Agiles Testen in Großprojekten mit TDD und Testaspekten: Beobachtungen und erste Erfahrungen

Melanie Späth
Capgemini sd&m AG, München
melanie.spaeth@gmx.de

Michael Mlynarski
Universität Paderborn, Software Quality Lab
mmlynarski@s-lab.upb.de

1 Einführung

Immer mehr Softwareentwicklungsunternehmen sehen Bedarf für mehr Agilität in ihrem Geschäft. Vorgehen zur agilen Softwareentwicklung, wie beispielsweise Extreme Programming (XP), betonen, dass Testen dabei eine wesentliche Rolle spielt [1]. Speziell Methoden wie Test-getriebene Entwicklung (TDD) [2], nach der die Entwickler gegen vorab definierte Testfälle programmieren, versprechen durch testbaren Code und weniger Fehler hohe Qualität. Obgleich für kleine und mittelgroße Projekte bereits der Mehrwert von TDD nachgewiesen wurde (bspw. [3]), fehlt dies für Großprojekte noch. Bei Capgemini sd&m führen wir Projekte mit agilen Entwicklungsmethoden wie SCRUM [11] durch. Zugleich mussten wir feststellen, dass in den meisten Projekten TDD nicht eingesetzt wird. In Interviews mit den verantwortlichen Testmanagern haben wir hierfür mehrere Gründe identifiziert:

- Projektleiter scheuen das Risiko zusätzlicher Kosten und nicht eingehaltener Termine.
- TDD-Ansätzen fehlt es oft an strukturierten Testdesign-Methoden sowie messbarer funktionaler Testabdeckung.
- Die entstehenden Testfälle sind sehr feingranular.
- Durch TDD entsteht meist eine große Menge an Testfällen, die über einen längeren Projektzeitraum nur mit viel Aufwand gewartet werden kann.

Bei Capgemini sd&m haben wir eine Testmethodik eingeführt, die auf Großprojekte zugeschnitten ist. Sie basiert auf ISTQB [8] und definiert für alle Teststufen einen klaren Testprozess sowie ein Testspezifikationsverfahren. Früh entworfene „Testaspekte“ ermöglichen es uns, dem Entwicklungsteam einen Vorschlag zu liefern, was in den Komponententests abgesichert werden soll. Hierdurch möchten wir die oben erwähnten TDD-Schwachstellen beheben.

Im folgenden Abschnitt 2 gehen wir näher auf typische Probleme beim Komponententest in großen Projekten ein. In Abschnitt 3 zeigen wir unsere und Industrie-Erfahrungen mit TDD in Großprojekten auf. Hierauf aufbauend leiten wir in Abschnitt 4 Anforderungen für einen praktikablen Ansatz ab. In Abschnitt 5 stellen wir unsere Lösung kurz vor, während wir in Abschnitt 6 zeigen, wie dieser für eine Zusammenarbeit zwischen Entwicklungs- und Testteam genutzt werden kann. In Abschnitt 7 zeigen wir die Erfahrungen mit unseren Ansatz auf.

2 Typische Probleme

Es ist zur gängigen Praxis für große Projekte geworden, dass die Entwicklungsteams die Komponententests schreiben, während ein separates Team für die höheren Teststufen verantwortlich ist. Leider hängt die Qualität der Komponententests sehr häufig davon ab, unter welchem Zeitdruck das jeweilige Entwicklungsteam steht. Um dem zu entgegnen, nehmen Projektleiter gerne Codeüberdeckungs-kennzahlen in die Entwicklungsziele auf.

Jedoch garantiert gute Codeüberdeckung in den Komponententests noch lange keine gute Testqualität. Zum einen kann durch Codeüberdeckungsmaße allein nicht die fehlende Funktionalität festgestellt werden. Testfälle, die durch viele Code-Teile laufen, aber keine Ergebnisse prüfen, führen zu hoher Codeüberdeckung während sie gleichzeitig von schlechter Testqualität sind.

