Wesley, 2003
 [EasyMock] Testisolation mit EasyMock, <http://www.easymock.org/>
 [EMMA] EMMA, Java Code Coverage, <http://emma.sourceforge.net/>
 [FEST] Java UI Testing-Bibliothek FEST (Fixtures for Easy Software Testing), <http://fest.easytesting.org/>
 [FindBugs] Find Bugs in Java Programs, <http://findbugs.sourceforge.net/>
 [Fit] Akzeptanztest-Framework Fit (Framework for Integrated Testing), <http://fit.c2.com>
 [Huet08] M. Hüttermann, Agile Java-Entwicklung in der Praxis, O'Reilly, 2008
 [Huet08a] M. Hüttermann, Agiles Testen von Java-Rich-UI-Anwendungen, in: JavaSPEKTRUM, 02/2008
 [Huet08b] M. Hüttermann, Funktionales Testen von Rich Internet Applications, in: JavaSPEKTRUM 05/2008
 [JaNo09] A. Janus, Ch. Nowak, Testen mit JMockIt, geplant in JavaSPEKTRUM, 01/2009
 [m2eclipse] Maven Integration for Eclipse, <http://m2eclipse.codehaus.org/>
 [Man] Das agile Manifest, <http://www.agilemanifesto.org/>
 [Maven] Integrationswerkzeug Maven, <http://maven.apache.org/>
 [PMD] Code Problem Detector PMD, <http://pmd.sourceforge.net/>
 [Rov70] W. W. Rovce, Managing the Development of Large Software Systems, in: Proceedings of IEEE WESCON, August 1970, <http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf>
 [Sel] Framework zum Testen von RIAs, Selenium, <http://www.openqa.org/selenium/>
 [SVN] Versionskontrollsystem Subversion, <http://subversion.tigris.org/>
 [TC] CI-Werkzeug TeamCity, <http://www.jetbrains.com/teamcity/>
 [TestNG] Komponenten- und Integrationstest-Werkzeug der „nächsten Generation“, <http://testng.org>
 [TorSVN] Subversion-Zugriff aus Explorer, TortoiseSVN, <http://tortoisesvn.tigris.org/>
 [Trac] freies, webbasiertes Projektmanagement-Werkzeug Trac, <http://trac.edgewall.org/>
 [WebTest] Framework zum Testen von RIAs, <http://webtest.canoo.com>
 [Well08] T. Wellhausen, Testabdeckung mit Open-Source-Tools, in: JavaSPEKTRUM, 06/2008



Sun Java Champion **Michael Hüttermann** ist freiberuflicher Experte für Java/JEE, SCM, SDLC Tooling und agile Softwareentwicklung. In seinem Buch „Agile Java-Entwicklung in der Praxis“ beschreibt er Methodik und Infrastruktur der agilen Entwicklung mit Java. Er ist zertifizierter SCJA, SCJP, SCJD, SCWCD, Mitglied des JCP und der Agile Alliance, Gründer und Organisator der Java User Group Köln, java.net JUGs Community Leader, Sprecher auf internationalen Konferenzen wie JavaOne, Jazoon und DevOxx sowie Autor zahlreicher Artikel. Er war im Programmkomitee der JavaOne 2008. E-Mail: michael@huettermann.net.