Kontinuierliche Prozessverbesserung durch Testautomatisierung

Andre Heidt, Stephan Kleuker

Archimedon Software GmbH
Marienstr. 66
D-32427 Minden
Andre.Heidt@archimedon.de

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

Abstract: Für kleine und mittelständische Unternehmen (KMU), die Software-Produkte herstellen, stellt sich kontinuierlich die Herausforderung, die Software wart- und erweiterbar zu halten. Dabei müssen die Maßnahmen meist im laufenden Betrieb eingeführt werden und der Return of Investment (ROI) möglichst garantiert sein. Dieser Bericht beschreibt, mit welchen Schritten eine Testautomatisierung zu einer kontinuierlichen Softwareentwicklungsprozessverbesserung bei einem KMU geführt hat. Weiterhin wird andiskutiert, welche Rahmenbedingungen analysiert werden müssen, um die unternehmensindividuell passendsten Maßnahmen zur Prozessverbesserung zu bestimmen, die auch unmittelbaren Einfluss auf die Nachhaltigkeit der Produkte haben.

1 Einleitung

Die typische Historie eines KMU im Bereich Software-Entwicklung besteht meist darin, dass Experten eines Anwendungsbereichs feststellen, dass sie eine Software suchen, die auf ihre individuellen Anforderungen zugeschnitten ist, und diese nicht finden. Dies führt entweder unmittelbar zur Neugründung einer Firma oder eine Software wird in einer Firma des Anwendungsbereichs entwickelt, dann festgestellt, dass diese auch für andere Unternehmen nutzbar sein kann, und dann eine Ausgründung einer Entwicklungsfirma durchgeführt. Für den Erfolg des KMU ist es 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 ersten Markterfolgen werden Schritt für Schritt neue eigene Ideen und individuelle Kundenwünsche in die Software eingebaut. Dabei muss ein oftmals kleines, eng zusammenarbeitendes Team von Entwicklern dann personell ergänzt werden. Irgendwann

muss dann der 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 bleibt die Frage, welcher Teilprozess zunächst verbessert werden sollte, nicht einfach beantwortbar. Falsche Entscheidungen können dabei leicht die Existenz eines KMU gefährden, da das tägliche Geschäft kontinuierlich weiterlaufen muss.

Dieser Bericht zeigt ein Beispiel, wie auf Grundlage einer Analyse eine erfolgreiche Verbesserung des Software-Entwicklungsprozesses im Bereich der Testautomatisierung mit Schwerpunkt auf dem GUI-Test durchgeführt wurde. Zunächst werden im folgenden Kapitel einige Hintergründe zu den Herausforderungen von KMUs zusammengefasst, dann die Ausgangssituation im Unternehmen beschrieben und dann die konkrete Umsetzung der Prozessverbesserung vorgestellt. Abschließend wird eine Analyse skizziert, wann der genutzte Weg auch für andere KMUs interessant ist. Eine effiziente Systementwicklung hat als wichtigen Nebeneffekt unmittelbaren Einfluss auf die Ressourcen, die bei der Produkterstellung benötigt werden.

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

2 Hintergrund

Software-Engineering [Ba00][Wi05][LL06][Kl11] hat sich als wissenschaftliche Ingenieurdisziplin schrittweise in der Informatik beginnend mit der Erkenntnis etabliert, dass die Entwicklung von Software für komplexere Systeme systematisch organisiert werden muss, um die Erfüllung von Qualitätsansprüchen sowie die Wart- und Erweiterbarkeit zu ermöglichen. Das Software-Engineering zerfällt in einige Teilprozesse, die Hand in Hand arbeiten können. Statt eines detaillierten Ausblicks werden hier einige dieser Prozesse mit Beispielen aus der Praxis aufgezählt, wie eine schwächere Umsetzung in einem KMU zu Problemen führte. Die Beispiele stammen aus den Tätigkeiten des zweiten Autors als Berater und Betreuer von Abschlussarbeiten.

Anforderungsanalyse: Durch die steigende Komplexität des Systems verstanden die Entwickler nicht mehr, was wozu in die Software eingebaut wurde. Bei unklaren Aufgabenstellungen wurden Annahmen getroffen, die in der Praxis nicht passten.