Blindes Vertrauen in hohe Codeüberdeckung führt zudem häufig zu vielen Testfällen, ohne Fokus auf fachlich besonders kritische Code-Teile oder solche mit hoher Fehlerwahrscheinlichkeit. Zusätzlich führt gerade hohe Komplexität auch oft zu hohem Druck in der Entwicklung. Hoher Druck wiederum führt zu weniger Testfällen. Wird die Zeit eng, sinkt meist die Testqualität. Häufig bleiben so viele Fehler bis zum Systemtest unentdeckt, schlimmstenfalls sogar bis zum Produktivbetrieb.

Um hohe Codeabdeckung zu erreichen, müssen viele Testfälle erstellt werden. Jeder Change Request führt dann dazu, dass mehrere Testfälle geändert werden müssen. Dieser plötzlich zusätzlich notwendige Aufwand wird durch die Entwickler wenn möglich vermieden. In lang laufenden Projekten ist es deshalb beinahe üblich, wenn bis zu 40% der Komponententests regelmäßig fehlschlagen. In solchen Projekten sind die Komponententests im Ganzen nicht mehr verlässlich.

3 Hilft Test-Driven Development?

Ausgehend von den geschilderten Problemen liegen Ansätze wie TDD als Lösung auf der Hand. Deshalb haben wir systematisch die Erfahrungen innerhalb unseres Unternehmens mit TDD zusammengetragen. Hierzu haben wir Interviews mit erfahrenen Entwicklern, Testmanagern und Projektleitern geführt. Ihnen haben wir Fragen gestellt zum Projektkontext, zu Testdesignmethoden, sowie zu Vorteilen und Problemen bei der Nutzung von TDD. Die Ergebnisse haben wir zudem mit aktuellen Forschungsergebnissen abgeglichen.

Insgesamt sahen alle Interviewpartner Vorteile in der Nutzung von TDD, merkten jedoch auch an, dass TDD in der Form, wie es von Kent Beck in [2] beschrieben wurde, nicht für große Projekte anwendbar ist.

Als Hauptvorteile erkennen unsere Interviewpartner das tiefere Verständnis an, das Entwickler von dem

implementierten Code gewinnen. Dadurch dass sie als erstes Testfälle schreiben, denken sie vorab über die Funktionalität nach, die das System erfüllen muss. Diese Art zu denken deckt auch Spezialfälle auf. Und dadurch, dass jeder geschriebene Testfall sofort ausgeführt wird, ermutigt das die Entwickler zusätzlich, über die Testbarkeit ihres Codes nachzudenken.

Ein weiterer Vorteil ist, dass überhaupt Testfälle für den Komponententest verfügbar sind, was in Nicht-TDD-Projekten nicht selbstverständlich ist. Die Testfälle werden früher geschrieben und sind größtenteils automatisiert. Dadurch wird Wiederverwendung in den Regressionstests möglich.

Andererseits haben die Interviewpartner auch verschiedene Restriktionen genannt. Großprojekte nutzen häufig Entwicklungsmodelle wie RUP [7]. In solche Entwicklungsprozesse TDD einzubetten scheint schwierig. Ein Grund hierfür ist, dass das initiale Systemdesign nicht vermieden werden kann. Die Komponenten- und Subsystemstruktur komplexer Systeme kann nicht in solch kleinen Schritten erfolgen, wie TDD diese beschreibt. Der Gesamtüberblick ist notwendig, um typische Designfehler zu vermeiden. Trotzdem sollte für sehr wichtige Subsysteme der Einsatz von TDD erwogen werden.

Die große Anzahl von Testfällen, die durch TDD erstellt werden, muss reduziert werden. Obwohl TDD insgesamt zu weniger Code führt (siehe [6]), steigt der Wartungsaufwand für die Testfälle an [12]. Rendell zeigt in [8] dass Entwickler dazu tendieren, zu viele Testfälle zu erzeugen wenn sie TDD nutzen. Test-getriebener Entwicklung fehlt es typischerweise an strukturierten Testdesignmethoden. Dies wurde auch von Fraikin et al. in [5] erwähnt. Ein grüner Komponententestfall ist nur dann bedeutend, wenn dadurch eine gute funktionale Testabdeckung erreicht wird. Im Unterschied zur Codeabdeckung beschreibt diese den Grad, zu dem Testfälle die Artefakte der Anforderungs- und Systemspezifikation überdecken.

