Gegenseitige Beeinflussungen von Testautomatisierung, Testmanagement und Entwicklung

Jan Düttmann[1], Stephan Kleuker[2]

[1]Archimedon Software GmbH
Marienstr. 66
D-32427 Minden
Jan.Duettmann@archimedon.de

[2]Hochschule Osnabrück
Barbarastr. 16
D-49076 Osnabrück
S.Kleuker@HS-Osnabrueck.de

Abstract: Die Automatisierung von Testprozessen bietet ein hohes Einsparungspotenzial bei der Software-Entwicklung. Abhängig vom zu testenden Produkt muss dazu ein passender Automatisierungsansatz mit den richtigen Werkzeugen, Vorgehensweisen und abgestimmten Entwicklungsprozessen gefunden werden. Der Erfolg hängt dabei auch davon ab, dass die richtigen Tests entwickelt, diese Tests strukturiert verwaltet und Testvorgaben in der Entwicklung berücksichtigt werden. Dieser Bericht abstrahiert Erfolgsfaktoren ausgehend von der Einführung einer Testautomatisierung bei einem kleinen Software-Entwicklungsunternehmen, die bei der Gestaltung eines individuellen Testprozesses eine wesentliche Rolle spielen, damit auch die Testautomatisierung mit neuen technologischen Herausforderungen mitwachsen kann.

1 Einleitung

Die typische Historie eines kleinen oder mittelgroßen Unternehmens (KMU) im Bereich Software-Entwicklung besteht meist darin, dass Experten eines Anwendungsbereichs feststellen, dass sie eine Software benötigen, die auf ihre individuellen Anforderungen zugeschnitten ist, diese aber nicht finden. Dies führt entweder zur Neugründung einer Firma oder eine Software wird in einer Firma innerhalb des Anwendungsbereichs entwickelt. Anschließend stellen Mitarbeiter dann fest, dass diese neu entwickelte Software auch für andere Unternehmen nutzbar sein kann. Schließlich wird die Ausgründung einer Entwicklungsfirma initiiert. Für den Erfolg des KMU ist es nun essenziell, die speziellen Wünsche des Anwendungsbereichs zu kennen, um sich gegen andere Hersteller, vielleicht funktional mächtigerer aber aufwändig anzupassender Software, durchzusetzen. Oft auch ohne Nachhaltigkeit im Kalkül zu haben.

Mit den ersten Markterfolgen werden Schritt für Schritt neue Ideen und individuelle Kundenwünsche in die Software eingebaut. Dabei muss ein oftmals kleines, eng zusammenarbeitendes Team von Entwicklern früher oder später personell ergänzt werden.

Schließlich muss ein reibungsloser Übergang von der „individuellen Heldenprogrammierung“, bei der alle Beteiligten alle Details des Programm-Codes kennen, zum systematischen Software Engineering stattfinden. Zwar gibt es hierzu eine Vielzahl hilfreicher Literatur, allerdings ist die Frage, welcher Teilprozess als erstes verbessert werden sollte, nicht einfach zu beantworten. Falsche Entscheidungen können hierbei leicht die Existenz eines KMU gefährden, schließlich muss das tägliche Geschäft kontinuierlich weiterlaufen.

Diese Arbeit zeigt zunächst, wie auf Grundlage einer sauberen Analyse eine erfolgreiche Verbesserung des Software-Entwicklungsprozesses im Bereich der Testautomatisierung mit Schwerpunkt auf dem GUI-Test durchgeführt wurde. Ausgehend von Erkenntnissen der Fallstudie werden Kriterien abgeleitet, die maßgeblich für ein systematisches erfolgreiches Testen sind. Dabei zeigt sich schnell, dass wesentliche Faktoren auch außerhalb des Kernbereichs der Qualitätssicherung (QS) liegen. Das Ziel ist es, langfristig erfolgreiche Software-Systeme zu gestalten, deren Entwicklung hier unter dem Begriff „Longterm Software Engineering“ (LotSE) zusammengefasst wird. LotSE beschäftigt sich mit den notwendigen Prozessen, die frühzeitig in einer Software- Entwicklung berücksichtigt werden müssen, damit Software über viele Jahre wart- und erweiterbar bleibt. Dabei müssen die Prozesse im Laufe der Entwicklungszeit mit der Produktgröße wachsen und kontinuierlich auf notwendige Korrekturmaßnahmen und mögliche Optimierungen hin untersucht werden.

Die Arbeiten wurden teilweise mit Mitteln des Bundesministeriums für Bildung und Forschung (BMBF) im Projekt KoverJa (Korrekte verteilte Java-Programme) durchgeführt.

2 Fallstudie

Dieses Kapitel fasst anfänglich wesentliche Ergebnisse der in [Hei12] ausgearbeiteten und in [HK12] aufbereiteten Fallstudie zusammen und ergänzt aktuelle Erkenntnisse beim Einsatz der Testautomatisierung.

2.1 Produkt und Unternehmen

Die Firma Archimedon mit den Standorten Minden und Osnabrück ist 2004 aus dem Technologiekonzern ABB hervorgegangen, einem der führenden Unternehmen für Anlagenbau, Energie- und Automatisierungstechnik.

Die modular gestaltete Softwarefamilie admileo® ist eine professionelle Lösung für zukunftsorientiertes Multiprojektmanagement (MPM), Organisations- und Personalmanagement (OPM), Geschäftsmanagement (GSM), Produktlebenszyklusmanagement (PLM) sowie für eine intelligente, abteilungsübergreifende Aufgabensteuerung. admileo® lässt sich nahtlos an bestehende ERP- und HRM-Systeme ankoppeln, wobei ein Datenaustausch bis auf tiefste Ebenen ermöglicht wird.

Die Project and Business Software admileo basiert ursprünglich auf einer internen Softwarelösung für das Management von Großprojekten im Kraftwerksbau. Das System wurde seit 2006 mit einer komplett neuen technischen Basis unter konsequenter Nutzung von Java neu entwickelt. Es wird seitdem kontinuierlich erweitert. Etwas detaillierter handelt es sich bei admileo um ein modular aufgebautes Client-Server-System mit Datenbankanbindung, welches fast ausnahmslos in Java implementiert ist. Der Server bildet die Schnittstelle zur Datenbank und übernimmt die Benutzerverwaltung und die Steuerung der Kommunikation mit den Clients. Er kann auf diese Weise zum Beispiel die Aktualisierung der Clients veranlassen, wodurch das System eine kollaborierende Arbeitsweise der Nutzer in Echtzeit gewährleistet. Die Benutzeroberfläche von admileo basiert auf dem Swing-Framework und verwendet zahlreiche modifizierte und erweiterte Swing-Elemente für die Darstellung und die Funktionalität der GUI-Elemente. Anwendungsbeispiele hierfür sind die grafische Aufwertung von Navigationsbäumen und die Darstellung und Editierung von Graphen und Diagrammen.