Design und Architektur: Durch eine unsaubere Trennung der Datenhaltung von der Business-Schicht und mangelnde Aufteilung in Software-Komponenten musste für jede Änderung das gesamte Programm geändert werden, was mangels Kenntnis des Gesamtsystems auch zu Doppelentwicklungen führte.

Implementierung: Dadurch, dass Entwicklern zu viele Freiheiten in der Codierung und der Umsetzung gegeben wurden, war der Code für andere Entwickler unleserlich und wurde ein konzipiertes Mehrschichtensystem nicht eingehalten.

Test: Dadurch, dass alle Tests manuell durchgeführt werden mussten, stieg mit wachsender Systemgröße die benötigte Zeit für das teilweise auch unsystematische Testen enorm an, wobei sich trotzdem die Fehlerquote erhöhte.

Konfigurationsmanagement: Dadurch, dass für verschiedene Kunden Varianten der gleichen Software entwickelt wurden, die nicht miteinander integriert wurden, musste jede Variante sehr aufwändig individuell gewartet und gepflegt werden, so dass es sich von außen um eigenständige Softwaresysteme handelte.

Aufwandsschätzung: Dadurch dass es keinen funktionierenden Prozess gab, in dem von Entwicklern angegebene, konsolidierte Aufwände in die Schätzung eingingen, Endzeiten aber eingehalten werden mussten, sank die Qualität durch nicht durchgeführte Tests.

Risikomanagement: Durch ein wachsendes Unternehmen wurde die Distanz zwischen Entwicklern und der Geschäftsleitung immer größer, so dass den Entwicklern bekannte potenzielle Problemquellen nicht an die Leitung kommuniziert wurden.

Die Beispiele zeigen deutlich, dass zwar alle Prozesse auch in KMUs sinnvoll umgesetzt sein müssen, es aber häufig einzelne oder kleine Gruppen von Prozessen gibt, bei denen entweder schon unmittelbarer Handlungsbedarf besteht oder zumindest das größte Optimierungspotenzial mit den größten mittelfristig zu erwartenden ROI liegt, der besonders durch die benötigte Arbeitszeit von Mitarbeitern beeinflusst wird.

3 Prozessanalyse

Das Projekt wurde in der Firma Archimedon durchgeführt. Hierbei wurde das Ziel verfolgt, den Qualitätssicherungsprozess für die Project and Business Software „admileo“ zu verbessern. admileo ist das Hauptprodukt der Archimedon Software + Consulting GmbH & Co. KG [@arc], einem KMU mit derzeit 16 Mitarbeitern, das auf die Entwicklung von Java Software spezialisiert ist. Die Firma mit Hauptsitz in Minden (NRW) wurde im Jahr 2004 gegründet und 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 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 hohe Wartbarkeit und Wiederverwendbarkeit 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. Dank dieser firmeninternen Zielsetzung ist die Idee für die Einführung der Testautomatisierung, die im Folgenden ausführlicher beschrieben wird, sowohl bei der Geschäftsführung als auch bei den Entwicklern kooperativ begrüßt worden.

Das Projekt zur Verbesserung der Qualitätssicherung ist in erster Linie darauf ausgerichtet, die Project and Business Software admileo, das Hauptprodukt mit den Bereichen Verkauf, Beratung und Anpassung bei der Einführung sowie der optionalen Möglichkeit die Software zentral zu hosten, der Firma Archimedon, zu testen. Alle Ergebnisse und Erfahrungen sollen aber auch in weitere, zukünftige Projekte von Anfang an mit einflie-

ßen. Bei admileo handelt es sich um ein modular aufgebautes Client-Server-System mit Datenbankanbindung, welches fast ausnahmelos in Java implementiert ist. Der Server bildet die Schnittstelle zur Datenbank und übernimmt die Benutzerverwaltung und Steuerung der Kommunikation mit den Clients. Er kann auf diese Weise zum Beispiel die Aktualisierung der Clients veranlassen. Die Benutzeroberfläche von admileo basiert auf dem Swing Framework und verwendet auch 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.

