WHITE PAPER | MAI 2016

Modellbasiertes Testing als Grundlage für verhaltensgetriebene Entwicklung

Huw Price
CA Technologies

ca technologies

Inhaltsverzeichnis

Abschnitt 1

Eindeutigkeit und Zusammenarbeit als treibende Prinzipien

Da Business und IT in ihrer Tätigkeit enger zusammenrücken, wird es immer wichtiger, die Anforderungen des Business so zu kommunizieren, dass sie direkt in technische Projekte übertragen werden können. Moderne Unternehmen greifen zunehmend auf Software zurück, die ihren Kunden Nutzen verschaffen kann, und so müssen IT-Teams vollständig getestete Software bereitstellen, die wechselnde Geschäftsanforderungen erfüllt, und das schneller und günstiger.

Allerdings verringern Unklarheiten bei den Softwareanforderungen die Wahrscheinlichkeit, dass IT-Teams verstehen, welche Software sich End User und Business-Analysten vorstellen. Das mindert wiederum die Wahrscheinlichkeit, dass die Software für das Unternehmen von Nutzen ist.

Diese Frustration bei der Entwicklung von Software, die zwar vollständig einer plausiblen Interpretation der Anforderungen entspricht, aber dennoch die Vorstellungen verfehlt, brachte Dan North dazu, sich für eine verhaltensgetriebene Entwicklung stark zu machen. Die ständigen „Verwirrungen und Missverständnisse“, mit denen er zu tun hatte, ließen ihn die Entwicklungsarbeit mit einer Reihe von Sackgassen vergleichen, in die er geführt wurde und in denen er meist mit dem Gedanken „Hätte mir das bloß jemand gesagt!“ zurückblieb.[1]

Angesichts der Herausforderungen bei typischen IT-Projekten ist diese Frustration unter Entwicklern und Testern offensichtlich verbreitet. Im Durchschnitt beginnen nur 4 % der Projekte mit eindeutigen Anforderungen[2], was wiederum Ursache für 56 % der Fehler ist[3]. Diese Fehler werden normalerweise erst in der Produktion entdeckt, wo ihre Behebung 50-mal länger dauert, als sie es bei einer Entdeckung in der Anforderungsphase getan hätte[4]. Die durch Unklarheiten bei den Anforderungen verursachte Nachbesserung macht 60 % der Softwareprojekte zum Fehlschlag oder teurer. Mangelhaft definierte Anforderungen sind für 82 % der Zeit verantwortlich, die insgesamt mit der Fehlerbehebung verbracht wird[5]. Dabei entstehen jährliche Kosten von 250 Milliarden US-Dollar[6]. North hatte daher guten Grund anzunehmen, dass es „möglich sein muss, die testgetriebene Entwicklung (Test-Driven Development, TDD) so zu präsentieren, dass die Anforderungen ohne Umwege erfüllt und die Fallstricke der Fehlkommunikation vermieden werden“.

Die verhaltensgetriebene Entwicklung (Behavior-Driven Development, BDD) beruht auf einer verstärkten Zusammenarbeit zwischen Business und IT, die durch den Begriff der aus dem Domain-Driven Design übernommenen „ubiquitären Sprache“ inspiriert ist. Da IT-Teams schnell auf wechselnde Unternehmensanforderungen reagieren müssen, ist dies ein angemessenes Konzept: eine gemeinsame Sprache, die sowohl ständige als auch gelegentliche Stakeholder verstehen (d. h. solche, die mit Unternehmenszielen und Anwendungsverhalten beschäftigt sind, und solche, die die gewünschten Ergebnisse und Verhaltensweisen implementieren) und die dank ihrer Semi-Formalität ohne Weiteres in die Logik eines zu entwickelnden und zu testenden Systems übersetzt werden kann.

BDD hängt dann davon ab, wie „verhaltensgetriebene Anforderungen“ akkurat zu formulieren sind, damit sie in der Entwicklung erfolgreich implementiert werden können. Genau darum geht es in diesem Dokument. Es wird untersucht, wie diese Grundprinzipien von mehr Zusammenarbeit und Eindeutigkeit zwischen Business und IT am besten im Entwicklungslebenszyklus implementiert werden können.