Abbildung 1: Anwendermodule von admileo

Die vier Hauptbereiche von admileo bestehen derzeit aus 24 Anwendermodulen, siehe Abbildung 1, und 25 Konfigurationsmodulen für eine individuelle Prozessanpassung in der Kundenumgebung. Das System besitzt einen Umfang von weit über 1,5 Millionen Lines of Code (LoC).

Die Software admileo ist das Premiumprodukt der Archimedon Software + Consulting GmbH & Co. KG [ASC], einem KMU mit derzeit 18 Mitarbeitern, das auf die Entwicklung von Java Software spezialisiert ist. Die Firma mit Hauptsitz in Minden (NRW) unterhält seit 2010 eine Zweigstelle in Osnabrück (NDS). Archimedon arbeitet eng mit der Hochschule Osnabrück zusammen. Die Firma legt sowohl bei der Softwareentwicklung

als auch in anderen Geschäftsprozessen großen Wert auf systematisches und strukturiertes Vorgehen und wiederholbare Prozesse. Dies zeigt sich beispielsweise darin, dass Archimedon ISO-9001:2008 zertifiziert ist und ausschließlich auf Entwickler mit Hochschulabschluss in der Informatik setzt. Bei der Softwareentwicklung hat Archimedon von Anfang an langfristig geplant. Ausgehend von einem Lebenszyklus der Software admileo von 15-20 Jahren sind schon in der Designphase Entscheidungen für eine möglichst einfache Erweiterbarkeit, eine hohe Wartbarkeit und Wiederverwendbarkeit sowie eine bestmögliche Qualität getroffen worden. Hierzu zählen vor allem die Modularisierung und die komponentenbasierte Softwareentwicklung. Für eine nachhaltige Sicherstellung dieser Ziele, führt die Firma regelmäßig Projekte zur Analyse und zur Verbesserung der eingesetzten Prozesse durch und evaluiert neue Techniken und Technologien.

Bereits beim Start der Neuentwicklung lag ein wichtiges Augenmerk auf der Erweiterbarkeit der Software. Das anfänglich noch kleine Team von Entwicklern folgt einem definierten Software-Entwicklungsprozess. Technisch wurde sich sofort für eine Modul- bzw. Komponentenstruktur entschieden, die eine möglichst unabhängige Entwicklung weiterer Module ermöglicht. Grundlage dafür ist eine Postgres-Datenbank, auf die eine mit JPA vergleichbare Persistenzschicht zur Vereinheitlichung der Datenbanknutzung gelegt wurde.

Für neue Module werden zunächst Spezifikationen in Form von Lastenheften geschrieben. Diese Prototypen dienen auch zur Analyse und zur Auswahl von Lösungsvarianten. Das zentrale Datenmodell ist so dokumentiert, dass Abhängigkeiten zwischen Modulen schnell erkennbar sind.

Von Beginn an spielte die Qualitätssicherung des Produktes eine wichtige Rolle. Es wurde darüber hinaus von Beginn an das Ziel verfolgt, sämtliche Schritte im Software-Entwicklungsprozess von der Anforderungsanalyse bis hin zur Qualitätssicherung möglichst vollständig zu automatisieren. So wurde aus der Anforderungsdefinition heraus jede Änderung und Neuentwicklung des Systems über ein workflowbasiertes Aufgabensteuerungsmanagement durchgeführt, was eine sehr detaillierte Versionsplanung sowie ein effizientes Projekt-Controlling ermöglichte. Die Automatisierung von Oberflächentests wurde zwar zu Beginn der Produktentwicklung evaluiert, jedoch auf Grund der mangelnden Werkzeugunterstützung zunächst wieder hinten angestellt. Aus diesem Grund akzeptierten die Entwickler die hohe Produktverantwortung, Fehler durch individuelle Entwicklertests zu finden.

So wurden für die frühen/ersten Versionen intensiv manuelle Tests durchgeführt, die das gesamte Zusammenspiel in admileo als Integrations- und Systemtest durch mehrere Personen, meist Entwickler, betrachten. Diese Tests sind in Dokumenten spezifiziert, so dass sie jederzeit nachvollzogen werden können. Grundsatz hierbei ist das „Vier-Augen-Prinzip“, nach dem jeder Testfall einmal durch den beteiligten Entwickler und anschließend durch einen weiteren Mitarbeiter getestet wird.

Im Laufe der Entwicklung benötigte dieser manuelle Testprozess immer mehr Zeit, da konsequent nicht nur neue Module, sondern auch alte Module mitgetestet wurden, um das Einsickern von Inkonsistenzen in die Software zu vermeiden. Als zweite Optimie-

rungsmöglichkeit wurde der relativ starre Release-Zyklus gesehen, der die schrittweise Einführung neuer Funktionalität erschwerte.

Dies führte zur strategischen Unternehmensentscheidung, den Prozess der Qualitätssicherung im Rahmen eines Continuous Integration Prozesses zu automatisieren. Aus Sicht der langfristigen Wart- und Erweiterbarkeit sollten neue Software-Versionen schnell erstellt und möglichst vollautomatisch getestet werden. Dabei liegt der Fokus zunächst auf den genannten manuell ausgeführten Integrations- und Systemtests.

2.2 Neuer QS-Prozess

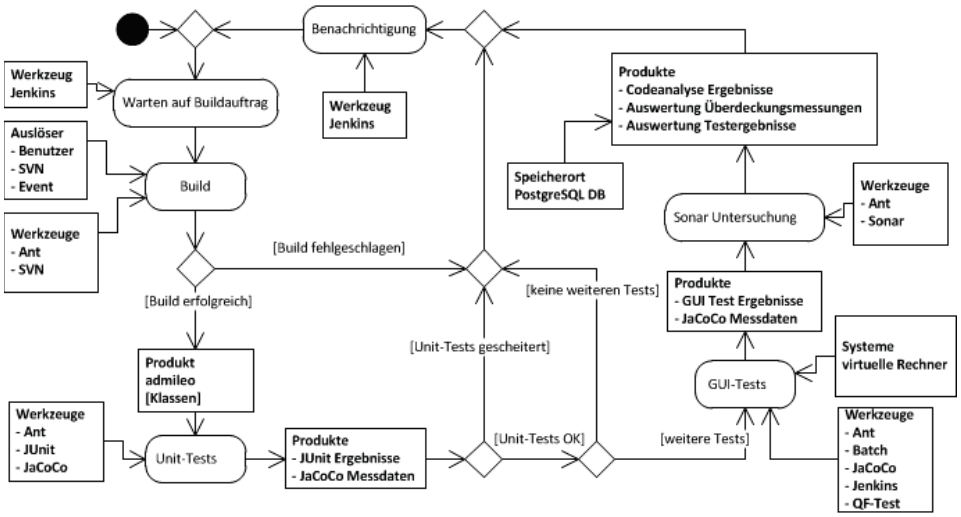

Abbildung 2: Build-Prozess mit integrierter Qualitätssicherung [HK12]