Unserer Erfahrung nach ist es nicht ratsam, TDD auf Komponententestfälle zu beschränken. Die Idee, zuerst strukturiert über die Funktionalität nachzudenken bevor programmiert wird, sollte auch auf die höheren Teststufen übertragen werden. Dies wurde auch von Crispin in [4] erwähnt. Sie schreibt, dass Testfälle für den Systemtest auch für Entwickler, die TDD nutzen interessant sind. Komponententestfälle können also durch tiefes Verständnis der Anforderungen verbessert werden.

Die Bedeutung separater Testteams für den System- und Subsystemtest wurde von allen Interviewpartnern hervorgehoben. Selbst wenn die erstellten Komponententests gut sind, können kritische Systeme nicht ohne weitere Tests auf System-Level ausgeliefert werden. Ähnliche Beobachtungen haben Sangwan and Laplante in [10] beschrieben.

Die letzte Einschränkung ergibt sich durch Kosten und Risiken bei der Nutzung von TDD. Durch das Schreiben vieler Testfälle riskieren die Entwickler, mit der eigentlichen Funktionalität nicht rechtzeitig fertig zu werden. Manager versuchen oft Zeit zu sparen, indem sie am Test kürzen. Auch andere Unternehmen haben bereits zusätzliche Kosten durch TDD festgestellt. Nach einer Studie eines Microsoft-Projektes in [3] wurde ein Zuschlag von 15-35% nachgewiesen. Diese Kosten müssen wiederum mit den Einsparungen für Bugfixing verglichen werden.

4 Lösungsanforderungen

Die bislang gesammelten Erfahrungen legen nahe, dass TDD allein die in Abschnitt 2 diskutierten Probleme nicht lösen kann, jedoch ein Schritt in die richtige Richtung ist. Um von den TDD-Vorteilen zu profitieren und gleichzeitig seine Nachteile zu vermeiden, definieren wir im Folgenden einige Anforderungen an einen praktikablen Ansatz.

Anforderung 1. Erst denken, dann coden.

Die Erfolgsgeschichte hinter Test-getriebener Entwicklung liegt darin, dass durch das Vorab-Schreiben der Testfälle, die Entwickler ein besseres Verständnis über die Anforderungen und Ziele gewinnen, bevor sie mit der Implementierung starten. Vereinfacht nennen wir das Prinzip „Think first“. Natürlich gilt dieser Grundsatz analog für die Testteams: Erst denken, dann detaillierte Testfälle schreiben. Genau hier kommen Testaspekte ins Spiel (siehe Abschnitt 5).

Anforderung 2. Priorisierung im Komponententest.

Risiko-orientiertes Testen ist in der Praxis das Nummer-1-Kriterium, um zu handhabbaren Test-Sets zu gelangen. Seltsamerweise werden Risiken und Prioritäten fast ausschließlich für höhere Teststufen genutzt. Hingegen programmieren die Entwicklerteams Testfälle je nach Lust und Laune. Manchmal werden zumindest Codeabdeckungskriterien als Ziele gesetzt. Jedoch gelten diese meist einheitlich für den gesamten Quell-Code, was schließlich in Tausenden von Komponententests endet. Deshalb empfehlen wir, die Prioritäten der Anforderungen herunterzubrechen auf feingranularere Spezifikationsartefakte oder die entsprechenden Testobjekte wie Komponenten.

Anforderung 3. Teststufen-übergreifende Strategie.

Großprojekte trennen meist die Entwickler- und Testteams. Während die Entwickler noch die Komponententests verantworten, konzentrieren sich dedizierte Testteams auf Subsystem- oder Systemtests. Meist schließen die Testmanager daraus, dass sie sich um die Komponententests nicht kümmern müssen.

Tatsächlich ist dies höchst ineffizient: Gute Testqualität zu vernünftigen Kosten kann nur durch eine teststufen-übergreifende Teststrategie erreicht werden. Das Testen auf unterschiedlichen Teststufen folg dem wollbekannten Prinzip “teile und herrsche”,. Die Capgemini sd&m Testmethodik unterscheidet vier Teststufen: Komponenten-, Subsystem-, System- und Abnahmetest. Hierdurch wird die Zusammensetzung des Systems direkt auf die Teststufen zugeordnet. Um Testfälle für die höheren Teststufen effektiv zu spezifizieren, müssen die Testdesigner wissen, was bereits in den darunter liegenden Teststufen abgesichert wurde – und was tatsächlich noch fehlt.

