bbv
MAKING VISIONS WORK.

BOOKLET

QUALITY-DRIVEN DEVELOPMENT IN BROWNFIELD-PROJEKTEN

PROFITIEREN SIE VON UNSERER ERFAHRUNG!

Kontakt Schweiz
bbv Software Services AG
Blumenrain 10
6002 Luzern
Telefon: +41 41 429 01 11
E-Mail: info@bbv.ch

Kontakt Deutschland
bbv Software Services GmbH
Agnes-Pockels-Bogen 1
80992 München
Telefon: +49 89 452 438 30
E-Mail: info@bbv.eu

INHALT

1	Einleitung	5
2	Ziel: Qualität	7
3	Software-Lifecycle	10
4	Refactoring	14
	Broken Window	15
5	Automatisierte Tests	16
	Unit-Test	18
	Acceptance-Test	18
5.1	Isoliertes Unit-Testing	19
	Test-Double	19
6	Brownfield-Projekte	21
6.1	Das Brownfield-Dilemma	22
7	Wie bringt man Qualität in Legacy-Code?	24
7.1	Build-Umgebung	25
7.2	Testumgebung und -werkzeuge	25
	xUnit-Test-Framework	26
7.3	Automatisierung	26
7.4	Schnelles Feedback	27
7.5	Abhängigkeiten aufbrechen	27
7.6	Unit-Test schreiben	29
	Charakterisierungstests	29
	Spezifikationstest	30
7.7	Testabdeckung	30
7.8	Code ändern und Refactoring vornehmen	31
8	Techniken und Praktiken	32
8.1	Defect-Driven Testing (DDT)	33
8.2	Mikado-Methode	33

8.3	Testgetriebene Entwicklung	34
8.4	Dependency-Injection	36
8.5	Vier-Augen-Prinzip	37
8.6	Kollektive Verantwortung	38
8.7	Clean-Code-Boy-Scout-Regel	38
9	Fazit	39
10	Anhang	42
10.1	Autor	43
10.2	Literaturverzeichnis	43

1 EINLEITUNG

Gerade bei älteren Embedded Projekten wird die Qualität von Software oft nur an ihren externen Faktoren wie Funktionalität, Zuverlässigkeit und Performance gemessen. Viele Langzeitprojekte werden mit der Zeit zunehmend komplexer. Sie sind früher oder später mit dem Problem konfrontiert, dass der Unterhalt, die Verifikation und die Fehlersuche sowie Erweiterungen und Änderungen aufwendig werden und mit dem Risiko verbunden sind, dass neue Fehler eingebaut werden.

Moderne, agile Entwicklungsprozesse legen Wert auf qualitätsgetriebene Entwicklung (QDD) mit TDD, Continuous Integration, Clean Code und weiteren bewährten Praktiken, also mit extensiven automatisierten Tests und kontinuierlichem Refactoring. Das funktioniert hervorragend für sogenannte Greenfield-Projekte, die ganz neu entwickelt werden.

In diesem Artikel zeigt der Autor auf, dass es sich trotz grossem Initialaufwand auch für Brownfield-Projekte lohnt, den Schritt Richtung QDD zu wagen und den Fokus vermehrt auf Unit-Testing, Refactoring und kurze Entwicklungszyklen mit schnellem Qualitätsfeedback zu richten. Der Gewinn ist eine während des ganzen Software-Lifecycles aktuelle, verständliche und testbare Software hoher Qualität, der man vertraut und die man mit geringem Risiko anpassen und erweitern kann.

Im Artikel wird durchgehend die männliche Form für Projektleiter und Entwickler benutzt, es sei hier aber ausdrücklich darauf hingewiesen, dass immer auch die Projektleiterinnen und Entwicklerinnen angesprochen sind.

2 ZIEL: QUALITÄT

«The bitterness of poor quality remains long after the sweetness of a low price is forgotten.» Benjamin Franklin

Spricht man von der Qualität einer Software, sollte man zwischen externer und interner Qualität unterscheiden. Die externe Qualität eines Produkts ist die Summe aller Qualitätsaspekte, die von einem Benutzer des Systems wahrgenommen werden. Sie umfasst die Funktionalität, die Performance, das Aussehen und die Bedienung (Look & Feel), die Zuverlässigkeit (Reliability) und die Compliance (Erfüllung der Policen, Regeln und Vorschriften).

Die Qualität gerade von Software ist aber stark durch die interne, vom Benutzer nicht direkt bewertbare Qualität bestimmt, d. h., wie die Subsysteme, die Komponenten, die Klassen und die Methoden aufgebaut sind. Die Erfahrung zeigt, dass schlechte interne Qualität früher oder später die externe Qualität beeinträchtigt. Bei mittel- und langfristigen Projekten hat der Mangel an interner Qualität entweder einen direkten negativen Einfluss auf die externe Qualität, oder er führt zu mehr Aufwand, um die externe Qualität auf hohem Niveau zu erhalten.