Unter Berücksichtigung des allgemein befürworteten Prozesses des Sammelns von Anforderungen wird erörtert, wie die Einführung von formalem Modeling synergetisch mit BDD zusammenwirken kann, sofern die richtigen Tools bereitgestellt werden. Die Auffassung, dass vollständige Dokumentation (im Vergleich zu „minimaler Dokumentation“) notwendigerweise bedeutet, dass auf Änderungen nicht schnell reagiert werden kann, wird infrage gestellt. Stattdessen wird gefolgert, dass formales Modeling zur Vollständigkeit von Anforderungen führt, was neben Eindeutigkeit nötig ist, um die Wahrscheinlichkeit zu erhöhen, dass die Software dem von den ständigen Stakeholdern gewünschten Verhalten entspricht.

Abschnitt 2

Unvollständigkeit

Wie aus den erwähnten treibenden Prinzipien von BDD hervorgeht, müssen Anforderungen nicht nur von Business und IT verstanden werden, sondern auch als technische Anforderungen für Tester und Entwickler implementierbar sein. Die Anforderungen werden aus den Interaktionen der ständigen Stakeholder abgeleitet, die tatsächlichen Nutzen aus einer Software ziehen und konkrete Beispiele für praktische Szenarien dessen liefern, wie die Anwendung sich verhalten soll. Dieses Vorgehen scheint auf dem Papier zu garantieren, dass die Anforderungen auch wirklich das vom Business gewünschte System widerspiegeln.

Doch obwohl diese „Von-außen-nach-innen"-Entwicklung Unklarheiten auszumerzen sucht, wirft der Charakter der Anforderungen und die Art ihrer Ableitung die Frage auf, wie gut sie bei Testing und Entwicklung von Systemen zu implementieren sind.

Wie Llyr Wyn Jones in „A Critique of Testing" ausführt, kann die Implementierung von Anforderungen als Prozess der Informationstransformationen betrachtet werden. Informationen werden in einer Form angenommen, um in einer anderen ausgegeben zu werden. So transformiert etwa ein Entwickler beim Programmieren Anforderungen in Software. Der gesamte Softwareentwicklungszyklus (Software Development Lifecycle, SDLC) kann als eine Art Informationsfluss betrachtet werden: Der Anwender erarbeitet mit einem Business-Analysten Anforderungen und Anwendungsfälle, der Entwickler schreibt den Code, und Tester erstellen schließlich manuell Testfälle und Test-Scripts. In jeder dieser Phasen werden Informationen weitergegeben und in eine neue Form „übersetzt".

Ausgehend von der Vorstellung von Anforderungen als aktive Informationen, definiert Llyr Wyn Jones Unklarheit mit Bezug auf das Konzept der „Unschärfe". Es entstehen, wie er beschreibt, verschiedene Informationsausgaben, da die Unschärfe der Bedeutung von Anweisungen dazu führt, dass eine Anforderung auf mehr als nur eine Weise implementiert werden kann. Ebenso kommt es zu Unvollständigkeit, wenn nötige Informationen über ein System der Spekulation überlassen werden[7]. Beides führt folglich zu „Unschärfe" und zur Wahrscheinlichkeit von Missverständnissen zwischen ständigen und gelegentlichen Stakeholdern.

Einfach ausgedrückt: Unvollständigkeit mindert wie auch Unklarheit die Wahrscheinlichkeit, dass eine Software sich verhält, wie es die ständigen Stakeholder beabsichtigt hatten. Unvollständigkeit erhöht die Wahrscheinlichkeit für die Art von Missverständnissen in der Entwicklung, die Dan North beseitigen wollte, und steht den Prinzipien, die zur Formulierung von BDD geführt haben, somit entgegen.