Das in Abbildung 2 gezeigte Schaubild gibt einen Überblick über den automatisierten Continuous-Integration-Prozess von der Integration einer Softwareänderung bis hin zur automatisierten Qualitätssicherung. Die wesentlichen Schritte des Prozesses sind:

Build: Mit Hilfe des Werkzeugs Jenkins wird hier auf ein Ereignis gewartet, welches einen kompletten Integrationslauf startet (üblicherweise das Einchecken von Quellcode oder die manuelle Ausführung). Ebenfalls übernimmt Jenkins die Steuerung des Build-Vorgangs, indem es über ein Auschecken des Quellcodes aus Subversion die für die lauffähige Software benötigten Artefakte durch Ant-Skripte erzeugt.

Unit-Test: Die ausführbare Software wird nun mittels JUnit den Unit- und Integrationstests unterzogen. Ant übernimmt dabei auch hier wieder die Ausführung. Mittels eingebundenem JaCoCo wird zeitgleich die Testüberdeckung protokolliert. Als Ergebnisse dieses Schrittes werden JUnit-Testergebnisse und die Messdaten der Testüberdeckung abgespeichert.

GUI-Tests: Im nächsten Schritt wird das Werkzeug QF-Test eingesetzt, um die Benutzeroberflächen-Tests durchzuführen. Um sicherzustellen, dass bei sämtlichen Tests die gleiche Ausgangssituation vorliegt, werden diese Tests auf virtualisierten Rechnern ausgeführt, auf denen eine Jenkins-Instanz im Slave-Betrieb läuft. Auch hier werden wieder die Testergebnisse und die Messdaten der Testüberdeckung protokolliert.

Sonar-Untersuchung: Dieser Ablaufschritt besteht im Wesentlichen aus zwei Vorgängen: Zum einen wird der gesamte Quellcode des Produktes einer statischen Codeanalyse mit PMD und Checkstyle unterzogen. Zum anderen werden die Ergebnisse dieser Analyse mit den zuvor gesammelten Testergebnissen und Überdeckungsdaten kombiniert in der Qualitätsmanagement-Plattform Sonar abgespeichert.

Integrationssysteme: Sind sämtliche Schritte erfolgreich abgelaufen, werden die sogenannten Integrationssysteme erzeugt, welche mit lauffähigen Demo-Daten gefüllt werden, um ständig die aktuelle Software präsentieren und den momentanen Entwicklungsstand nachvollziehen zu können.

Für jeden einzelnen Schritt im Build-Vorgang gilt: Scheitert der entsprechende Vorgang, bricht der Prozess an dieser Stelle mit einer Benachrichtigung an die verantwortliche(n) Person(en) ab. Damit wird gewährleistet, dass auf die Integration einer Software-Änderung schnellstmöglich eine Rückmeldung aus dem Continuous-Integration-System darüber erfolgt, welche negativen wie positiven Auswirkungen die soeben getätigte Änderung auf das Produkt hat.

2.3 Fazit der Prozessanpassung

Zur rein technischen Umsetzung des Integrations- und Testprozesses lässt ist festzustellen, dass diese sehr reibungslos implementiert werden konnte und sich nahtlos in den bisherigen Entwicklungsprozess einfügt. Sämtliche hier verwendeten Tools sind so ausgereift und stabil, dass sie in der alltäglichen Verwendung keine nennenswerten Probleme bereiten. Der Betrieb eines solchen Continuous Integration- und Testing-Systems bringt einen Wartungsaufwand von wenigen Mannstunden pro Monat mit sich, dem stehen viele Vorteile gegenüber, die eine solche Umgebung vollautomatisch mit sich bringt. Insbesondere sind hier zu nennen:

- Ständige Verfügbarkeit von ausführbaren Testsystemen für manuelle Tests sowie Produktpräsentationen
- Ausführung aller automatisierten Tests ohne Benutzereingriff
- Risikominimierung für den Releasewechsel beim Kunden (dieser wird mit jedem Build-Schritt mitgetestet)

Die wesentliche Herausforderung bei der Umsetzung des Prozesses liegt daher weniger in der technischen Implementierung der Werkzeuge und Verfahren, sondern vielmehr in der Integration der Testautomatisierung in den Entwicklungsprozess. Damit ein kontinuierlicher Testprozess seine Vorzüge voll entfalten kann, ist es unabdingbar, die Umstellung von manuellen auf automatische Tests möglichst weitreichend durchzuführen. Hier-

bei hat es sich als Erfolgsfaktor herauskristallisiert, die technische Verantwortung für das Testwerkzeug an eine definierte Stelle zu geben. Grund dafür ist, dass der Aufwand, ein gesamtes Entwicklungsteam in sämtliche Tiefen eines Testwerkzeugs wie QFTest einzuarbeiten, sehr hoch ist. Darüber hinaus bietet QFTest die Möglichkeit, Testfälle dahingehend zu strukturieren, dass sämtliche Benutzerinteraktionen mit dem zu testenden System in Bibliotheken als Einzelfunktionen hinterlegbar sind, und optimaler Weise in den Testfällen selbst diese nur noch aneinandergereiht werden müssen. Darüber hinaus hat eine teamübergreifende Struktur dieser Bibliotheken und auch der Testfälle selbst den gleichen Effekt der Wiederverwendungsmöglichkeit von bereits implementierten Methoden wie bereits aus der Software-Entwicklung selbst bekannt.

3 Lessons Learned

Die Kapitel fasst, motiviert von den Ergebnissen des vorherigen Kapitels, wesentliche Erfolgsfaktoren für ein langfristig wart- und erweiterbares Projekt mit dem Fokus auf die QS zusammen. In die Ergebnisse fließen Kenntnisse aus weiteren Projekten ein, die im Bereich QS und Software-Engineering oft als Abschlussarbeiten bei weiteren Unternehmen, meist KMU, durchgeführt wurden.