Die interne Qualität ist durch folgende Hauptfaktoren bestimmt:

- Lesbarkeit: Wie verständlich ist ein Stück Code? Softwareentwickler verbringen den grössten Teil ihrer Arbeit mit Lesen von Code (Atwood, 2006). Es ist deshalb von grosser Bedeutung, dass Code verständlich geschrieben ist und der Leser schnell erkennt, wozu eine Komponente dient, was eine Methode macht und wie sie mit dem Rest der Software interagiert. Verständlicher Code spart viel Projektzeit, weil oft verschiedene Entwickler immer wieder denselben Code lesen.
- Wartbarkeit: Wie einfach kann eine bereits ausgelieferte Software geändert werden, um Defekte zu korrigieren oder einer neuen Umgebung anzupassen? Wie einfach kann ein System konfiguriert werden?
- Änderbarkeit: Anforderungen ändern sich. Es ist deshalb wichtig, dass man die bestehende Funktionalität einer Software einfach neuen Anforderungen anpassen kann und obsolete Funktionen entfernen kann.
- Erweiterbarkeit: Anforderungen ändern sich nicht nur, es kommen auch neue Anforderungen hinzu. Mit einfach erweiterbarer Software ist man flexibler und kann schnell auf neue Anforderungen reagieren.

- Testbarkeit: Funktionalität muss man verifizieren können. Ein sehr wichtiges Qualitätsmerkmal ist deshalb, wie einfach die Software auf verschiedenen Stufen getestet werden kann.

3 SOFTWARE-LIFECYCLE

«Time goes by, reputation increases, ability declines.» Dag Hammarskjöld

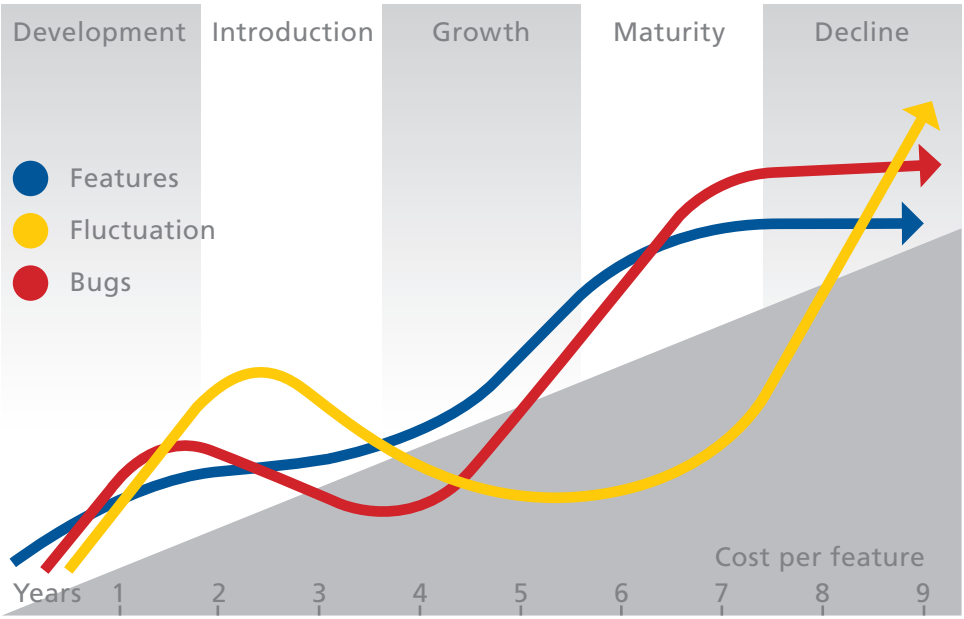


Abbildung 1: Software-Lifecycle. Adaptiert von (Peter & Ehrke, 2009)

Ein Software-Lebenszyklus kann in fünf Abschnitte aufgeteilt werden: Entwicklung (Development), Markteinführung (Introduction), Wachstum (Growth), Reife (Maturity) und Rückgang (Decline). Jeder dieser Abschnitte kann durch drei voneinander abhängige Indikatoren charakterisiert werden (Abbildung 1):

- Funktionalität: Anzahl implementierter Features
- Qualität: Anzahl Bugs
- Mitarbeiterfluktuation

Die Performance ist meist während der Wachstumsphase und am Anfang der Reifephase am besten. Es werden viele neue Features implementiert und die Anzahl Bugs ist einigermassen stabil und

überschaubar. Wird der internen Qualität keine oder zu wenig Beachtung geschenkt, verschlechtert sich aber die Situation mit fortschreitender Lebensdauer der Software drastisch (Peter & Ehrke, 2009):

- Durch die ungenügende Testabdeckung wächst das Risiko, Code zu ändern. Die Angst, bestehende Funktionalität zu verändern, hindert die Entwickler daran, das nötige Refactoring vorzunehmen.
- Um neue Funktionalität zu implementieren, müssen Workarounds gemacht werden, und die neuen Features sind schlecht in die bestehende Software integriert.
- Jeder Bugfix führt zu neuen Fehlern.
- Zunehmender Mitarbeiterwechsel wirkt sich auf die Softwarearchitektur aus. Das Verständnis für die Originalarchitektur geht verloren, und neue Konzepte, die nicht mit den alten zusammenpassen, werden eingeführt. Die neuen und alten Konzepte koexistieren im Code und machen ihn unverständlich und erschweren die Entkopplung.
- Die Anzahl Code-Smells[1], wie doppelter oder unbenutzter Code, wächst.