Zur Zeit des Projekts bestand admileo aus 3 Haupt- und 46 erweiternden Modulen und besaß einen Umfang von weit über 1,5 Millionen Lines of Code (LoC). Das mit steigender Tendenz, da die Weiterentwicklung und Verbesserung von admileo ein ständiges Ziel der Firma darstellt. Ein erster Rückblick auf die Art und Weise der vorangegangenen Softwareentwicklung hat verdeutlicht, dass das modulare Konzept eine Weiterentwicklung des Systems generell zwar wesentlich vereinfacht hat, der Prozess zum Testen der Software für große Releases jedoch stark angestiegen ist. Wegen dieser Steigerung, ausgehend von wenigen Personenmonaten auf über das Doppelte, wurde der Bereich „Test“ als Optimierungsfeld identifiziert.

Hierbei ist eine Anforderung an die zu entwickelnde Lösung ihre Integrierbarkeit in die bisherigen Entwicklungsprozesse gewesen. Um zu beurteilen, welche weiteren Schritte dabei sinnvoll sind, wurde der bereits etablierte Entwicklungs- und Testprozess analysiert. Dieser Entwicklungsprozess von admileo ist angelehnt an das V-Modell, mit klar definierten Entwicklungs- und Testphasen. In den folgenden konkreten Testphasen wird vollständig manuell getestet:

- Fortlaufender Test in der Entwicklungsumgebung
- Kontinuierlicher Integrationstest im Testsystem
- Betatest der Module auf Testsystemen
- Qualitätssicherungstest
- Abnahmetest

Das Bauen von admileo und die Integration zu Gesamtsystemen für Tests und Releases findet mit Hilfe von Build-Tools wie Ant [@ant] und dem Continuous Integration (CI)-Server Jenkins [@jen] vollständig automatisiert statt. Der Release-Zyklus von admileo weist zwei Versionen pro Jahr auf. Hierbei gehen jedem Release Phasen des intensiven manuellen Testens aller Bestandteile sowie des Gesamtsystems voraus.

Als Resultat der Analyse ist die Idee entstanden, durch eine Testautomatisierung oft wiederholte Testvorgänge abzudecken. Die Abläufe sollen dabei stets identisch sein und somit zwischen den einzelnen Versionen konkret vergleichbare Ergebnisse liefern. Hierdurch soll die Regression der Qualität des Codes verhindert werden. Darüber hinaus eignet sich die unbeaufsichtigte Durchführbarkeit von automatisierten Tests dazu, die Entwickler von monotonen, wiederkehrenden Testvorgängen zu entlasten und ihnen mehr Zeit für anspruchsvollere Tests sowie den Ausbau und die Pflege der Testfallsätze zu verschaffen. Automatisierte Tests können zudem regelmäßig ausgeführt werden und Fehler möglichst zeitnah zu ihrer Entstehung aufzeigen.

Das kontinuierliche, automatisierte Bauen und Integrieren von admileo mit Hilfe von Ant und Jenkins lässt es zu, diese Testvorgänge in das System zu integrieren. Es soll somit ein CI-System [Hu09] entstehen, das kontinuierlich und automatisiert bauen und testen kann. Folgende Testmethoden wurden für solch ein System im Rahmen einer Teststrategie [Li09] festgelegt:

- Unit Tests und Integrationstests
- Funktionstests über die Benutzeroberfläche (GUI-Tests)
- Messung der Codeüberdeckung (Code Coverage)
- Statische Quellcodeanalysen

Um nach einem Einstieg in die Testautomatisierung mitten im Entwicklungszyklus von admileo möglichst schnell umfangreiche Teile erfolgreich testen zu können, wurde beschlossen, das Hauptaugenmerk auf die GUI-Tests zu legen, wodurch eine hohe Systemabdeckung mit den Tests erzielt werden kann. Die Multiprojektmanagement-Software admileo ist ein typisches Beispiel für eine Software, deren Funktionalität meist unmittelbar von einem GUI aus zugänglich ist. Durch die Verwendung der Überdeckungsmessung sowohl bei den Unit Tests als auch bei den GUI-Tests, können sich beide Testverfahren ergänzen. Codestellen mit geringer Codeüberdeckung können so durch die Ergänzung sinnvoller Tests aus einer der beiden Testmethoden abgedeckt werden.