Anforderung 4. Gemeinsames Testdesign zwischen Test- und Entwicklungsteam.

Anforderung 3 verdeutlicht, dass Testteams der höheren Teststufen wissen müssen, was in den Komponententests getestet wird, um sich darauf verlassen zu können. Nachdem sie über das detaillierte Design und die Codestrukturen wenig wissen, sind sie üblicherweise weit davon entfernt, Komponententestfälle selbst schreiben zu können. Die Entwicklerteams andererseits kennen die Details im Code sehr gut, ihnen fehlt jedoch häufig der Blick für das Ganze.

5 Think-First mit Testaspekten

Ein wichtiger Bestandteil unserer Testmethodik, das Testspezifikationsverfahren, unterscheidet bewusst zwischen Testdesign und Testrealisierung. Im Testdesign konzentrieren wir uns auf die Frage „Was soll getestet werden“, erst in der Testrealisierung steht die Frage “Wie soll les getestet werden” im Vordergrund. Ergebnis sind letztendlich Testfälle für die manuelle und Testskripte für die automatisierte Testdurchführung.

Während viele Testdesign-Methoden direkt Testfälle aus der Spezifikation ableiten legen wir besonderen Wert auf einen Zwischenschritt, die so genannten Testaspekte. **Testaspekte beschreiben kurz, “Was” genau getestet werden soll, ohne das “Wie” zu beschreiben**. Testaspekte werden strukturiert für jedes test-relevante Artefakt aus der Anforderungs- oder Systemspezifikation (bswp. Anwendungsfall) abgeleitet. Hierfür nutzen wir ein eigenes Verfahren, das primär auf Kombinationstabellen und Äquivalenzklassen basiert.

Um Nachvollziehbarkeit zu gewährleisten, bezieht sich jeder Testaspekt auf das Spezifikationsartefakt, von dem es primär abgeleitet wurde. Zusätzlich gehört ein Testaspekt immer zu einer bestimmten Teststufe. Schließlich werden die Testaspekte entlang der Produktrisiken priorisiert.

Testaspekte enthalten keine Testschritte oder detaillierte Beschreibungen wie beispielsweise Vor- und Nachbedingungen. Trotzdem werden sie so präzise beschrieben, dass in der Testrealisierung ein Testaspekt in jeweils genau einen Testfall, maximal zwei Testfälle, übergeht. Ein Beispiel findet sich in Abbildung 1.

Id	Titel	Beschreibung	Trace	Teststufe	Prio
1	Erstelle eine erfolgreiche Hotelbuchung.	Eine Buchungsreservierung für ein Standard-Hotel ohne Extras wird erstellt.	UC_Buche_Hotel	System-Test	hoch
2	Prüfe das Abflugsdatum.	Das Abflugsdatum wird auf syntaktische Korrektheit und Konflikte mit dem Ankunftsdatum geprüft.	DIA_Buchung	Komponenten-Test	niedrig
3	Suche nach einem Luxus-Hotel in einem EU-Staat.	Ein Luxus-Hotel (mehr als 3 Sterne) in einem EU-Staat wird gesucht.	UC_Suche_Hotel	Subsystem-Test	mittel

Abbildung 1. Beispiel dreier Testaspekte verschiederner Teststufen.

Wir werden das Konzept an einem Beispiel verdeutlichen anhand dreier Testaspekte für ein neues Hotelbuchungs-Portal. Testaspekt 1 bezieht sich auf den erfolgreichen Prozessdurchlauf für eine Hotelbuchung. Testaspekt 2 adressiert die Plausibilisitätsprüfung für das eingegebene Ende-Datum. Testaspekt 3 zielt auf eine erfolgreiche Hotelsuche mit definierten Kriterien ab.