Als Konsequenz wird es nahezu unmöglich, neue Funktionalität mit vertretbarem Aufwand zu implementieren. Die möglichen Nebeneffekte sind unüberschaubar und das Projekt bleibt schliesslich vollständig stecken. Für die Langlebigkeit einer Software ist also die interne Qualität entscheidend.

[1] Mit Code-Smell bezeichnet man ein Konstrukt, das eine Überarbeitung des Programm-Quelltextes nahelegt und möglicherweise auf ein tiefer liegendes Problem hinweist. Der Begriff stammt von Kent Beck und fand durch die Verwendung von Martin Fowler (Fowler, Refactoring: Improving the Design of Existing Code, 1999) weite Verbreitung.

Den allmählichen Zerfall einer Software aufzuhalten, ist einer der Gründe, warum die agile Softwareentwicklung grossen Wert auf die qualitätsgetriebene Entwicklung legt und auf Techniken wie TDD (Test-Driven Development) und Refactoring setzt sowie Clean Code und objektorientierte Design-Prinzipien befolgt[2].

[2] Siehe Martin, Clean Code: A Handbook of Agile Software Craftsmanship, 2009, und Martin, Principles Of OOD, 2005.

4 REFACTORING

«Refactoring: a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour.» (Fowler, Refactoring: Improving the Design of Existing Code, 1999)

Die einzige existierende Massnahme, um dem Qualitätsverlust bei steigender Softwarekomplexität entgegenzuwirken, ist stetiges Refactoring (Westphal, 2006).

Durch Refactoring versucht man, die Codestruktur zu verbessern und auf eine einfachere Form zu bringen, ohne dabei das beobachtbare Programmverhalten zu verändern. Refactoring sollte dabei Hand in Hand mit der Entwicklung neuer Funktionalität stattfinden. Den Entwicklungsphasen folgen immer mal wieder Refactoringphasen mit dem Ziel, die neu hinzugefügten Komplexitäten wieder etwas aufzulösen. Oft braucht es auch im Vorfeld eine Refactoringphase, um neue Features überhaupt einfacher implementieren zu können.

Broken Window

In der Softwareentwicklung bestätigt sich leider oft die «Broken-Window»-Theorie (Wilson & George, 1982): Ein zerbrochenes Fenster in einem Gebäude, das nicht repariert wird, zieht innerhalb kurzer Zeit die Zerstörung weiterer Fenster nach sich. Ein nicht wieder instand gesetztes Fenster ist ein Zeichen dafür, dass an diesem Ort niemand daran Anstoss nimmt. Code-Smells, die nicht behoben werden, können schnell dazu führen, dass wir als Entwickler selber nachlässig werden. Oft werden Code Smells auch kopiert und an andere Stellen übernommen, und so vermehren sie sich überproportional.

Wann immer man auf Code-Smells stösst, sollte man sich deshalb nicht scheuen und ein Refactoring vornehmen. Es gibt keinen besseren Zeitpunkt, als wenn man gerade darüber stolpert (siehe auch Abschnitt 8.6).

Refactoring birgt immer das Risiko, dass sich die bestehende Funktionalität verändert. Um dieses Risiko zu reduzieren, muss die Funktionalität mit Regressionstests in Form automatisierter Tests überprüft werden können.

5 AUTOMATISIERTE TESTS

«A unit test is a piece of code written by a developer who exercises a very small, specific area of functionality in the code being tested.» (Hunt & Thomas, Pragmatic Unit Testing In C# with NUnit, 2007)

Jeder professionelle Softwareentwickler ist sich der Wichtigkeit von Softwaretests bewusst, und doch wird ihnen meist nicht genügend Aufmerksamkeit geschenkt.

In der klassischen Softwareentwicklung werden Tests parallel zum testenden System und unabhängig vom ihm entwickelt. Oft sogar nach ihm. Zudem beschränkt sich das Testen der Software meist nur auf Integrations- und Systemtests am Ende einer Entwicklungsiteration. Die Folgen sind eine ungenügende Testabdeckung und schwierig zu wartende Tests. Solche Tests dienen zur Messung der Qualität und nicht als Mittel, um Qualität in das Produkt zu bringen.

Die Gründe für fehlende Tests sind vielfältig: Zeitdruck, mangelnde Testbarkeit des Systems aufgrund des Designs und der komplexen Abhängigkeiten oder einfach nur Nachlässigkeit und mangelnde Disziplin der Programmierer bei der Testerstellung.