Abschnitt 3

Unvollständigkeit bei verhaltensgetriebenen Anforderungen

Im Rahmen von BDD kommt der von Jones beschriebene Informationsfluss als „Feedbackschleife" vor: Anforderungen werden formuliert, die Software wird entwickelt und getestet, und schließlich werden Berichte zur Evaluierung an ständige Stakeholder übergeben. Hier besteht das Risiko der Unschärfe darin, dass die „Schleife" sich unendlich wiederholt, falls bereitgestellte Software nie den Vorstellungen entspricht oder die Anforderungen sich ständig weiterentwickeln.

Durch Ableiten der Anforderungen und Testszenarien anhand von Unternehmenszielen und gewünschtem Verhalten verdeutlicht das Testing, was passieren soll. Anders ausgedrückt: Es ist vor allem auf ideale Testfälle konzentriert. Dies wird in den natürlichen Sprachen sichtbar, die zur Formulierung von Szenarien vorgeschlagen werden.

Wenn man die Logik eines Systems anhand von booleschen Operatoren betrachtet, kann es vollständig auf drei Funktionen reduziert werden: NOT, OR und AND, mit XOR als deren Kombination. Dies kann wiederum in Form von „IF-THEN"-Aussagen gedacht werden.

So legt beispielsweise Gherkin mehr Gewicht auf „IF", „AND" und „THEN". Die Sprache beschreibt die Anfangsbedingungen und die Argumente für den Aufbau eines Szenarios sowie die tatsächlich vorhandenen „Trigger" (d. h. das „IF") und das erwartete Ergebnis des Szenarios (das „THEN"). „AND" kommt insofern vor, als jeder Schritt unterschiedliche Annahmen oder Trigger enthalten kann. „OR" spielt beim Sammeln von Testszenarien in diesem Format offensichtlich keine Rolle. Dies wird deutlicher bei der in RSpec verwendeten alternativen natürlichen Sprache, in der Szenarien in Form einiger „Wenn-Dann"-Aussagen formuliert werden können[8]. Anders ausgedrückt berücksichtigen solche natürlichen Sprachen nicht, was passiert, wenn keine Trigger vorhanden sind.

Doch die durch „OR" reflektierten Entscheidungen sind für die Logik eines Systems grundlegend, und jede von ihnen steht für einen potenziell anderen Pfad, der vom System als Bestandteil des Testing hervorgehoben werden muss. Dies trifft insbesondere auf negative Pfade zu, die rund 80 % des gesamten Testing-Aufwands ausmachen sollten. Solche unerwarteten Ergebnisse treten genau dann auf, wenn die Trigger nicht vorhanden sind, wodurch „IF" nicht erfüllt wird. Es sind genau diese unerwarteten Ergebnisse und Sonderfälle, die ein System am ehesten zusammenbrechen lassen.

Manch einer führt an, dass BDD eine gute Methode für die Identifizierung der Pfade ist, die beim Design des Programms ausgelassen wurden, denn es weist unerwartete Ergebnisse bei ihrem Auftreten aus. Im Grunde ist BDD ein iterativer Prozess, weshalb das Verkürzen der „Feedbackschleife" bedeutet, dass man sich früher mit diesen negativen Pfaden befassen kann, da das Testing im Lebenszyklus vorgezogen wird. Manche gehen sogar so weit zu empfehlen, dass Organisationen die „Unschärfe begrüßen" sollen[9].

Doch wie erwähnt ist es sowohl kostspielig als auch zeitaufwendig, wenn Fehler erst beim Testing erkannt werden, und in der Praxis werden kurze Iterationen oft zu Mini-Wasserfallmodellen, bei denen das Testing sich überschlägt oder gar nicht erst stattfindet. Außerdem besteht das gravierendere Problem der Beobachtbarkeit von Fehlern. Das erwartete Ergebnis („THEN") eines Szenarios soll eine verifizierbare Bedingung sein, und das Testing soll sicherstellen, dass das erwartete Ergebnis aufgetreten ist, von den korrekten Triggern hervorgerufen wurde und aus den korrekten Aufbau-Argumenten hervorging. Doch wenn das Testing ad hoc stattfindet, gibt es keine Sicherheit darüber, ob die Ergebnisse aus dem vorgesehenen Grund aufgetreten sind oder weil zwei oder mehr Fehler sich gegenseitig aufgehoben haben.