Für die Umsetzung der Testmethoden mussten passende Werkzeuge ausgewählt werden, die in ein CI-System integrierbar sind. Ausgewählt wurde sowohl aus dem kommerziellen Bereich, als auch aus Open Source Werkzeugen.

4 Werkzeugauswahl und Umsetzung des Prozesses

Für das CI-System wurden Testwerkzeuge ausgewählt, die sowohl die Anforderungen erfüllen sollen, die durch die Teststrategie vorgegeben sind, als auch die Integrierbarkeit in bestehende Prozesse und Systeme gewährleisten. Die Verwendung weiterer Werkzeuge, um diese Testwerkzeuge zu einem automatisierten Gesamtsystem zu integrieren und die Benutzbarkeit und Wartung zu erleichtern, wurde während prototypischer Umsetzungen zu Evaluationszwecken nach Bedarf beschlossen.

Folgende Testwerkzeuge wurden nach der Analyse für das CI-System ausgewählt:

- QF-Test [@qfs] für GUI-Tests
- Sikuli X [@sik] als Ergänzung zu QF-Test
- JUnit [@jun] für Unit Tests und Integrationstests
- JaCoCo [@jac] zur Überdeckungsmessung
- Sonar [@son] mit den Checkstyle und PMD Plugins für Quellcodeanalyse, Softwaremetriken und die Verwaltung der Testergebnisse

Für die Automatisierung des Bau- und Testprozesses kommen der CI-Server Jenkins und das Build-Werkzeug Ant zum Einsatz.

Weitere Werkzeuge in unterstützenden Rollen sind:

- Hyper-V [@hyp] zur Virtualisierung von Testsystemen
- Jython als Skriptsprache

- Apache Tomcat [@tom] als Plattform für Sonar und Jenkins
- PostgreSQL [@pos] als Datenbank
- Subversion (SVN) [@svn] zur Versionsverwaltung

Das System soll als Benutzerschnittstellen die Weboberflächen von Jenkins und Sonar zur Verfügung stellen, wobei Änderungen an dem Bau- und Testprozess sowie manuelle Builds in Jenkins vorgenommen werden können. Sonar hingegen bietet neben einer umfangreichen Übersicht über den Projektstand anhand eines „Cockpits“ mit konfigurierbaren Metriken und Trends auch die Testergebnisse und stilistische Auswertungen des Quellcodes. Das CI-System soll also, gesteuert von Jenkins, möglichst jedes neue Build einer Reihe von automatisierten Tests unterziehen und die Ergebnisse von Sonar auswerten lassen, um sie darauf folgend in einer Datenbank zu persistieren. Testfälle für admileo werden in einem SVN-Repository verwaltet.