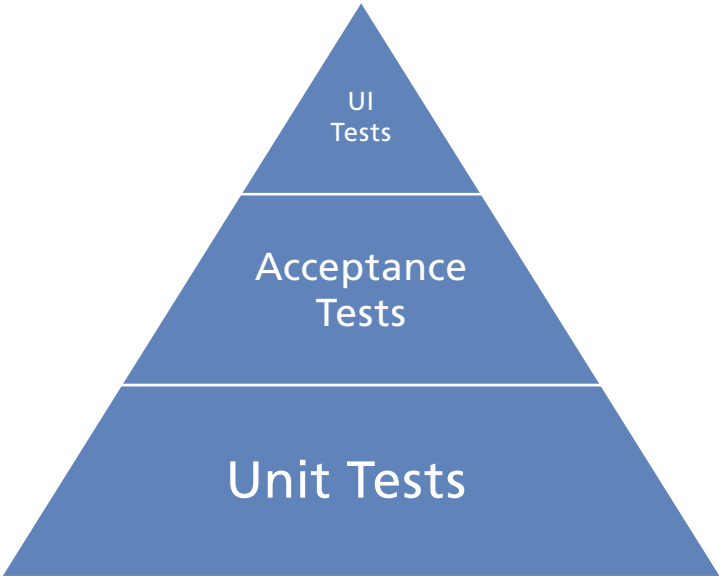


Abbildung 2:
Testautomatisierungs-
pyramide

Eine erfolgreiche Teststrategie besteht aus automatisierten Tests auf mindestens drei Stufen der sogenannten Testautomatisierungspyramide[3] (Abbildung 2). Die Basis und den grössten Anteil der Testpyramide sollten dabei die Unit-Tests einnehmen (Cohn, 2010).

Unit-Test

Unit-Tests werden ausgeführt, um zu verifizieren, dass ein Stück Code sich so verhält, wie der Entwickler es erwartet. Es bleibt dabei offen, ob dieses Verhalten den eigentlichen Erwartungen des Endkunden entspricht. Dazu hat man die Acceptance-Tests.

Unit-Tests sind also vom Entwickler für den Entwickler und ermöglichen, als Regressionstests die Funktionalität vor und nach jeder Änderung zu verifizieren.

Acceptance-Test

Im agilen Umfeld werden Story-Acceptance-Tests benutzt, um die Businesslogik, also eine ganze Funktionalität zu testen. Sie heissen Acceptance-Tests, weil sie dazu dienen, eine Funktionalität oder ein Teil davon als erledigt zu akzeptieren. Acceptance-Tests werden typischerweise vom Business-Analysten in Zusammenarbeit mit der QA und den Entwicklern geschrieben oder spezifiziert und bestehen aus System- und/oder Subsystemtests. Idealerweise haben die Acceptance-Tests die Form ausführbarer Beispielszenarien oder Spezifikation in einer (pseudo-)natürlichen domänenspezifischen Sprache (Adzic, 2011).

[3] In der Literatur und im Web finden sich verschiedene Test-Automatisierungspyramiden mit meist drei, aber auch mehr Stufen und unterschiedlichen Bezeichnungen der Stufen. Über die Unit-Tests als Basis der Pyramide sind sich aber alle seriösen Autoren einig. Siehe http://blog.goneopen.com/2010/08/test-automation-pyramid-review/

Für mehr Information über automatisiertes Testen siehe auch (Müller U., 2010).

5.1 ISOLIERTES UNIT-TESTING

Gute Unit-Tests sollten folgende Kriterien erfüllen:

- Sie sind schnell ausgeführt.
- Sie erlauben eine schnelle Lokalisierung eines Problems.

Unit-Tests sollten deshalb jeweils eine Funktionseinheit in Isolation von anderen Einheiten testen. Leider führen aber oft schon einfache Abhängigkeiten im Programmcode dazu, dass sich Klassen nicht isoliert, sondern nur in einer eng zusammenhängenden Gruppe testen lassen, was zu aufwendigen Testumgebungen führt. In eingebetteten Systemen erschweren Zugriffe auf die Hardware das Testen, weil sie eine funktionierende Hardware oder eine aufwendige Simulation voraussetzen. Durch testgetriebene Entwicklung und das dadurch entstehende testbare Design können diese Abhängigkeiten frühzeitig erkannt und teilweise vermieden werden. Minimale Abhängigkeiten zwischen Klassen werden aber immer vorhanden sein, und doch sollten sich im effektiven Unit-Test Softwarestrukturen unabhängig voneinander testen lassen. Einerseits weil das Verhalten der Komponenten, die von der zu testenden Klasse benutzt werden, nicht oder nur mit grossem Aufwand innerhalb des Testfalles beeinflusst werden kann, andererseits weil wir nicht unnötig fremden Code testen wollen. Die allfällige Fehlersuche und Problembehebung kann dadurch lokaler in einer einzelnen Komponente stattfinden.

Test-Double

Einen Ausweg aus dem Abhängigkeitsproblem in Tests bieten sogenannte testspezifische Doubles wie Stub- oder Mock-Objekte.

Stubs liefern vorgegebene Rückgabewerte zurück und können auch genutzt werden, um Informationen zu den Aufrufen zu speichern. Mock-Objekte unterscheiden sich von Stubs insofern, als sie mit Erwartungen vorprogrammiert werden können. Sie verifizieren, dass sie so benutzt wurden, wie man es erwartet (Meszaros, 2007). Hilfreich ist es, wenn die gesamte Testlogik an einem Ort implementiert ist: also nicht gewisse Logik im Test-Driver und andere in den Test-Doubles. Bei Mock-Objekten, wie sie z. B. in Google Mock vorkommen, ist dies gegeben, d. h., die Erwartungen werden im Test-Driver gesetzt.

6 BROWNFIELD-PROJEKTE

«... a project, or codebase, that was previously created and may be contaminated by poor practices, structure, and design but has the potential to be revived through comprehensive and directed refactoring.» (Belcham, 2009)

TDD (Abschnitt 8.3), Clean Code und Refactoring eignen sich gut für sogenannte Greenfield-Projekte, die man neu startet und from scratch zu entwickeln beginnt. Meist leiden aber gerade bereits weit fortgeschrittene, ältere Projekte an mangelnder interner Qualität.