3.1 Testarchitektur

Der Hauptkritikpunkt an einer Testautomatisierung ist, dass die entstandenen Tests sehr gut zum aktuell getesteten System passen, allerdings mit der Weiterentwicklung des Systems immer wertloser werden. Die Ursachen sind meist, dass die Anpassung der Tests als zu aufwändig angesehen wird und dass neue Tester die Einarbeitung in die komplexe Testentwicklung scheuen.

Die elementare Lösung hier ist die Modularisierung von Tests, die zum systematischen Aufbau einer Testarchitektur führen. Dies lässt sich leicht anhand von Tests einer Oberfläche veranschaulichen, da gerade Oberflächen oft geändert werden. In einem schlechten Testsystem wird in jedem Test direkt jeder Mausklick auf bestimmte Knöpfe und jede Ausgabe in bestimmten Feldern ausprogrammiert. Wesentlich sauberer werden die Tests, wenn zunächst eine Basisfunktionalität aufgebaut wird, in der in kleinen Methoden einzelne Elemente der Oberfläche bedient oder ausgelesen werden. Findet dann nur eine Umgestaltung der Oberfläche statt, sind oft nur diese elementaren Methoden zu verändern, da für die nutzenden Tests nur wichtig ist, dass überhaupt diese Funktionalität genutzt wird. In [Kle13] befindet sich dazu ein Beispiel, bei dem für eine Java-Swing-Oberfläche ein nahtloser Wechsel von einem Test-Framework, das Swing direkt nutzt (FEST, Fixtures for Easy Software Testing [@Fes]) zu einer Test-Bibliothek, die beliebige Oberflächen steuert (Sikuli Java API [@Sik]) für eine kleine Fallstudie gezeigt wird. Durch Unterschiede in der angebotenen Funktionalität kann so ein nahtloser Übergang nicht immer vollständig garantiert werden.

Eine Testarchitektur ist damit ein zentrales Mittel, flexibel auf Änderungen im zu testenden System oder auf neue Möglichkeiten zu reagieren. Die Testarchitektur spiegelt oft in

vereinfachter Form die Software-Architektur wieder, da so die Tests der einzelnen Bausteine und die Integrationstests in natürlicher Weise strukturiert werden. Etwaige Mängel der Software-Architektur werden dabei mittelfristig auch sichtbar. Dies wurde in einer anderen Studie [Wit13] für eine kommerzielle Web-Software deutlich, deren Oberfläche nach gut fünf Jahren neu gestaltet werden sollte. Die ursprüngliche Oberfläche wurde mit JSF 1.1 entwickelt, einer relativ frühen Version, die recht wenig graphische Möglichkeiten mitbringt und mit der die Entwicklung recht aufwändig ist. Dies gilt insbesondere, wenn man sich aktuelle Oberflächen-Technologien und Frameworks ansieht. JSF basiert auf einer Model-View-Controller Variante, die eine erste Strukturierung bei der Architektur vorgibt. Dieser Ansatz, zusammen mit einem sauber aufgebauten fachlichen Modell und der zugehörigen Datenhaltung, war Grundlage des ursprünglichen Systems. Der Architekturansatz war für das bisherige Projekt erfolgreich, was sich dadurch zeigte, dass die wachsende Software wart- und erweiterbar blieb. Durch die Forderung nach einer neuen Oberflächen-Technologie musste diese Bewertung leicht revidiert werden. Voruntersuchungen zeigten, dass es eine Vielfalt von Alternativen gibt, von denen zwei mit Prototypen genauer verfolgt wurden. Die Ergebnisse zeigen, dass generell neue Technologien nutzbar sind, allerdings eine einfache Abtrennung oberhalb der Model-Schicht nicht vollständig möglich war. Im konkreten Fall ist der Ursprung des Problems eine Klasse FacesContext aus JSF, die sehr mächtige Möglichkeiten bietet, in die gesamte JSF-Steuerung einzugreifen. JSF basiert generell auf Servlets, wobei dies nur selten bei der Entwicklung zu beachten ist. Sucht man allerdings nach schnellen und effizienten Lösungen, wird oft die FacesContext-Klasse genutzt, da man über sie auf alle Details der Servlet-Nutzung zugreifen und den gesamten Verarbeitungsprozess modifizieren kann. Da in anderen Ansätzen eine solche Klasse nicht vorhanden ist, muss die gesamte Software und damit auch die zugehörigen Tests umgebaut werden. Da fast jedes Framework solche besonderen Möglichkeiten z. B. auch bei der Verknüpfung der Oberfläche mit Steuerungselementen wie Event- und PropertyListenern bietet, muss eine potenziell abtrennbare Schicht weiter gekapselt werden.

Ein solcher Architekturansatz wird z. B. mit dem Model-View-ViewModel-Ansatz verfolgt, der Grundlage des XAML-Oberflächenansatzes mit der Windows Presentation Foundation [Hub10] von Microsoft ist, der aber auch auf andere Ansätze übertragbar ist. Das ViewModel stellt die Informationen für die Anzeige bereit und übernimmt die Verknüpfung der Steuerungselemente der Oberfläche mit der Software. Auf abstrakter Ebene gibt es damit einen leicht wiederverwendbaren Anteil der Informationsbereitstellung und einen individuellen Anteil zur Umsetzung des Kommunikationsmechanismus. Bei einem Austausch der Oberflächen muss nur dieser individuelle Teil geändert werden, was sich auch unmittelbar auf die Testerstellung auswirkt. Sind die Tests modular aufgebaut, können sie mit relativ wenig Aufwand an die neue Software-Architektur angepasst werden. Dies macht den Zusammenhang zwischen guter Software- und guter Testarchitektur unmittelbar deutlich. Der generelle Aufbau eines Systems ist in Abbildung 3 skizziert.

Für Komponenten, die nicht eng in der Architektur vernetzt sind, wie z. B. Oberflächen, bietet sich dann eine Schicht mit WebServices an, über die die Oberflächen die benötigten Daten abrufen können. Dieser Ansatz verlangsamt eventuell die Umsetzung bei der Nutzung spezieller Frameworks, wobei dann individuell zu klären ist, ob dies Auswir-

kungen auf die Nutzbarkeit hat. Studien für kommerzielle Produkte haben gezeigt, dass durch die Nutzung von WebServices [Bur14][@RES] basierend auf JSON eine Trennung zwischen Client und Server keinen Effekt für den Endnutzer haben muss [Klo13]. Da durch WebServices die Kommunikation üblicherweise nur vom Aufrufer zum Aufrufenden geht, muss bei Bedarf darüber nachgedacht werden, wie eine echte bidirektionale Kommunikation hergestellt werden kann. Studien, wie in [Dit14] zeigen, dass WebSockets [@Web] einen sehr vielversprechenden Ansatz darstellen.