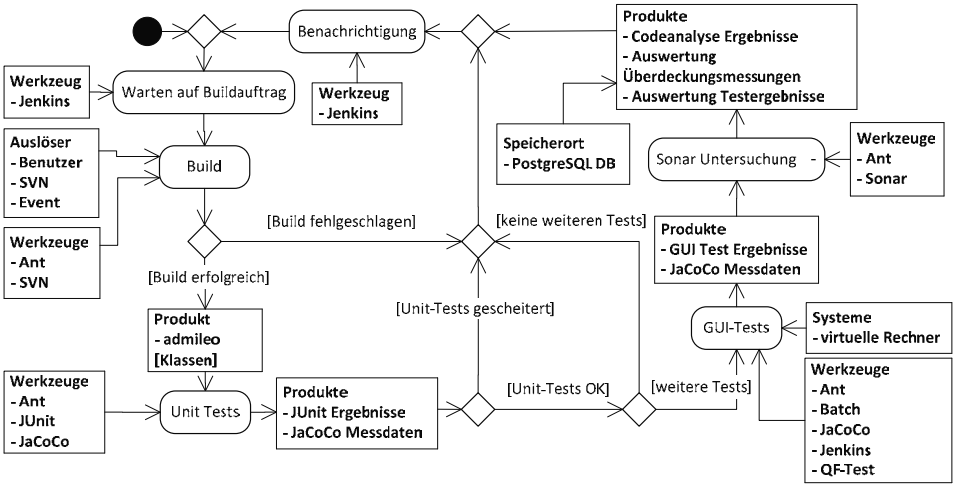

Abbildung 4: Build-Prozess mit integrierter Qualitätssicherung

Der bisherige Ablauf eines Builds (ohne Tests) bestand aus dem Bauen von admileo, gefolgt von der Integration zu unterschiedlichen Testsystemen. Dieser Ablauf wurde nun dahingehend erweitert, dass unmittelbar nach dem Bauen Tests mit möglichst kurzer Ausführungszeit durchgeführt werden. Daraus resultiert der Vorteil, dass Entwickler eine schnelle Rückmeldung erhalten, falls in diesem Schritt bereits Fehler entdeckt wurden. Weiteres, zeitaufwändiges Testen wäre in diesem Fall nämlich ineffizient, bevor die Fehler behoben sind. Im ersten Schritt sollen zu diesen „schnellen“ Tests alle Unit Tests zählen, bis deren Anzahl und ihre Laufzeit so weit angewachsen sind, dass eine weitere Unterteilung sinnvoll wird. Diesen ersten schnellen Tests folgen schließlich weitere Tests, wozu vor Allem die GUI-Tests gehören. Vorausgesetzt in den vorhergehenden Tests wurden keine kritischen Fehler gefunden, wird der Code abschließend einer statischen Codeanalyse unterzogen. Hierzu kommen diverse Plugins, unter Sonar, darunter Checkstyle und PMD, zum Einsatz. Alle bis zu diesem Zeitpunkt akkumulierten Testergebnisse fließen in die Berechnungen der Softwaremetriken ein. Der weitere Ablauf kann nach erfolgreichem Bestehen der Tests wie bisher stattfinden. Der Gesamtablauf ist auch in Abb. 1 dargestellt, Verfeinerungen der Aktionen stehen in [He12].

Die technische Umsetzung des CI-Systems besteht hauptsächlich aus der Installation und der Konfiguration der Werkzeuge in der Einsatzumgebung, der Automatisierung aller einzelnen Teilschritte im Prozessablauf und der Integration der einzelnen Komponenten zum Gesamtsystem.

Bei der Installation musste geplant werden, wie die Verteilung des Systems auf die vorhandene Hardwarelandschaft des Unternehmens sinnvoll umzusetzen wäre. Jenkins und Sonar wurden auf einem Tomcat Applikationsserver installiert, anstatt ihre eingebetteten Webcontainer zu verwenden oder sie als Dienste zu installieren. Dies geschah, damit sie Betriebssystem-unabhängig und zentral verwaltet werden können. GUI-Tests sollen mit virtuellen Testrechnern auf einem extra dafür vorgesehenen Server durchgeführt werden, wobei die Virtualisierung als Nebeneffekt die Energiekosten senkt. Somit können auch gleich mehrere gleichkonfigurierte Rechner einfach angelegt werden, um beispielsweise Testfälle parallel durchzuführen. Zum Austausch von Daten und gemeinsam genutzter Software und Bibliotheken wird ein Fileserver verwendet.

Ein Konfigurationsaufwand ist hauptsächlich bei Jenkins und Sonar angefallen. In Jenkins mussten die entsprechenden Projekte für die einzelnen Testschritte angelegt werden. Hierbei war einige Einarbeitungszeit notwendig, um die Funktionsweisen und die möglichen Projektstrukturen kennenzulernen. Es hat sich als ein Problem herausgestellt, das gewünschte Auslösen eines neuen Builds bei Änderungen in SVN-Repositories umzusetzen, da admileo für jedes Modul ein eigenes Repository verwendet. Jenkins kann zwar in aktuellen Versionen mit mehreren Repositories gleichzeitig umgehen, jedoch überprüft es diese durch Polling, was bei ca. 50 Repositories für keine attraktive Lösung befunden wurde. Eine mögliche Alternative ist das Verwenden von „Hook-Skripten“, die nach dem Commit auf dem SVN-Server ausgeführt werden und einen neuen Build anstoßen. Bei Sonar musste eine sinnvolle Kombination aus den verfügbaren Überprüfungsregeln und Metriken gefunden werden, die den Konventionen und Erwartungen der Entwickler und Geschäftsführer entspricht. Darüber hinaus musste Sonar die Projektstruktur von admileo beigebracht werden. Dies ist grundsätzlich über Kommandozeilenparameter, Ant oder Maven, machbar. Allerdings unterstützen nur die beiden letzteren die Multi-Modul Projekte dahingehend, dass sie entsprechend der Paketstruktur navigierbar sind, anstatt alle Klassen in einer flachen Struktur abzubilden. Ein Problem dabei ist jedoch, dass für jedes Modul und Paket, das gesondert auswertbar sein soll, ein eigenes Ant-Skript angelegt werden muss, in dem ein eindeutiger Schlüssel und Pfadangaben zu Quellcodes und Klassen vorhanden sind. Bei einem großen Projekt wie admileo würde dies einiges an Aufwand bedeuten, weshalb ein Jython Skript erstellt wurde, das die Projektstruktur automatisiert auswertet und passende Ant-Skripte generiert.