Projekte mit einer umfangreichen, komplexen Codebasis, die schwierig zu erweitern und zu pflegen ist und mit wenigen oder keinen Unit-Tests abgedeckt ist, nennt man oft Legacy-Code (Feathers, 2005). In Analogie zu Greenfield-Projekt ist der Begriff Brownfield-Projekt aber passend. Also Code mit Altlasten (Legacy), die man zuerst wegräumen muss, um sinnvoll und effizient daran weiterarbeiten zu können.

6.1 DAS BROWNFIELD-DILEMMA

Brownfield-Projekte leiden darunter, dass nicht von Anfang an der Fokus auf die interne Qualität und Evolvierbarkeit[4] gelegt wurde. Jeder Workaround, um eine neue Funktion hinzuzufügen oder um Defekte zu reparieren, verschlechtert die Qualität und verschärft das Problem der schlechten Wartbarkeit überproportional.

Wie bereits erwähnt ist der einzige Ausweg ein ausgiebiges Refactoring der bestehenden Codebasis. Gerade bei Brownfield-Projekten ist aber das Risiko, bereits implementierte Funktionalität zu verändern, sehr hoch. In schlecht strukturiertem Code mit vielen Abhängigkeiten können selbst lokale Änderungen zu unerwarteten Problemen an scheinbar nicht betroffenen Codestellen führen. Das hohe Risiko, gepaart mit dem fehlenden Verständnis des Legacy-Codes, ist der Grund, warum sich die Entwickler nicht an das Refactoring wagen. Wird dieser Teufelskreis nicht durchbrochen, kommt das Projekt früher oder später zum Stillstand. Die Kosten für neue Funktionalität und Anpassungen steigen exponentiell.

[4] Siehe «Clean-Code-Developer-Wertesystem» http://www.clean-code-developer.de/Wertesystem.ashx.

Um das Risiko des Refactorings zu reduzieren, braucht es ein gutes Sicherheitsnetz, bestehend aus automatisierten Tests, Versionskontrollsystem und Continuous Integration. Das Dilemma bei Brownfield-Projekten ist aber, dass automatisierte Unit-Tests aufgrund des Softwaredesigns schwierig zu implementieren sind. Um die Unit-Tests zu implementieren, ist ein Refactoring erforderlich, mit dem Risiko, dadurch die bestehende Funktionalität zu verändern.

Um das Risiko beim Refactoring zu reduzieren, brauchen wir automatisierte Unit-Tests. Um solche Tests zu implementieren, ist aber oft ein Refactoring nötig.

7 WIE BRINGT MAN QUALITÄT IN LEGACY-CODE?

«A journey of a thousand miles begins with a single step.» Lao-tzu

Wie gezeigt kann man langfristige Projekte nur vor dem Stillstand retten, indem man den Fokus auf die interne Qualität legt und aktiv Massnahmen einleitet, um die Qualität zu erhöhen.

Um das Risiko beim Refactoring von Legacy-Code zu verringern, muss zuerst ein möglichst grosses Sicherheitsnetz aufgespannt werden. Das Sicherheitsnetz sollte aus einer Source-Code-Versionsverwaltung, einer Build-Maschine und automatisierten Tests bestehen.

7.1 BUILD-UMGEBUNG

Wichtig für die qualitätsgetriebene Entwicklung ist häufige Integration und schnelles Feedback. Nach jedem Refactoring sollte der Entwickler schnell die Bestätigung haben, dass seine Änderungen auch im Zusammenspiel mit den anderen Komponenten nicht im Konflikt stehen und sich die Funktionalität der Software nicht geändert hat.

Falls nicht bereits vorhanden, sollte deshalb als erster Schritt eine Build-Umgebung mit Versionsverwaltung aufgebaut werden, die das automatisierte Starten der Tests ermöglicht. Idealerweise baut man ein Continuous-Integration (CI)-System auf, das automatisch bei jedem Code-Check-in das Projekt kompiliert und die automatisierten Tests laufen lässt.

Auf dem Markt gibt es verschiedenste kommerzielle sowie frei erhältliche Versionsverwaltungs- und CI-Systeme wie z. B. git, FinalBuilder und Jenkins (Hudson). (Dolder, 2012) und (Büchi, 2012) geben eine Beschreibung der Konzepte und Anleitungen zur Umsetzung.

7.2 TESTUMGEBUNG UND -WERKZEUGE

Bevor mit dem Refactoring begonnen werden kann, müssen erste Feature-Tests geschrieben oder die bestehende Testabdeckung erweitert werden. Das bestehende Design erlaubt oft noch nicht, isolierte Unit-Tests zu schreiben. Es sollte aber möglich sein,

automatisierte Tests auf Acceptance (Integration)-Stufe zu schreiben, um eine bereits bestehende Funktionalität zu testen. Sind manuelle Tests vorhanden, sollte man versuchen, diese zu automatisieren.