Abschnitt 4

Modellbasiertes Testing und BDD

Um Vollständigkeit zu erreichen und die Wahrscheinlichkeit zu erhöhen, dass Software den Erwartungen entspricht, muss BDD zusätzlich zu „AND" auch mit „OR" arbeiten. Die einzeln präsentierten Szenarien von BDD und Sprachen wie Gherkin sollten dann zusammengeschlossen werden, um die Logik eines Systems widerzuspiegeln. Dadurch wird beim Testing im BDD auch berücksichtigt, was passiert, wenn die Trigger nicht feuern, obwohl sie es sollten, und ein System dadurch auf einen negativen Pfad bringen.

Beim Schreiben und Durchführen von Tests erstellen Tester normalerweise Modelle, wenngleich implizit. Beim negativen Testing berücksichtigen sie so vielleicht, was passiert, wenn ein Trigger nicht vorkommt, die Aufbau-Argumente aber vorgekommen sind. Noch einmal: Wenn beim BDD zwei Testszenarien den gleichen Schritt beinhalten, sind sie implizit durch „OR" verbunden. Sie können etwa den einen Trigger gemeinsam haben und den anderen nicht, was eine Entscheidung innerhalb der Systemlogik darstellt.

Wenn solches Modeling jedoch ad hoc geschieht, deckt das Testing wahrscheinlich nur jene Szenarien ab, die die Tester und Entwickler im Kopf haben. Das ist im BDD der Fall, wenn User Stories als separate, lineare Einheiten präsentiert werden, die nicht widerspiegeln, wie sie innerhalb der Logik eines Systems miteinander in Zusammenhang stehen. Die funktionale Abdeckung des Testing beträgt daher meist nur 10 bis 20 %, da selbst ein einfaches System tausende möglicher Kombinationen von Inputs und Outputs haben kann – mehr als irgendjemand im Kopf behalten und akkurat miteinander verbinden kann.

Wenn BDD erfordert, dass Tester ohnehin modellieren, besteht kein Grund dafür, dies nicht auf systematische Art und Weise zu tun und die Anforderungen dabei eindeutig und vor allem vollständig zu dokumentieren. Eine solche systematische Vorgehensweise ist nötig, wenn Testing alle möglichen Pfade durch ein System einschließlich negativer Pfade und unerwarteter Ergebnisse abdecken soll.

Abschnitt 5

Flowchart Modeling als Teil von BDD

Unten sehen Sie einen Vorschlag dafür, wie Modeling in die Entwicklung integriert werden kann, und zwar so, dass es synergetisch mit den besprochenen Prinzipien von BDD zusammenwirkt.

1. Spezifikation per Beispiel: Es sollte betont werden, dass Anforderungen nach wie vor von Verhalten getrieben werden können und dass aus den Interaktionen der Stakeholder abgeleitet werden kann, wie diese sich das System wünschen. Die vom Business formulierten Anforderungen werden entweder als Flowchart rekonstruiert, oder die ständigen Stakeholder bilden selbst einen Flow. Letzteres wird dadurch ermöglicht, dass ein Flowchart als ubiquitäre Sprache fungiert. Philip Howard von Bloor Research beschrieb ein Flowchart-Modell als etwas, das auf der gesamten für Tester und Entwickler nötigen funktionalen Logik über ein System aufbauen kann, ohne dass den Business-Verantwortlichen „der Mund offen stehenbleibt“[10].

   Wenn Entwickler und Tester vorhandene BDD-Anforderungen verwenden, besteht der Prozess darin, die sich überlagernden Schritte der Szenarien zusammenzufassen, sodass sie Entscheidungen (ORs) in der Systemlogik darstellen. Dies zwingt Modellierer dazu, in der Logik des Systems zu denken und dadurch Vollständigkeit zu erwirken, während CA Agile Requirements Designer (früher CA Test Case Optimizier) jegliche fehlerhaften Pfade identifiziert und Pfadhinweise für eventuell ausgelassene Szenarien gibt. Fehlende oder negative Pfade werden somit identifiziert, bevor die Feedbackschleife abgeschlossen ist, was die Zahl der erforderlichen Iterationen reduziert.