Die Abhängigkeiten zwischen einzelnen Schritten im Prozess sind grundsätzlich nicht sonderlich hoch, von der Voraussetzung abgesehen, dass der Build eine lauffähige Version von admileo liefern muss. Ansonsten sollten die Testergebnisse alle in einer von Sonar unterstützten Formatierung bereitliegen, so dass sie alle ausgewertet werden können. Dies gestaltete sich weitestgehend problemlos, da Sonar JUnit-formatierte Ergebnisse sowie JaCoCo Messdaten unterstützt und QF-Test auf Wunsch seine Ergebnisse auch im JUnit-Format generieren kann.

Bei der Evaluation von QF-Test ist festgestellt worden, dass das Werkzeug mit einigen Situationen wie „Drag and Drop“ nicht ohne weiteres zurechtkommt. Elemente von admileo, die nicht in Java implementiert sind wie beispielsweise der Update- und Launch-Mechanismus, kann es nicht bedienen. Dies war auch bei anderen Werkzeugen der Fall, die nach einem ähnlichen Prinzip arbeiten. Zur Abhilfe wurde auf den Testrechnern das zusätzliche, technologieunabhängige Werkzeug Sikuli eingesetzt, das anhand von Bildverarbeitung GUI-Elemente in Screenshots wiedererkennt und bedient. Der Nachteil an diesem Werkzeug ist, dass es trotz einstellbarer Toleranzgrenzen bei der Erkennung stark vom gleichbleibenden Aussehen der GUI abhängig ist. Aus diesem Grund wurde beschlossen, es möglichst sparsam einzusetzen, um ein häufiges Anpassen der Testfälle bei Änderungen an admileo zu vermeiden. Damit die Testergebnisse einheitlich und zusammen mit den anderen Testergebnissen verwendbar bleiben, wird Sikuli aus QF-Test Testfällen heraus gestartet. Dies ist problemlos möglich, da QF-Test Kommandozeilenbefehle und Skripte ausführen und anhand ihrer Rückgabewerte den Ausgang der Sikuli Tests auswerten kann.

Nach der Evaluation der prototypischen Implementierung des beschriebenen CI-Systems wurde eine Umsetzung für den produktiven Einsatz beschlossen. In die Prozessverbesserung sind Ressourcen eingeflossen. Ein ROI wird nach vier Testphasen (Bei 2 Releases pro Jahr: 2 Jahre) erwartet, eine Anzahl, die im typischen Rahmen der häufiger genannten drei bis fünf benötigten Wiederholungen liegt. Ein wichtigeres Ziel für Archimedon als das direkte ROI ist jedoch die Verbesserung der Softwarequalität. Es muss positiv bewertet werden, dass jetzt häufiger getestet werden kann. Kritisch ist zu beachten, dass Änderungen an der GUI mehr Nacharbeiten in den Testfällen benötigen, als früher.

5 Analyse der Erfolgsfaktoren

Naiv betrachtet, kann man aus den vorherigen Kapiteln ableiten, dass man alle KMUs zur Testautomatisierung als offensichtlich schnell ROI bringenden Ansatz auffordern muss. Kritischer betrachtet, handelt es sich sicherlich um eine Maßnahme mit großem Potenzial, allerdings trafen in dieser Fallstudie mehrere Randbedingungen zu, die den Erfolg gesichert haben, die kurz skizziert wie folgt aussehen.