Für alle gängigen Sprachen gibt es Test-Frameworks, die es mit mehr oder weniger Aufwand ermöglichen, automatisierte Tests zu schreiben.

xUnit-Test-Framework

Das kostenlose, frei verfügbare xUnit-Framework, ursprünglich von Kent Beck in Smalltalk geschrieben (Beck, Simple Smalltalk Testing: With Patterns) und später auf Java portiert (JUnit), ist weit verbreitet. Es gibt für die meisten gängigen Sprachen eine Implementation. CppUnit[5] von Michael C. Feather ist eine bekannte und häufig verwendete C++-Portierung von JUnit. Andere xUnit-Frameworks sind z. B. CppUnitLite (vom CppUnit Author M. C. Feather) und Google Test[6]. Gerade wenn man auch ein Mock-Framework benutzen möchte, bietet sich Google Mock[7], welches Google Test integriert, als Mocking- und Test-Framework an.

7.3 AUTOMATISIERUNG

Ein essenzieller Aspekt bei der qualitätsgetriebenen Entwicklung ist die Automatisierung der Build- und Integrationsschritte. Ziel sollte es sein, dass ein Entwickler mit geringstem Aufwand, z. B. einer einzigen Zeile auf der Konsole, seinen Source-Code in die Versionsverwaltung einchecken kann und der Build sowie alle Tests automa-

[5] CppUnit Documentation http://cppunit.sourceforge.net/doc/lastest/index.html
[6] Google C++ Testing Framework http://code.google.com/p/googletest/
[7] C++ Mocking Framework http://code.google.com/p/googlemock/

tisiert gestartet werden. Nur wenn es für einen Entwickler keinen Mehraufwand bedeutet, regelmässig zu integrieren und die Tests laufen zu lassen, wird er es auch machen.

7.4 SCHNELLES FEEDBACK

Wie bereits erwähnt, ist für einen Entwickler ein schnelles Feedback zum Erfolg seiner Änderungen äusserst wichtig. Je länger es dauert, bis er das Resultat erhält, umso schwieriger ist es für ihn, sich wieder in den Kontext der damaligen Aufgabe einzuarbeiten. Hilfreich sind zudem kleine Iterationsschritte. Je kleiner die Änderungen zwischen den Feedbacks, desto einfacher ist es, eine Problemstelle zu lokalisieren und wenn nötig die Änderungen rückgängig zu machen.

Um ein schnelles Feedback zu erlauben, müssen die Tests in kurzer Zeit das Ergebnis liefern. Die Komponenten und Integrationstests in Brownfield-Projekten dauern aber oft lange. Das führt dazu, dass sie nur selten und unregelmässig ausgeführt werden. Es ist sinnvoll, die schnell ausführbaren Unit-Tests von den aufwendigen und langsamen Integrationstests zu trennen. Die Unit-Tests können nach jedem Build und auch individuell vom Entwickler jederzeit gestartet werden und erlauben ein schnelles Feedback. Die Komponenten, Integrations- und UI-Tests werden seltener, aber auf einer regelmässigen Basis, z. B. nach einem Nightly Build, gestartet.

7.5 ABHÄNGIGKEITEN AUFBRECHEN

In Brownfield-Projekten sind komplexe Abhängigkeiten zwischen Klassen die grösste Hürde beim Implementieren von Unit-Tests. Damit man eine Klasse testen kann, muss man zuerst oft verschiedene andere Klassen instanziieren, teilweise fast das ganze System. Das macht Testen aufwendig und isoliertes Testen nahezu unmöglich.

Ist ein grosser Teil der Funktionalität durch Integrationstests (Feature-Tests) abgedeckt, müssen durch konservatives Refactoring mit wenig Risiko die Abhängigkeiten so weit aufgelöst werden, dass Unit-Tests mit Zuhilfenahme von Test-Doubles geschrieben werden können.

Der erste Schritt besteht darin, die Klasse, die man testen will, in einem Unit-Test zu instanziieren. Der Compiler hilft dabei, zu identifizieren, was man für andere Klassen braucht, um erfolgreich zu sein.

Objektinstanzen, die dem Konstruktor als Parameter übergeben werden müssen, können durch Test-Doubles ersetzt werden. Probleme bereiten versteckte Abhängigkeiten, d. h. Ressourcen, die der Konstruktor benutzt, die ihm aber nicht übergeben werden und auf die man in einem Test nicht einfach zugreifen kann. Es gibt verschiedene Refactoring-Methoden, die dieses Problem lösen, z. B. Parametrisierung des Konstruktors, d. h., den Konstruktor so zu erweitern, dass alle Objekt übergeben werden können und nicht selbst kreiert werden (siehe auch 8.4). Eine andere Möglichkeit ist das Einführen einer Factory-Methode, die das Instanziieren dieser Objekte übernimmt[8]. Ziel ist es, alle Abhängigkeiten, die zu unerwünschten Nebeneffekten im Konstruktor führen können, durch Test-Doubles unter die Kontrolle des Testcodes zu bringen.

Wenn wir Abhängigkeiten aufbrechen mit dem Ziel, einen Unit-Test schreiben zu können, müssen wir oft unsere Vorstellung von schönem Code etwas dehnen. Gewisse Techniken, um Abhängigkeiten aufzulösen, hinterlassen den Code aus Designsicht vielleicht etwas suboptimal. Sie machen den Code aber besser, weil sie Unit-Tests

[8] siehe (Feathers, 2005) Kapitel 9 für Beispiele und Hilfestellungen.

ermöglichen. Feathers vergleicht es mit den Skalpellschnitten bei einem chirurgischen Eingriff: «There might be a scare left in your code after your work, but everything beneath it can get better.» (Feathers, 2005). Haben wir dann einen Test, kann der Code durch Refactoring wieder verschönert werden.