2. Modeling von Szenarien: User Stories sind genau genommen zu anspruchsvoll für Testing und Entwicklung. Stattdessen werden die zu entwickelnden und zu testenden Szenarien im Flowchart modelliert, wobei ihre Schritte Prozess- und Entscheidungsblöcke eines Flowcharts bilden und die Szenarien zu Pfaden durch eine Systemlogik werden. Wenn alle Szenarien modelliert sind, ist jede User Story gleichermaßen erfasst, da die Szenarien aus User Stories abgeleitet wurden.

   Alle Aspekte einer User Story können ohne Weiteres in ein Flowchart übertragen werden. So kann etwa in einem Prozessblock spezifiziert werden, wer der agierende Stakeholder ist, und beschrieben werden, wie ein „Kunde versucht, Geld an einem Geldautomaten abzuheben“. Die Pfade durch ein Flowchart können dann einen Unit Test in BDD widerspiegeln, wobei sie eine Rolle, ein gewünschtes Leistungsmerkmal und einen Nutzen oder ein erwartetes Ergebnis spezifizieren.

   Aufgrund von kombinatorischen Methoden gibt es keinen singulären, individuellen Pfad von Anfang bis Ende für jedes Szenario. Stattdessen werden Elemente der funktionalen Logik unterschiedlicher Szenarien oder Leistungsmerkmale in bestimmten Abschnitten des Flowcharts kombiniert. Dies lässt einen wichtigen Unterschied zwischen der Ableitung von

Tests aus BDD-Anforderungen mithilfe natürlicher Sprachen wie Gherkin und dem Modeling der funktionalen Kernlogik eines Systems erkennbar werden: Beim Modeling kann ein einzelner Testfall unterschiedliche Szenarien abdecken, da diese innerhalb eines Systems als Ganzes zusammengefasst statt als separate Einheiten präsentiert werden. Diese Zusammenfassung von User Stories in einem vollständigen System ist ein wichtiger Schritt für den Abbau von Unschärfe.

Unterschiedliche Szenarien können etwa bis zu einer bestimmten „Annahme" zusammengefasst werden, wenn ein System als einzelnes Flowchart modelliert wurde. Diese „Annahme" könnte ein Teilfluss oder ein Teilbereich unterschiedlicher Pfade sein, der dann durch einen Entscheidungsblock abgegrenzt wird.

Auf ähnliche Weise können mehrere „Wenns" in einem beliebigen Pfad in Form von Entscheidungs- oder Prozessblöcken auftreten. Unterschiedliche Szenarien werden dadurch so zusammengefasst, dass sie im Fall von gemeinsamen Triggern bei zwei oder mehr Szenarien durch dieselben Blöcke verlaufen können. Jede Entscheidung auf diesem Weg kann dann zu einem neuen Pfad oder einer Reihe von Pfaden führen, wobei sich dies so lange fortsetzt, bis ein System vollständig modelliert ist, einschließlich aller negativen Pfade. Wenn beispielsweise zwei Szenarien alle bis auf einen Trigger gemeinsam haben, könnten sie demselben Pfad folgen, bis sie den letzten Entscheidungsblock erreichen, wo sie sich dann trennen.

3. Die Feedbackschleife: Verifizierung des Modells. Wenn ein System von Anwendern und Business-Analysten modelliert wurde, ist keine Verifizierung des Modells erforderlich, und das BDD kann zum nächsten Schritt übergehen. In diesem Fall ist die Feedbackverzögerung sehr kurz, wie noch beschrieben wird.

   Wenn ein System von Testern und Entwicklern modelliert wurde, ist die Verifizierung fast ebenso schnell und einfach. Mit CA Agile Requirements Designer kann dies etwa durch Anwendungsfälle geschehen. Jeder Pfad durch das Flowchart steht für einen individuellen Anwendungsfall, der beim Modeling des Systems einem Testfall oder Szenario entspricht. Die Verifizierung besteht darin, eine Reihe von Anwendungsfällen durchzugehen, die als gemeinsame Klartextsprache und als Flowchart dargestellt sind, und zu bestätigen, dass das System dem beabsichtigten Design entspricht.