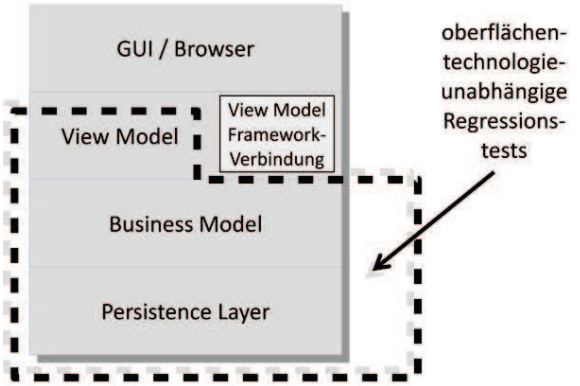

Abbildung 3: Basisarchitektur für wart- und erweiterbare Systeme

Aus Sicht der Qualitätssicherung haben die Überarbeitungen der betrachteten Systeme auch deutlich gemacht, dass es wichtig ist, dass Tests, insbesondere Abnahmetests klar spezifiziert sind, da sie die Grundlage der Tests für das neue System bilden. In den konkreten Fällen kommen klassische Testspezifikationen in Text-Form zum Einsatz.

3.2 Testorganisation und Testmanagement

Ohne den Einsatz spezieller Testmanagementwerkzeuge werden die Tests typischerweise in Unit-testbasierten Systemen recht einfach geordnet. Einzelne Tests werden Klassen oder Komponenten zugeordnet, diese einzelnen Tests werden dann pro Klasse, Komponente oder Teilsystem zu sogenannten Test-Suites zusammengefasst [Kle13]. Dabei können Test-Suites wieder Test-Suites enthalten, so dass ein hierarchischer Aufbau der Testorganisation entsteht, der eng mit der Software-Architektur verknüpft ist. Generell kann ein Test auch in mehreren Test-Suiten enthalten sein, so dass er gegebenenfalls mehrfach ausgeführt wird. Diese Strukturierungsmöglichkeit ist dann sinnvoll, wenn nicht immer alle Tests ausgeführt werden sollen, da man z. B. Zeit sparen möchte.

Arbeitet man mit einem Continuous Build Managementsystem [HF10], das beim Nightly Build auch alle Tests ausführt, kann es passieren, dass aus Zeitgründen nicht mehr alle Tests ausführbar sind. Hier wäre es sinnvoll, Tests mit Meta-Daten zu versehen, so dass man über Filterkriterien individuell eine passende Test-Suite für eine Nacht oder für eine bestimmte Entwicklungsphase auswählen kann. Ein solches System kann z. B. über die Analyse bestimmter strukturierter Kommentare in Tests oder durch Annotationen die

passenden Tests auswählen. Ein freies System, das sich leicht in Build Managementsysteme einbauen lässt, fehlt aber.

3.3 Neue Technologien

Die kontinuierliche Weiterentwicklung im Bereich Software Engineering hat unmittelbare Auswirkungen auf die Testentwicklung. Ein konkretes Beispiel sind die Möglichkeiten, die in Java durch CDI (Contexts and Dependency Injection [@CDI]) relativ neu für die Enterprise Edition (JEE), aber auch nutzbar in der Standard Edition (Java SE) hinzugekommen sind. CDI umfasst mehrere Möglichkeiten, die Umsetzung von Funktionalität alternativ zu programmieren. Ein wichtiger Ansatz ist, für Objektvariablen nur Schnittstellen zu nutzen, so dass durch verschiedene Implementierungen der Schnittstellen diese leicht gegeneinander ausgetauscht werden können. Im Fall von JEE kann der umgebende Server dafür verantwortlich sein, die passende Implementierung zur Verfügung zu stellen. Dies kann im konkreten Fall bedeuten, dass der Programmierer nur noch spezifiziert, dass er eine transaktionale Datenbankverbindung haben möchte.

```
@Inject
private PersistenzService db;
...
public void persist(Object object) {
  db.persist(object);
}
```

Im weiteren Programm wird dann die Variable db z. B. in der Methode persist() einfach genutzt, ohne dass ihr explizit einen Wert zugewiesen wurde.

CDI unterstützt weiterhin direkt, dass mit Hilfe einer XML-Konfigurationsdatei einfach angegeben werden kann, welche konkreten Klassen für welche Schnittstellen genutzt werden sollen. So wird es z. B. sehr einfach, Mock-Objekte zu injizieren. Dies kann bei der Erstellung von Tests den Aufwand beim Aufbau eines vollständigen Testobjekts deutlich erleichtern und die Lesbarkeit von Tests erhöhen.

Steht ein solcher Ansatz neu zur Verfügung, muss kritisch geprüft werden, ob er bei der Erstellung neuer Tests genutzt wird und wie mit existierenden Tests umgegangen werden soll. Man kann existierende Tests einfach übernehmen, da sie weiterlaufen sollten, was in Java fast immer garantiert ist. Dem wesentlichen Kriterium für die einfache Übernahme, dass keine zusätzliche Arbeit entsteht, ist gegenüber zu stellen, dass dann zwei verschiedene Konzepte in den Tests genutzt werden. Dies kann sich mittelfristig bei der Wartung der Tests negativ auswirken, da immer beide Konzepte von den Bearbeitern verstanden werden müssen.

Als weiteren Ansatz bietet CDI Möglichkeiten der Aspektorientierung. Dies beinhaltet, dass man festlegen kann, dass bei bestimmten Aktionen wie einem Methodenaufruf zusätzlicher Programmcode ausgeführt werden soll. In dem Programmcode kann dann auf die beteiligten Objekte zugegriffen und ihre Informationen ausgewertet werden. Innerhalb der aufgerufenen Funktionalität kann bei CDI festgelegt werden, ob die unterbrochene Aktion in der ursprünglichen Form ausgeführt oder abgebrochen werden soll.

```
@InterceptQualifier // Methoden dieser Klasse beobachten
public class EventConsumer {
   ...
}

@InterceptQualifier
@Interceptor // so annotierte Klasse beobachtet @InterceptQualifier
public class MeinInterceptor {

  @AroundInvoke // Methode wird vor jeder Methode von EventConsumer
                // aufgerufen, Details über Parameter abfragbar
  public Object logCall(InvocationContext context)
                              throws Exception {
    ...
    return context.proceed(); // ursprüngliche Methode fortsetzen
```