In diesem Beispiel ist Testaspekt 1 ein Systemtestaspekt. Um Testaspekt 3 zu testen, müssen wir warten bis das Subsystem “Hotelsuche” fertig gestellt ist, während Testaspekt 2 bereits im Komponententest abgesichert werden kann.

Der Zwischenschritt über Testaspekte bringt verschiedene Vorteile: Die Liste der Testaspekte liefert eine schnelle und komfortable Übersicht über alles, was in den verschiedenen Teststufen mit welcher Priorität getestet werden muss. Sie bilden damit eine perfekte Basis für Reviews auf Vollständigkeit.

Durch die Trace-Links auf die korrespondierenden Spezifikationsartefakte können die Testdesigner leicht herausfinden, welche Testaspekte und damit Testfälle angepasst werden müssen, wenn sich einzelne Anforderungen ändern. Die Trace-Links ermöglichen zudem eine präzise Messung der funktionalen Testabdeckung.

Die bewusste Zuordnung von Testaspekten auf Teststufen unterstützt frühes Testen. Jeder Testaspekt wird dadurch so früh wie möglich abgesichert. Zusätzlich werden unbeabsichtigte Redundanzen zwischen Teststufen wirkungsvoll durch eine Testdesignstrategie vermieden, die alle Teststufen umfasst.

Priorisierte Testaspekte bilden einen „Notfallplan“ für Testmanager. In gewissen Projektsituationen müssen Zeit oder Kosten eingespart werden durch Streichung von Testfällen. Konsequent gesetzte Prioritäten erleichtern diese Aufgabe. Außerdem helfen sie dem Testmanager, verbleibende Risiken schnell einsehen und benennen zu können.

Die Priorisierung von Testaspekten geschieht durch Analyse der Kritikalität und Ausführungshäufigkeit des zugrunde liegenden Spezifikationsartefakts (bspw. eines Anwendungsfalls). Ausführungshäufigkeit und Kritikalität können beide in drei Kategorien eingeteilt werden: 1=hoch, 2=mittel und 3=niedrig. Durch Multiplikation beider Faktoren definiert der Testmanager die Prioritätsklassen. Meist sind hier wiederum drei Klassen für “hoch“ (Werte 1-2), „mittel“ (Werte 3-6) und niedrig (Werte >6) ausreichend.

Letztendlich bilden Testaspekte das perfekte Werkzeug für einen gewinnbringenden Austausch zwischen Test- und Entwicklungsteam.

6 Koordination mit Testaspekten

Von einer Top-Down-Sicht kommend spezifizieren Testdesigner ihre Testaspekte, die entsprechend ihrer Granularität in verschiedene Teststufen aufgeteilt werden (Schritt 1). Alle Komponententestaspekte werden dabei in einer Liste gesammelt. Diese Liste wird initial an das Entwicklungsteam übergeben (Schritt 2). Der Austausch wird in Abbildung 2 aufgezeigt.

Prioritäten für Subsysteme, Komponenten oder die zugrunde liegenden Spezifikationsartefakte werden dem Entwicklerteam mitgeteilt. Der Testmanager legt entlang dieser Prioritäten Qualitätsrichtlinien fest. Solche Richtlinien können Abdeckungskriterien umfassen, beispielsweise 90% Codeabdeckung für Komponenten mit Prio 1, 50% für solche mit Prio 2, 30% für Komponenten mit Prio 3. Dank der Testaspekt-Übersicht können zusätzlich funktionale Testabdeckungsmaße genutzt werden. Die Richtlinien sollten