4. Die Feedbackschleife: Validierung des Modells. Da das Flowchart durch die funktionale Logik des Systems untermauert wird, können Testfälle automatisch daraus abgeleitet werden. Wie erwähnt, stellt jeder mögliche Pfad durch ein System einen Testfall dar, und CA Agile Requirements Designer kann sowohl automatisch mögliche Pfade identifizieren als auch die korrekten Messdaten für die durch sie gelieferte funktionale Abdeckung berechnen.

   Vor allem können unzweckmäßige oder redundante Tests wegfallen und mögliche Testfälle dedupliziert werden, wodurch anhand der kleinstmöglichen erforderlichen Testfallreihe die größtmögliche funktionale Abdeckung erreicht wird. Wie bereits ausgeführt, kann die Logik unterschiedlicher Szenarien zusammengefasst werden, sodass eine Reihe von Szenarien anhand einer kleineren Testfallreihe getestet werden kann, beispielsweise 15 mögliche Szenarien anhand dreier Testfälle. Die Testoptimierung bietet unterschiedliche Algorithmen zur Identifizierung der Pfade, die die größte funktionale Abdeckung bieten. In einem Fall etwa senkte CA Agile Requirements Designer die Zahl möglicher Tests von 326 auf nur 17, und das bei hundertprozentiger Abdeckung.

   Sobald diese Pfade gespeichert wurden, können sie exportiert und ausgeführt werden. Dies kann manuell geschehen, indem die mit Testdaten und erwarteten Ergebnissen verknüpften Testfälle durch ein Testmanagementtool wie HP ALM/QC geschickt werden. Alternativ können die Pfade als automatisierte Test-Scripts exportiert werden, die in zahlreichen Automation-Engines ausgeführt werden. In jedem Fall werden „lebendige Dokumentation" und Berichte durch das verwendete Testmanagementtool erzeugt.

Dieser automatisierte Prozess mag ähnlich wie der von Cucumber wirken, doch die Feedbackverzögerung ist wahrscheinlich wesentlich kürzer, und es sind weniger Feedbackschleifen erforderlich. Erst einmal macht die automatische Testfallgenerierung die manuelle Testfalldefinition überflüssig – im Fall von Cucumber macht sie die manuelle Umwandlung von Gherkin in Schrittdefinitionen in Ruby überflüssig. In einem Fall etwa brauchte CA Agile Requirements Designer 90 Minuten, um 108 Testfälle zu erstellen, die hundertprozentige funktionale Abdeckung lieferten. Dies beinhaltet auch die Zeit für das Design des Flowcharts. In der Praxis wird die Zeitersparnis größer sein, da das Flowchart leicht optimiert und wiederverwendet werden kann, wie unten erklärt wird.

Da den Entwicklern zudem vollständige, verifizierte, eindeutige Anforderungen zur Verfügung stehen, ist es wahrscheinlicher, dass sie auf Anhieb die gewünschte Software bereitstellen. Wie erwähnt, sind Unklarheiten für 56 % der Fehler verantwortlich, und in dieser Hinsicht konnte CA Agile Requirements Designer die Zahl der Fehler um bis zu 95 % senken. Bei Fehlern, die es von der Entwicklung ins Testing schaffen, ist es wahrscheinlich, dass sie dank der durch die Testfälle erzeugten hundertprozentigen funktionalen Abdeckung beim ersten Auftreten erkannt werden. Neben der Zeitersparnis durch weniger Nachbesserung nimmt wahrscheinlich auch die Testdurchführung weniger Zeit in Anspruch. Das Duplizieren von Tests hat den Testing-Aufwand in der Regel um bis zu 30 % gesenkt, während die Testoptimierung die Gesamtzahl der zur Abdeckung aller Szenarien nötigen Tests signifikant reduzieren kann.

Abschnitt 6

Schnell und mühelos auf Änderungen reagieren

Ein Einwand könnte der Tatsache gelten, dass das Modeling von Flowcharts in einer BDD-Umgebung zeitaufwendig ist und die Zeit besser für die Ausführung und Wiederausführung von Testiterationen genutzt werden könnte.