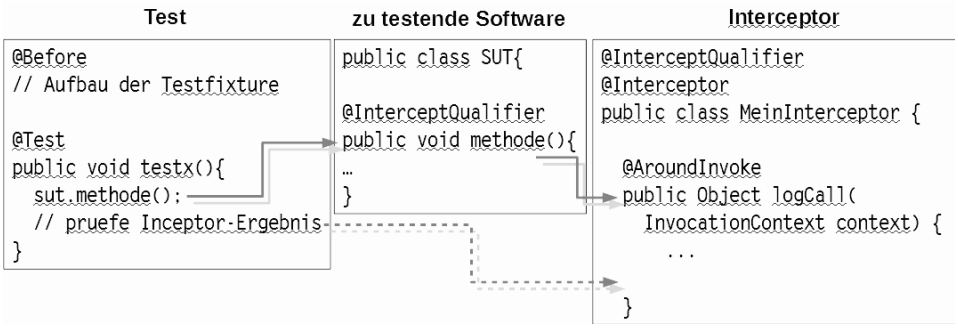

Abbildung 4: Aufruf von Tests mit Nutzung von Aspektorientierung

Dieser Ansatz bietet gerade aus Sicht des Testens auf Komponentenebene sehr gute Möglichkeiten, da Tests einfach dann „lauschende“ Methoden werden, deren Aufruf genau dann erfolgt, wenn die zu testende reale Aktion aufgerufen wird. Das gesamte Programm würde dann wieder im Rahmen normaler Tests aufgerufen werden, wobei für diese aspektorientierten Tests nicht sichergestellt ist, ob bzw. wann ihr genauer Aufruf erfolgt. Da man in den aspektorientierten Tests z. B. auch in globalen Klassenvariablen gehaltene Hilfsvariablen zum Zählen der Anzahl der Aufrufe ergänzen kann, ist diese andere Art der Testerstellung nur konsequent mit der klassischen Testerstellung zu verknüpfen. Die resultierende Aufrufstruktur ist Abbildung 4 dargestellt

CDI ist damit ein weiterer Baustein, der bei der Testplanung, was soll wann wie getestet werden, zu beachten ist. Angemerkt sei, dass das Thema Aspektorientierung schon relativ lange bekannt ist [Lad03] und es auch mehrere gut nutzbare Frameworks dazu gibt, die Verbreitung des Ansatzes aber immer gering war. Da durch CDI die Aspektorientierung ein weiterer direkt zu nutzender Baustein der Entwicklung wird, steigt dadurch die Popularität des Ansatzes deutlich an.

Die freie Mock-Bibliothek Mockito [@Moc] bietet neben der Mock-Erstellung für noch nicht realisierte Klassen die Möglichkeit, existierende Klassen oder einzelne Methoden davon zu mocken. Dies bedeutet, dass in der laufenden Software ein Methodenaufruf abgefangen und durch eigene Funktionalität ersetzt werden kann. Da die nicht gemockten

Methoden erhalten bleiben, kann so eine konkrete Implementierung einfach in einen Mock umgewandelt werden. Dies ist z. B. dann sehr nützlich, wenn man sicherstellen will, dass eine bestimmte Methode für konkrete Parameterwerte garantiert eine Exception wirft. Das nachfolgende Beispiel zeigt, dass der konkrete Aufruf get(0) für eine Liste statt dem eingetragenen Wert 1 den Wert 42 liefert.

```
public static void main(String[] args) {
    List<Integer> list = new LinkedList<Integer>();
    list.add(1);
    List<Integer> spy = Mockito.spy(list);
    Mockito.doReturn(42).when(spy).get(0);
    System.out.println(spy.get(0) +" :: " + spy.size());
}
```

Die Möglichkeiten des Ansatzes sind wieder weitreichend und verwandt mit den vorher genannten Möglichkeiten der aspektorientierten Tests. Neben der direkten Nutzung des Ansatzes bei der Testerstellung, ergibt sich wieder die Möglichkeit, Tests in die gemockten Methoden zu integrieren, die dann nicht direkt von der Teststeuerungs-Software sondern innerhalb anderer Tests aufgerufen werden.

Direkt auf den Testbereich bezogen sieht man die neuen Testmöglichkeiten z. B. an der kontinuierlichen Weiterentwicklung der integrierten oder leicht zu installierenden Erweiterungen in der Entwicklungsumgebung Visual Studio von Microsoft, was insbesondere die sogenannten Ultimate Editions betrifft. Relativ neu ist die Möglichkeit, unmittelbar ein ausführbares Programm zu beeinflussen, indem festgelegt wird, dass bei bestimmten Methodenaufrufen anderer Code ausgeführt wird, was verwandt mit dem vorgestellten Mockito-Ansatz ist. Das Microsoft Fakes Framework hilft dabei, zu testenden Code zu isolieren. Dabei werden Teile des Codes durch sogenannte Stubs oder Shims ersetzt. Dies sind kleine Code-Fragmente, die völlig unter Kontrolle der Tests stehen [@Mic]. Der Ansatz erlaubt es, fertige Software zu analysieren und zu beeinflussen, was auf konzeptioneller Ebene den vorgestellten Möglichkeiten der Aspektorientierung und damit CDI entspricht.

Natürlich gibt es auch viele Neuerungen, wie der in Java 8 eingeführte Lambda-Ansatz, die keine direkte Auswirkung auf geschriebene Tests oder die Testentwicklung haben. Bei neu zu schreibenden Tests können Tests durch den Lambda-Ansatz etwas kompakter, damit auch lesbarer geschrieben werden, so dass die Nutzung für neue Tests sinnvoll ist. Eine Reimplementierung existierender Tests ist aber nicht notwendig. Ähnliches gilt für das mit Java 8 neu eingeführte File-Handling, das deutlich angenehmer in der Nutzung für Entwickler ist, aber ursprüngliche Tests nicht beeinflusst.

3.4 Metriken und Überdeckungen

In der vorgestellten Fallstudie wurden Werkzeuge zur Messung einiger Metriken und zur Analyse der erreichten Überdeckungsgrade eingebaut. Beide Ansätze zeigen sich als hilfreich. Metriken eignen sich insbesondere zur Analyse der Komplexität einzelner Methoden und Klassen sowie zur Analyse der Anzahl der Verknüpfungen und damit Abhängigkeiten von Klassen. Mit Überdeckungen wird insbesondere festgestellt, welche

Bereiche des Programms nicht oder nur in geringem Maße getestet werden. Die genannten Maße sind aber mit Vorsicht zu betrachten, da sie immer nur Indikatoren liefern, aber weder eine Aussage erlauben, dass die zu testende Software und die Tests in einem guten Zustand sind, noch dass Verstöße direkt auf Probleme hinweisen müssen.