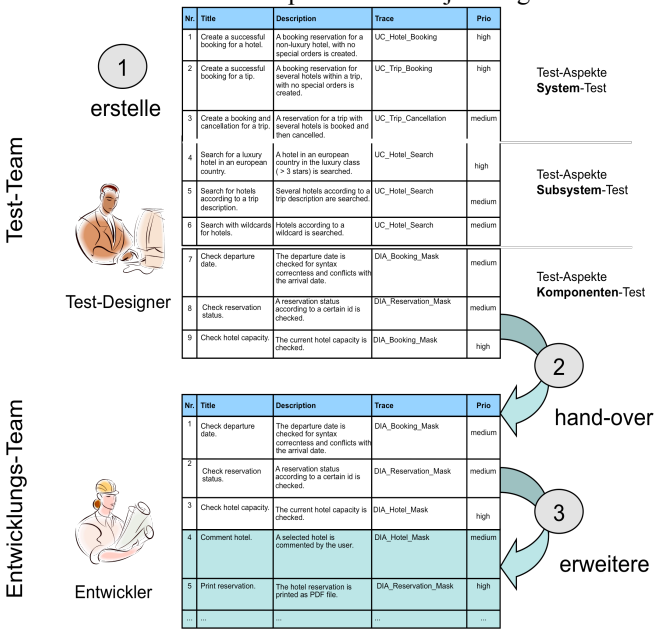
zudem enthalten, wie die Testaspekte abgeleitet werden sollen, beispielsweise formal durch Äquivalenzklassenbildung für Funktionalität mit Prio 1.

Nun sind die Entwickler dran. Sie können die Testaspektliste erweitern durch ihre White-Box-Sicht (Schritt 3). Zum Beispiel kann Testaspekt 2 aus Abbildung 1 erweitert werden durch Prüfung, ob das eingegebene Datum im gregorianischen Kalender existiert. Die dadurch entstehenden Testaspekte können schließlich für manuelle oder automatisierte Testdurchführung ausformuliert werden. Die Testaspekte werden vor der Implementierung erstellt – sie erfüllen damit den Hauptgedanken von TDD.

7 Erfahrungen

Wir integrieren diese Ergebnisse in unseren firmenweiten Testansatz. Hierin haben wir einige der Hauptideen von Test-getriebener Entwicklung und anderen Test-First-Ansätzen kombiniert mit Standardtestprozessen wie [8], sowie mit praktischen Ansätzen, die in verschiedenen Projekten entwickelt wurden. Seit die unternehmensweite Testmethodik im Dezember 2008 publiziert wurde, ist das Testspezifikationsverfahren in der gesamten Firma verbreitet.

Seitdem nutzen bereits mehr als 15 Großprojekte Testaspekte, darunter 6 Projekte nutzen bereits eine Teststufen-übergreifende Teststrategie. Die genannten Projekte sind allesamt große Projekte aus den Branchen Automobil, Öffentlicher Bereich, Telekommunikation und Logistik. Deren Feedback zu dem neu eingeführten Testdesign-Ansatz und speziell dem gemeinsamen Design der Komponententests ist durchgängig sehr positiv. Die Projekte gaben an, dass sie speziell von der Übergabe der Testaspekte profitieren: Hierdurch wird redundantes Testen wirkungsvoll vermieden. Dadurch können sie sich besser auf die relevanten Testaspekte ihrer jeweiligen Teststufe konzentrieren, während sie sich darauf verlassen können, dass Testaspekte der darunter liegenden Teststufen tatsächlich bereits getestet wurden. Erste Beobachtungen zeigen auch den Einfluss auf die Kommunikation zwischen Entwicklungs- und Testteam.



Nr.	Title	Description	Trace	Prio
1	Create a successful booking for a hotel.	A booking reservation for a non-luxury hotel, with no special orders is created.	UC_Hotel_Booking	high
2	Create a successful booking for a tip.	A booking reservation for several hotels within a trip, with no special orders is created.	UC_Trip_Booking	high
3	Create a booking and cancellation for a trip.	A reservation for a trip with several hotels is booked and then cancelled.	UC_Trip_Cancellation	medium
4	Search for a luxury hotel in an european country.	A hotel in an european country in the luxury class. (> 3 stars) is searched.	UC_Hotel_Search	high
5	Search for hotels according to a trip description.	Several hotels according to a trip description are searched.	UC_Hotel_Search	medium
6	Search with wildcards for hotels.	Hotels according to a wildcard is searched.	UC_Hotel_Search	medium
7	Check departure date.	The departure date is checked for syntax correctness and conflicts with the arrival date.	DIA_Booking_Mask	medium
8	Check reservation status.	A reservation status according to a certain id is checked.	DIA_Reservation_Mask	medium
9	Check hotel capacity.	The current hotel capacity is checked.	DIA_Booking_Mask	high