Hier wäre zunächst zu beachten, dass diese Kritik nur dann berechtigt ist, wenn Tester und Entwickler Flowcharts selbst konstruieren müssen: Wenn Business-Analysten und End User die Flowchart-Erstellung dem Gebrauch von Sprachen wie Gherkin vorziehen würden, würde es sie nicht geben. Doch selbst wenn technische Teams BDD-Anforderungen rekonstruieren müssen, beansprucht die Erstellung eines Flowcharts im Allgemeinen nicht viel Zeit. Wenn ein Flowchart einmal erarbeitet wurde, kann es ohne Weiteres wiederverwendet werden, auch wenn sich die Anforderungen ändern. Dies steht im Gegensatz zu der Annahme, dass vollständige Dokumentation notwendigerweise bedeutet, dass auf Änderungen nicht schnell reagiert werden kann.

Mit CA Agile Requirements Designer können Änderungen an Flowcharts schnell und einfach vorgenommen werden, da Diagrammen problemlos neue Elemente funktionaler Logik hinzugefügt werden können. Jegliche fehlerhaften Testfälle werden dann automatisch identifiziert und repariert, während doppelte oder redundante Tests wegfallen. CA Agile Requirements Designer erzeugt dann alle neuen Testfälle, die erforderlich sind, um maximale funktionale Abdeckung zu erreichen.

Das maximiert den Nutzen der ursprünglichen Arbeit – der Ausarbeitung des Flowcharts – und eliminiert den Zeitaufwand für die manuelle Überprüfung der Testfälle im Fall von Änderungen. In einem Unternehmen konnte ein vorhandenes Flowchart nach einer Änderungsanfrage innerhalb von zwei Minuten optimiert werden. CA Agile Requirements Designer identifizierte und reparierte automatisch die drei betroffenen Testfälle und ließ dabei 64 unberührt. Wie dieses Beispiel zeigt, kann Flowchart Modeling außerdem bedeuten, dass nur die Aspekte eines Systems erneut getestet werden, die von Änderungen betroffen sind, was die Effizienz der Feedbackschleife verbessert.

Insgesamt wird die Zeit für die Erstellung eines Flowcharts sehr wahrscheinlich durch eine Zeitersparnis bei Folgendem wettgemacht: Nachbesserung und Debugging aufgrund von Unvollständigkeit und Unklarheiten, Erstellung und Durchführung von Testfällen sowie Implementierung von Änderungsanfragen. Somit steht vollständige Dokumentation den Prinzipien der verhaltensgetriebenen Entwicklung nicht entgegen, sondern kann deren Implementierung verbessern.

Abschnitt 7

Zusammenfassung

Flowchart Modeling bietet eine Technik, durch die BDD mit vollständiger Dokumentation ergänzt werden kann, ohne dass ihre Grundprinzipien infrage gestellt werden. Ein Flowchart kann direkt aus den Interaktionen der Stakeholder abgeleitet werden und ist für Business wie IT gleichermaßen verständlich – es liefert eine ubiquitäre Sprache, die mit der für Testing- und Entwicklungsteams nötigen funktionalen Sprache über ein System überlagert werden kann. Ein solches Flowchart kann also helfen, die „Feedbackverzögerung" zu reduzieren, da Testing-Zyklen verkürzt werden und der manuelle Aufwand beim Testing verringert wird.

Formales Modeling erhöht die Wahrscheinlichkeit, dass Software auf Anhieb besser den Anforderungen entspricht, da es eine Reihe verhaltensgetriebener Anforderungen liefert, die sowohl eindeutig als auch vollständig sind. Das reduziert die Anzahl der Feedbackschleifen, die nötig ist, damit die Software zu jedem beliebigen Zeitpunkt die Anforderungen der Business-Verantwortlichen erfüllt, sodass diese sich auf Innovation konzentrieren können, statt viel Zeit mit Wiederholungen und frustrierender Nachbesserung zu verbringen. Die Fähigkeit zur kontinuierlichen Entwicklung von Lösungen in dem Bestreben, ihren Nutzen für Kunden zu maximieren, wird zusätzlich durch die Fähigkeit begünstigt, schnell und mühelos auf Änderungen zu reagieren.