7.6 UNIT-TEST SCHREIBEN

Ist es möglich, die zu testende Klasse in einem ersten Konstruktionstest zu instanziieren, können wir daran gehen, die Unit-Tests zu schreiben. Sie geben uns beim Modifizieren des Quellcodes die nötige Sicherheit. Um eine einzelne Methode oder ein Szenario zu testen, wird es wahrscheinlich nötig sein, weitere Abhängigkeiten aufzulösen. Dynamische Bindung ist aus Testsicht der statischen vorzuziehen. Die zu testende Klasse muss deshalb evtl. angepasst werden und z. B. durch Setter-Methoden erweitert werden, damit mittels Dependency-Injection Objekte, von denen eine Methode abhängt, durch Test-Doubles ersetzt werden können (siehe Abschnitt 8.4).

Charakterisierungstests

Die Unit-Tests, die wir für bereits bestehende Funktionalität in Legacy-Code schreiben, sind sogenannte Charakterisierungstests, d. h., sie beschreiben das aktuelle Verhalten und schützen den Code vor ungewollten Änderungen dieser Funktionalität. Das aktuelle Verhalten, wie es durch den Test beschrieben wird, kann durchaus fehlerhaft sein. Charakterisierungstests haben nicht das Ziel, Fehler aufzudecken, sondern dienen dazu, neue Fehler bei Änderungen am Code zu vermeiden, d. h. Fehler, die auftreten, weil sich der Code nicht mehr gleich verhält[9].

[9] siehe auch Alberto Savoia's Weblog (Savoia, 2007)

Zusätzlich zur Funktion als Regressionstests haben Charakterisierungstests den Vorteil, dass sie das Verhalten der getesteten Methode auf anschauliche Weise dokumentieren und uns helfen, den Code, den wir ändern müssen, zu verstehen.

Charakterisierungstests sollten nicht mit Black-Box-Tests verwechselt werden. Es ist nötig, den Code, den wir testen, anzuschauen, damit auf die richtigen Werte und Zustände getestet werden kann und für die Testein- und -ausgaben Äquivalenzklassen[10] gebildet werden können.

Spezifikationstest

Für neue Funktionalität schreiben wir Unit-Tests, die das erwartete Verhalten beschreiben und verifizieren. Weil sie nicht das System as is, sondern as intended beschreiben, sollten sie vor der eigentlichen Implementation geschrieben werden (siehe 8.3).

7.7 TESTABDECKUNG

«Wie viel Tests muss ich schreiben?» ist eine häufig gestellte Frage und Anlass zu heftigen Debatten. Aus der QDD-Sicht sollte klar eine Branch-Coverage gegen 100% das Ziel sein, d. h., jede mögliche Verzweigung wurde mindestens einmal durchlaufen. Wenn es nicht möglich ist, einen Codeblock innerhalb eines isolierten Unit-Tests mit Test-Doubles zu erreichen, ist die Chance hoch, dass der Block nicht (mehr) gebraucht wird. In Legacy-Code können wir dieses Ziel realistischerweise aber nur für Code, der geändert wird, anstreben (siehe Abschnitt 8.2).

[10] http://de.wikipedia.org/wiki/Äquivalenzklassentest

Code-Coverage-Statistiken sind ein unverzichtbares Hilfsmittel, um die Qualität der Tests zu beurteilen und ungenügend getestete Komponenten zu identifizieren. Sie helfen dabei, obsoleten und unerreichbaren Code ausfindig zu machen.

Es gibt einige kommerzielle und frei erhältliche Code-Coverage-Tools, die einfach in eine Build- und Testumgebung zu integrieren sind. Ein frei erhältliches C++ Code-Coverage-Tool ist gcov/LCOV[11].

7.8 CODE ÄNDERN UND REFACTORING VORNEHMEN

Haben wir genügend gute Unit-Tests für den Code, an dem wir Änderungen vornehmen müssen, können diese nun in Angriff genommen werden. Ist ein umfangreicheres Refactoring nötig, haben wir jetzt die nötigen Tests, die das Risiko, die bestehende Funktionalität zu verändern, minimieren.

[11] http://ltp.sourceforge.net/coverage/lcov.php

8 TECHNIKEN UND PRAKTIKEN

«There's a big difference between saying, ‹Eat an apple a day› and actually eating the apple.» Kathy Sierra (Sierra, 2006)

Um den Fokus auf die interne Qualität zu richten, haben sich verschiedene Techniken und Praktiken z. B. aus dem eXtrem-Programming (XP)-Umfeld bewährt und können durchaus auch in «traditionellen», nicht agilen Brownfield-Projekten genutzt werden. Andere Techniken wie die Mikado-Methode und die Method-Use-Regel eignen sich speziell für Legacy-Code, um Schritt für Schritt die Qualität zu erhöhen.