Dass es sich nur um Indikatoren handelt, ist unbedingt bei der Testerstellung zu berücksichtigen. Generell fokussieren Tests zunächst auf die typischen, vom Endnutzer später am meisten genutzten Abläufe, was natürlich für Systemtests aber oft auch für Unit-Tests gilt. Im nächsten Schritt werden die Randfälle und erwarteten Ausnahmefälle berücksichtigt. Erst dann ist es sinnvoll, auf Überdeckungen zu achten und z. B. die Fragestellung aufzuwerfen, warum gewisse Methoden in den erstellten Testfällen nicht berücksichtigt werden. Erst wenn sichergestellt ist, dass die Methode benötigt wird, sollte man für sie weitere Tests ergänzen. Sehr wichtig bleibt dabei trotzdem, dass die Überdeckung ein sehr trügerisches Maß sein kann, wie das Beispiel in Abbildung 5 zeigt, das [Kle13] entnommen ist und in [Spi14] mit weiteren Beispielen aufgegriffen wird.

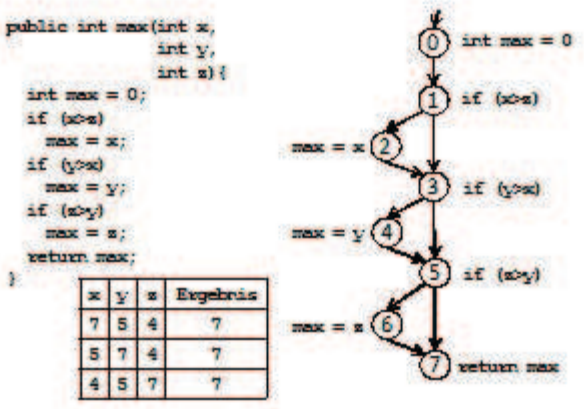

Abbildung 5: falsche Software mit perfekter Überdeckung

Die Abbildung 5 zeigt links eine einfache Methode zum Berechnen des Maximums dreier übergebener Werte. Die rechte Seite zeigt den zugehörigen Kontrollflussgraphen. Mit den linksunten spezifizierten Testfällen werden alle Kanten des Graphen genutzt, man spricht von einer vollständigen Zweigabdeckung (C1-Überdeckung) oder auch der Garantie, dass jeder mögliche Ablaufschritt einmal ausgeführt wurde. Trotz der vollständigen Testabdeckung, wobei in jedem Test das erwartete Ergebnis vorliegt, werden die zwei enthaltenen gravierenden Fehler nicht gefunden. Die Methode liefert zunächst immer den Wert null, wenn alle Parameter den gleichen Wert haben. Weiterhin liefert die Methode das falsche Ergebnis, wenn x der größte Wert und z größer als y ist.

3.5 Qualifikation von Mitarbeitern

Die vorherigen Kapitel machen sehr deutlich, dass die Anforderungen an die Qualifikation von Mitarbeitern der QS ständig steigen. Neben den zentralen Grundbegriffen und elementaren Prozessen der QS [SW01], müssen zumindest einige Mitarbeiter Erfahrungen in der Software-Architektur besitzen und sich kontinuierlich in neuen Entwicklungs- und Testtechnologien schulen.

Einen zentralen Ansatz bietet dabei die ISTQB-Zertifizierung [SL12] [SRW14], die es über mehrere Stufen ermöglicht, eine einheitliche Begriffswelt für Fachbegriffe und typische Prozesse zu schaffen. Gerade in KMU sollte ein enger Austausch zwischen Entwicklung und Qualitätssicherung erfolgen. Optimal hat ein Test-Team zumindest einen erfahrenen Entwickler, der analytisch sehr stark und in Prozessen und Methoden der Qualitätssicherung ausgebildet ist. Ein Austausch, bei dem Entwickler in einem Projekt zu Testern in anderen Projekten werden, kann die Qualität in beiden Bereichen verbessern, da bei der Entwicklung mehr über die Testbarkeit und bei der Testerstellung mehr über Programmierprobleme nachgedacht wird. Zu beachten bleibt, dass Kernaufgaben der Qualitätssicherung, wie der Aufbau von Testanlagen und das Testmanagement weiterhin parallel von erfahrenen QS-Mitarbeitern erfolgen müssen.

Ein deutlich kritischer zu bewertender Weg ist gerade bei der Einführung einer Testautomatisierung, QS-unerfahrene Mitarbeiter mit dem Aufbau eines solchen Prozesses zu beauftragen. Passiert dies z. B. im Rahmen von Abschlussarbeiten, hat man oft engagierte Studierende, die aber nicht den Überblick über die Möglichkeiten haben und bei Entscheidungen die Kriterien für den richtigen Weg nicht unbedingt korrekt bewerten können. Dies ist durch eine erfahrene Betreuung auszugleichen, da ansonsten die resultierende Qualität des Ergebnisses eher ein Zufallsprodukt ist. Ein Beispiel ist die Nutzung des Oberflächentestwerkzeugs für webbasierte Software Selenium [@Sel], das zwei Varianten anbietet. In der ersten Variante werden Tests bei der Ausführung aufgezeichnet und können während und nach der Aufzeichnung um Zusicherungen erweitert werden. Die zweite Variante ermöglicht die vollständige Testerstellung direkt in einer Programmiersprache, in der zunächst die passenden Ansteuerungsbefehle für die Oberflächen entwickelt werden müssen. Während der erste Ansatz für schnelle Tests, die typischerweise nicht für Regressionstests genutzt werden, sehr gut geeignet ist, passt der zweite anfänglich aufwändigere Ansatz deutlich besser zur Entwicklung wartbarer Tests basierend auf einer klaren Testarchitektur. Es ist deshalb fragwürdig, nach Konzepten zu suchen, mit denen die Wartbarkeit beim ersten Ansatz leicht verbessert wird.

4 Zusammenfassung und Ausblick

Der Weg zu langfristig wart- und erweiterbarer Software besteht aus der kontinuierlichen Anpassung der Entwicklungsprozesse an neue Projektgegebenheiten, wie die Projektgröße und Projektvarianten. Ein zentraler Prozess des LotSE ist dabei die Qualitätssicherung. Die Automatisierung wesentlicher Schritte der Qualitätssicherung garantiert dabei die Testbarkeit des Systems, ist aber nur ein Teil des Ansatzes. Weitere, hier nicht im Detail betrachtete Bestandteile, umfassen den komponentenbasierten Software-Aufbau,

die hohe Qualifikation der Mitarbeiter und die kontinuierliche Suche nach Prozessverbesserungsmöglichkeiten.