- klar formulierte Anforderungsspezifikation: Durch strukturiert erfasste Anforderungen wird es überhaupt erst möglich, zu überprüfen, ob das gewünschte System entwickelt wird.
- wiederholbar spezifizierte Systemtests: Durch die strukturierte textuelle Ausformulierung von Systemtests mit den Vorbedingungen, der Ausführung und den erwarteten Ergebnissen wurde eine gute Grundlage zur Automatisierung geschaffen.
- Client-Server-System: Durch die klassische Architektur kann diese unmittelbar auch als Testarchitektur umgesetzt werden, wodurch realistische Testszenarien ermöglicht werden und das Zusammenspiel zwischen Client und Server analysierbar wird.
- modular aufgebaute Software mit meist unveränderten GUI-Anteilen: Durch die modulare Software-Architektur ist gewährleistet, dass neue Komponenten typischerweise nicht zu vielen kleinen Änderungen an unterschiedlichsten Stellen führen, die den Testaufwand deutlich erhöhen.

- Erfahrung in automatisierten Prozessen: Durch die Nutzung automatisierter Build-Prozesse, z. B. mit den Werkzeugen Ant und Jenkins, ist die Ergänzung dieser Prozesse ein kleinerer Schritt, als zunächst ein Build Management einzuführen und zu ergänzen.
- nutzbare Werkzeuge zur Testautomatisierung: Durch die detaillierte Analyse konnten Werkzeuge gefunden werden, die eine Testautomatisierung der GUI-Tests überhaupt erst ermöglichen. Der Einsatz von Swing trug dazu bei, dass es überhaupt passende Werkzeuge gibt.
- akademisch qualifizierte Software-Entwickler: Die systematische Ausbildung der Entwickler und ihre bisherigen Erfahrungen ermöglichten es, dass eine Akzeptanz für schrittweise Prozessänderungen vorliegt.

Man kann sicherlich folgern, dass unter exakt gleichen Randbedingungen die Automatisierung der GUI-Tests eine höchstwahrscheinlich gewinnbringende Maßnahme ist. Allerdings sind die Software-Entwicklungsprozesse und die entstehenden Produkte bei genauer Analyse so unterschiedlich, dass es unwahrscheinlich ist, eine identische Situation vorzufinden. Die Analyse der genannten Randbedingungen kann aber wesentlich bei der Auswahl der zuerst umzusetzenden Optimierungsmaßnahme helfen.

6 Zusammenfassung und Ausblick

Die vorherigen Kapitel zeigen ein erfolgreiches Beispiel, wie durch Verbesserungen eines aktuell gelebten Entwicklungsprozesses eine Effizienzsteigerung in einem Software-herstellenden KMU durch Automatisierung von GUI-Tests erreicht wird. Weiterhin wird deutlich, 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 der Idee führen können. Weiterhin soll der beschriebene Automatisierungsansatz noch weiter entwickelt werden, so kann durch eine Parallelisierung die Ausführungszeit reduziert, können Testergebnisse noch weiter aufbereitet und kann eine visuelle Komponente zur Auswahl bestimmter Teiltestgruppen umgesetzt werden.

Nicht betrachtet wurde aus Platzgründen eine Diskussion über notwendige Ausbildungsmaßnahmen von Mitarbeitern in KMUs. Dies umschließt die Frage, wann ein Selbststudium ausreicht und wann Schulungsmaßnahmen oder die temporäre Begleitung der Entwicklung durch einen Coach sinnvoll sind. Werden z. B. Mitarbeiter neu in die Qualitätssicherung eingearbeitet, ist eine ISTQB-Schulung [@ist] unterstützt durch Literatur [SL10] und evtl. durch Coaching sehr zu empfehlen.