Abschnitt 8

Informationen zum Autor

Während seiner fast 30-jährigen Karriere war Huw Price leitender technischer Architekt bei verschiedenen amerikanischen und europäischen Softwareunternehmen und leistete multinationalen Banken, großen Versorgungsanbietern und Gesundheitsdienstleistern Unterstützung bei anspruchsvollsten Architekturentwürfen. Price, der 2010 von QA Guild zum „IT Director of the Year" gewählt wurde, hat sich über viele Jahre auf Testautomatisierungstools spezialisiert und zahlreiche innovative Produkte auf den Markt gebracht, die das in der Softwarebranche vorherrschende Testing-Modell transformiert haben. Derzeit hält er weltweit Vorträge, und seine Arbeiten wurden in zahlreichen technischen Publikationen wie beispielsweise im Professional Tester oder CIO Magazine veröffentlicht.

Sein letztes Projekt, Grid-Tools, wurde im Juni 2015 von CA Technologies erworben. Zu dem Zeitpunkt hatte Grid-Tools bereits fast zehn Jahre lang an der Neudefinition von Testing-Strategien großer Unternehmen mitgewirkt. Unter der visionären Herangehensweise und Führung von Huw Price übernahm das Unternehmen einen stark datenorientierten Ansatz für das Testing und lancierte neue Konzepte aus der Feder von Price, etwa „Datenobjekte", „Datenvererbung" und „ein zentrales Test Data Warehouse".

Kontaktieren Sie CA Technologies unter ca.com/de.

CA Technologies (NASDAQ: CA) entwickelt Software, die Unternehmen bei der Umstellung auf die Application Economy unterstützt. Software steht in allen Branchen und in allen Unternehmen im Mittelpunkt. Ob Planung, Entwicklung, Management oder Security – CA Technologies arbeitet weltweit mit Unternehmen zusammen, um die Art, wie wir leben, Transaktionen abwickeln und kommunizieren, in mobilen, privaten und öffentlichen Cloud-Umgebungen oder in verteilten Systemen und Mainframe-Umgebungen neu zu gestalten. Weitere Informationen finden Sie unter **ca.com/de**.

1 Dan North: Introducing BDD (2006), abgerufen am 06.03.2015 von http://dannorth.net/introducing-bdd/

2 Chaos Manifesto 2013 (Standish Group: 2013), abgerufen am 07.03.2015 von http://www.versionone.com/assets/img/files/CHAOSManifesto2013.pdf

3 Requirements Based Testing Process Overview (Bender RBT: 2009), abgerufen am 05.03.2015 von http://benderrbt.com/Bender-Requirements%20Based%20Testing%20Process%20Overview.pdf

4 Why testing should start early in software development life cycle? (Software Testing Class: 2012), abgerufen am 06.03.2015 von http://www.softwaretestingclass.com/why-testing-should-start-early-in-software-development-life-cycle/

5 Bender RBT, Requirements Based Testing Process Overview

6 Kathleen Barret: Business Analysis: The Evolution of a Profession (IIBA: 2013), abgerufen am 05.03.2015 von http://www.iiba.org/Careers/Careers/Business-Analysis-The-Evolution-of-a-Profession.aspx

7 Vgl. Erik Kamsties: „Understanding Ambiguity in Requirements", zitiert in: Engineering and Managing Software Requirements (Springer: 2005), S. 250

8 Ein Beispiel finden Sie hier: http://en.wikipedia.org/wiki/Behavior-driven_development#Story_versus_specification

9 Dan North: Embracing Uncertainty (Goto Con: 2013), abgerufen am 06.03.2015 von http://gotocon.com/dl/goto-chicago-2013/slides/DanNorth_EmbracingUncertainty.pdf

10 Test Case Generation, abgerufen am 06.03.2015 von https://www.ca.com/de/collateral/industry-analyst-report/bloor-research-market-report-test-case-generation.register.html

CS200-196736_0516