Die in der Fallstudie beschriebene Prozessoptimierung zeigt ein erfolgreiches Beispiel dafür, wie durch Verbesserungen eines aktuell gelebten Entwicklungsprozesses eine Effizienzsteigerung in einem Software-herstellenden KMU durch Automatisierung von (GUI-)Tests erreicht wird. Beachtet werden muss, dass der Weg von der Idee zur Automatisierung bis hin zur Umsetzung nicht trivial ist, genau geplant werden muss und es durchaus Faktoren geben kann, die zur Nichtumsetzung führen können. Der beschriebene Automatisierungsansatz wird weiterentwickelt.

Bei den QS-Prozessen und QS-Werkzeugen, aber auch in anderen Bereichen, muss man über Innovationen und ihre mögliche Integration in den bisherigen Entwicklungsweg nachdenken. Zu beachten ist, dass ein einfacher Kriterienkatalog bzgl. der geforderten Funktionalität zu restriktiv sein kann. Man muss zunächst die Möglichkeiten analysieren, welche Werkzeuge anbieten, die aber im aktuellen Entwicklungsprozess noch keine Rolle gespielt haben. Diese Innovationsmöglichkeit ist in der Abbildung 6 zusammengefasst, die von der Ausgangssituation auf der linken Seite nach der Vorauswahl geeigneter Werkzeuge (W) den Prozess zur Evaluation der Werkzeugmöglichkeiten und die resultierende neue Entwicklungsumgebung auf der rechten Seite zeigt.

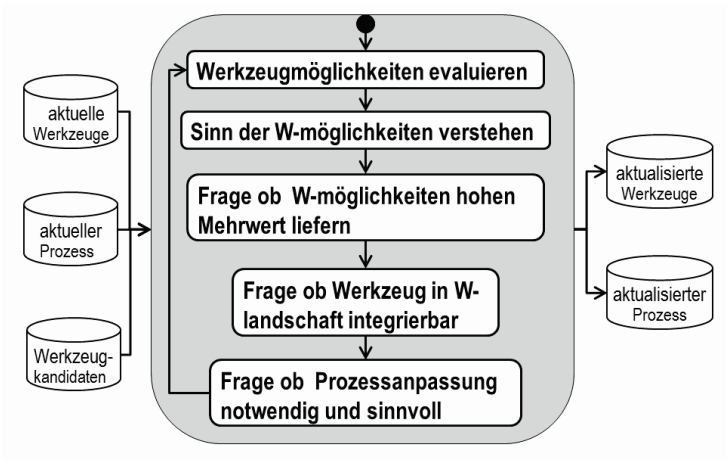

Abbildung 6: Evaluation innovativer QS-Werkzeuge

Generell ist die QS ein Prozess, der durch neue Technologien auch im Bereich des Software-Engineering sich in ständiger Entwicklung befindet und nur durch hochqualifizierte Mitarbeiter langfristig bei einer sich weiterentwickelnden Software effizient nutzbar bleiben kann.

Literaturverzeichnis[15]

[@CDI] JSR 346: Contexts and Dependency Injection for Java EE 1.1, https://jcp.org/en/jsr/detail?id=346

[@Fes] fest - Fixtures for Easy Software Testing, https://code.google.com/p/fest/

[@Mic] Microsoft Developer Network., Isolating Code under Test with Microsoft Fakes, http://msdn.microsoft.com/de-de/library/hh549175.aspx

[@Moc] mockito – simpler & better mocking, https://code.google.com/p/mockito/

[@RES] JSR 339: JAX-RS 2.0: The Java API for RESTful Web Services, https://jcp.org/en/jsr/detail?id=339

[@Sel] Selenium – Web Browser Automation, http://www.seleniumhq.org/

[@Sic] sikuli-api - Sikuli Java API, https://code.google.com/p/sikuli-api/

[@Web] JSR 356: Java API for WebSocket, https://jcp.org/en/jsr/detail?id=356

[Bur14] B. Burke, RESTful Java with JAX-RS 2.0, O'Reilly, Sebastopol (CA), USA, 2014

[Dit14] A. Ditler, Prototypische Realisierung eines Echtzeit-Webchats als Crossplattform-Applikation auf Basis von WebSockets, Bachelorarbeit, Hochschule Osnabrück, 2014

[Hei12] A. Heidt, Konzeption und Realisierung einer automatisierten Testumgebung in einem Continuous Integration Prozess für admileo, BA-Arbeit, Hochschule Osnabrück, 2012

[HF10] J. Humble, D. Farley, Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, 1, Addison-Wesley Longman, Amsterdam, 2010

[HK12] A. Heidt, S. Kleuker, Kontinuierliche Prozessverbesserung durch Testautomatisierung. In: H. Brandt-Pook, A. Fleer, T. Spitta, M. Wattenberg, Nachhaltiges Software Management, Bielefeld, 2012, GI-Edition - LNI, P-209. Seiten 69-78, Köllen, Bonn 2012

[Hub10] T. C. Huber, Windows Presentation Foundation – Das umfassende Handbuch, Galileo Computing, Bonn 2010

[Kle13] S. Kleuker, Qualitätssicherung durch Softwaretests, Springer Vieweg, Wiesbaden, 2013

[Klo13] G. Klompmaker, Erweiterung des Prototyps einer mobilen Android-Portalanwendung mit anschließender Ermittlung und Gegenüberstellung von Performance-Kennzahlen REST- und SOAP-basierter Systeme, BA-Arbeit, HS Osnabrück, 2013

[Lad03] R. Laddad, Aspectj in Action: Practical Aspect-Oriented Programming, Manning Publications, USA, 2013

[SW01] H. M. Sneed, M. Winter, Testen objektorientierter Software, Hanser, München et al 2001

[SL12] A. Spillner, T.Linz, Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester - Foundation Level nach ISTQB-Standard, dpunkt, Heidelberg 2012

[Spi14] A. Spillner, Warum White-Box-Test kein Test ist, in: GI Softwaretechnik-Trends, Band 33, Heft 4, 2014.

[SRW14] A. Spillner, T. Roßner, M.Winter, Praxiswissen Softwaretest - Testmanagement (iSQI-Reihe): Aus- und Weiterbildung zum Certified Tester - Advanced Level nach ISTQB-Standard, dpunkt, Heidelberg 2014

[Wit13] V. Witt, Evaluation von View-Technologien im Java EE-Umfeld zur prototypischen Migration eines Qualitätssicherungssystems der Lebensmittelbranche, MA-Arbeit, HS Osnabrück 2013.

[15] Letzte Aufrufe der genannten Web-Seiten stammen vom 14.12.2014.