Neben der offenen Frage nach der Ausbildung besteht eine wichtige, hier andiskutierte, Aufgabe darin, die richtigen Maßnahmen in KMUs zu treffen, um ihnen einen langfristigen Erfolg zu ermöglichen. Im vorherigen Kapitel wurden erste Randbedingungen andiskutiert, die bei der Auswahl geeigneter Maßnahmen beachtet werden müssen. Dabei stellt sich die offene Frage, welche Randbedingungen zu welcher Maßnahme führen sollen. Natürlich ist die Liste aus dem vorherigen Kapitel nicht vollständig; viele weitere Randbedingungen, wie das Potenzial und die Fähigkeiten der Konkurrenten und die besonderen Erwartungen von Kernkunden, können Einfluss nehmen. Auch die Anzahl potenziell interessanter Maßnahmen geht weit über die bisher andiskutierten hinaus und umfasst z. B. präzisierte Aufwandsschätz- und Risikomanagement-Prozesse, beinhaltet

aber auch Ansätze zur modellgetriebenen Entwicklung oder auch den Einsatz formaler Methoden [Kl09]. Dieses Netzwerk von Randbedingungen und Maßnahmen soll an der Hochschule Osnabrück im Rahmen eines Forschungsschwerpunkts „Long Term Software-Engineering“ für KMU entschlüsselt werden.

Vereinfacht kann man für KMU folgern, dass nur durch einen hochwertigen, auf das Unternehmen passenden Entwicklungsprozess langlebige Software entstehen kann, die schon während ihrer Erstellung und Nutzung Aspekte der Nachhaltigkeit berücksichtigt.

Literaturverzeichnis

Letzte Aufrufe der genannten Web-Seiten stammen vom 6.8.2012.

[@ant] Apache Ant, http://ant.apache.org/
[@arc] Archimedon, http://www.archimedon.de/
[@hyp] Hyper-V, http://www.microsoft.com/de-de/server/hyper-v-server/default.aspx
[@ist] ISTQB International Software Testing Qualifications Board, http://www.istqb.org/
[@jac] JaCoCo Java Code Coverage Library, http://www.eclemma.org/jacoco/
[@jen] Jenkins, http://jenkins-ci.org/
[@jun] JUnit, http://kentbeck.github.com/junit/javadoc/latest/
[@pos] PostgreSQL, http://www.postgresql.org/
[@qfs] Quality First Software, http://www.qfs.de/de/qftest/index.html
[@son] Sonar, http://www.sonarsource.org/
[@sik] Project Sikuli, http://sikuli.org/
[@sub] Subversion, http://subversion.tigris.org/
[@tom] Apache Tomcat, http://tomcat.apache.org/
[@wq] Werkzeuge für die Qualitätssicherung, http://home.edvsz.hs-osnabrueck.de/skleuker/CSI/Werkzeuge/kombiQuWerkzeuge.html

[Ba00] Balzert, H.: Lehrbuch der Software-Technik: Software-Entwicklung, 2. Auflage, Spektrum Akademischer Verlag, Heidelberg Berlin Oxford, 2000

[He12] Heidt, A.: Konzeption und Realisierung einer automatisierten Testumgebung in einem Continuous Integration Prozess für admileo, Bachelorarbeit, HS Osnabrück, 2012

[Hu09] Humble, J.; Farley, D.: Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation, 1. Auflage, Addison-Wesley Longman, Amsterdam, 2010

[Kl09] Kleuker, S.: Formale Modelle der Softwareentwicklung, Vieweg+Teubner, Wiesbaden, 2009

[Kl11] Kleuker, S.: Grundkurs Software-Engineering mit UML, 2. Auflage, Vieweg+Teubner, Wiesbaden, 2011

[Li09] P. Liggesmeyer, Software-Qualität. Testen, Analysieren und Verifizieren von Software, 2. Auflage, Spektrum Akademischer Verlag, Heidelberg Berlin Oxford, 2009

[LL06] Ludewig, J.; Lichter, H.: Software Engineering – Grundlagen, Menschen, Prozesse, Techniken. dpunkt.verlag, Heidelberg, 2006

[SL10] Spillner, A.; Linz, T.: Basiswissen Softwaretest, 4. Auflage, Dpunkt Verlag, Heidelberg, 2010

[SW02] Sneed H. M.; Winter, M.: Testen objektorientierter Software, Hanser, München Wien, 2002

[Wi05] Winter, M.: Methodische objektorientierte Softwareentwicklung, Dpunkt Verlag, Heidelberg, 2005