Nr.	Title	Description	Trace	Prio
1	Check departure date.	The departure date is checked for syntax correctness and conflicts with the arrival date.	DIA_Booking_Mask	medium
2	Check reservation status.	A reservation status according to a certain id is checked.	DIA_Reservation_Mask	medium
3	Check hotel capacity.	The current hotel capacity is checked.	DIA_Hotel_Mask	high
4	Comment hotel.	A selected hotel is commented by the user.	DIA_Hotel_Mask	medium
5	Print reservation.	The hotel reservation is printed as PDF file.	DIA_Reservation_Mask	high
...	...	...	...	...

Abbildung 2. Nutzung von Testaspekten zur Abstimmung der Komponententestinhalte zwischen Test- und Entwicklungsteam

8 Ausblick

In diesem Artikel haben wir die typischen Probleme mit Komponententests in großen Projekten zusammen getragen. Wir haben dann einen Ansatz vorgestellt, basierend auf den Grundsätzen von TDD sowie den Erfahrungen aus tatsächlichen Projekten bei Capgemini sd&m. Unser Ansatz verbindet strukturiertes Testdesign mit TDD-Konzepten. Dadurch heben wir die Testqualität in den Projekten und verbessern die Kommunikation zwischen Test- und Entwicklungsteam. Erste Erfahrungen zeigen, dass diese Verbindung beide Teams bei der effektiven Planung und Durchführung von Komponententests unterstützt.

Wir werden den Ansatz in weiteren Projekten evaluieren. Das Hand-Over-Konzept ist gerade auch für stark verteilte oder Offshore-Projekte sehr interessant. Wir planen deshalb den beschriebenen Ansatz in die Offshore-Methodik bei Capgemini sd&m zu integrieren.

LITERATUR

[1] K. Beck *"Embracing Change with Extreme Programming"* In: Computer, Vol. 32 / 1999, p. 70--77. IEEE Computer Society Press, Los Alamitos 1999.

[2] K. Beck *"Test-Driven Development"* Addison-Wesley Pearson Education, Boston, 2003

[3] T. Bhat and N. Nagappan *"Evaluating the Efficacy of Test-Driven Development: Industrial Case Studies"*. In: Proceedings of the International Symposium on Empirical Software Engineering, p. 356--363. ACM, New York 2006

[4] L. Crispin *"Driving Software Quality: How Test-Driven Development Impacts Software Quality"*. In: IEEE Software, vol. 23 / 2006, p. 70--71. IEEE Computer Society Press, Los Alamitos 2008

[5] F. Fraikin, M. Hamburg, S. Jungmayr, T. Leonhardt, A. Schönknecht, A. Spillner and M. Winter *„Die trügerische Sicherheit des grünen Balkens"* Objekspektrum, vol. 1, p. 25--29, 2004

[6] D. S. Janzen and S. Hossein *„Does Test-Driven Development Really Improve Software Design Quality?"* In: IEEE Software, Vol. 25, p. 77—84, IEEE Computer Society Press, Los Alamitos, 2008

[7] P. Kroll, P. Krutchten and G. Booch *„The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP"*, Addison-Wesley Longman, Amsterdam, 2003

[8] T. Linz, H. Schäfter and A. Spillner *"Software Testing Foundations: A Study Guide for the Certified Tester Exam - Foundation Level - ISTQB compliant"*, Rocky Nook, 2007

[9] A. Rendell "*Effective and Pragmatic Test Driven Development"* In: Proceedings of the AGILE 2008 Conference, p. 298--303. IEEE Computer Society, Washington 2008

[10] R. S. Sangwan and P. A. Laplante *"Test-Driven Development in Large Projects"* In: IT Professional, vol. 8 / 2006, p. 25--29. IEEE Computer Society Press, Los Alamitos 2006

[11] K. Schwaber and M. Beedle *"Agile Software Development with Scrum"*, Pearson Studium, 2008

[12] M. Siniaalto and A. Pekka *"A Comparative Case Study on the Impact of Test-Driven Development on Program Design and Test Coverage"* In: Proceedings of the ESEM, p. 275—284, IEEE Computer Society, Washington, 2007