8.1 DEFECT-DRIVEN TESTING (DDT)

Die Zeit reicht selten aus, um für stabilen Code vorsorglich Unit-Tests zu schreiben. Ein gutes Mittel, um trotzdem allmählich die Codequalität zu verbessern, ist Defect-Driven Testing, kurz DDT (Richardson, 2010). Wann immer ein Fehler auftritt, schreibt man einen Test, der den Fehler reproduziert. DDT ist auch dem Management einfach zu verkaufen, denn ein Fehler ist ein offensichtliches Problem und bedeutet sowieso Aufwand.

Wenn man einen Test geschrieben hat und der Fehler behoben ist, kann er sich nicht mehr unbemerkt erneut einschleichen. Denn leider ist es nicht ganz ungewöhnlich, dass derselbe Fehler mehr als einmal im Laufe der Softwareentwicklungszeit auftaucht.

8.2 MIKADO-METHODE

Gerade in Brownfield Projekten haben wir oft das Problem, dass das aktuelle System auch während der Weiterentwicklung funktionsfähig sein muss und die Entwicklung nicht durch langwierige Umbauten blockiert sein sollte. Das Refactoring sollte deshalb in möglichst kleinen Einheiten durchgeführt und regelmässig integriert werden können. Die Mikado-Methode (Ellnestam, 2014) ist eine genial simple Methode, die dabei hilft, ein bestehendes System in ein gewünschtes neues System zu wandeln, ohne dabei das existierende für längere Zeit unbrauchbar zu machen. Die Methode hilft, in kleinen Schritten das Refactoring vorzunehmen, und man vermeidet, sich in grossen unüberschaubaren Änderungen zu verlieren[12].

[12] Ausführliche Informationen zur Methode findet man im Buch von Ellnestam und Brolund (Ellnestam, 2014) und auf der Mikado Method website (https://mikadomethod.wordpress.com)

8.3 TESTGETRIEBENE ENTWICKLUNG

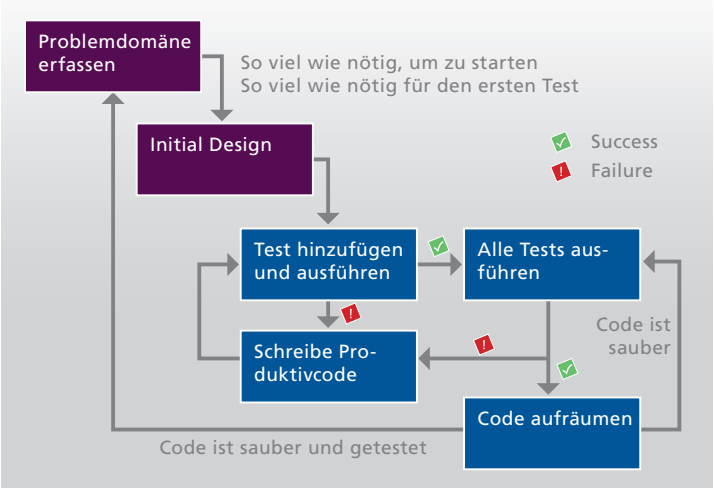


Abbildung 3:
Test-Driven Development

In der agilen Softwareentwicklung hat sich die testgetriebene Entwicklung (Test-Driven Development, TDD) als Methode zur qualitätsbewussten Softwareherstellung bewährt.

Bei der testgetriebenen Entwicklung erstellt der Programmierer die Tests konsequent unmittelbar vor den zu testenden Komponenten. Die dazu erstellten Unit-Tests sind deshalb sogenannte Grey-Box-Tests statt klassischer White-Box-Tests, weil sie zwar wie White-Box-Tests ebenfalls von den gleichen Entwicklern wie das zu testende System geschrieben werden, aber zum Zeitpunkt der Testimplementierung die Interna der zu testenden Komponente noch unbekannt sind. Abbildung 3: Test-Driven Development zeigt eine

leicht modifizierte schematische Darstellung des TDD-Zyklus wie von Kent Beck in (Beck, 2002) beschrieben.

Unter Softwareentwicklern gehen die Meinungen über den Nutzen von TDD auseinander. Während die Test-First-Verfechter überzeugt sind, dass durch TDD die Softwarequalität und auch die Produktivität gesteigert wird, kritisieren die Gegner die hohe Test-code-to-production-code-Rate und sprechen von verminderter Produktivität.

Weitgehend einig ist man sich aber, dass durch gute Unit-Tests die Qualität der Software im Allgemeinen erhöht werden kann (Nagappan, Maximilien, Bhat, & Williams, 2008). Es spielt dabei keine grosse Rolle, ob man einen Test-First (TDD)- oder einen klassischen Test-Last-Ansatz verfolgt (Erdogmus, Morisio, & Torchiano, 2005). In der Praxis hat sich aber gezeigt, dass Unit-Tests im klassischen Ansatz eher vernachlässigt werden. Ist der produktive Code einmal geschrieben, hat der Entwickler wenig Ansporn, noch einen Unit-Test dafür zu schreiben. Ausserdem ist die Qualität der Tests beim TDD-Ansatz im Allgemeinen höher, und TDD hat den Vorteil, dass die Software by Design testbar ist, was beim klassischen Ansatz meist ein Problem darstellt. Weiter erzeugt TDD kurze Feedbackloops für den Entwickler und unterstützt den iterativen Entwicklungsprozess.

Es hat sich auch gezeigt, dass das Argument der verminderten Produktivität gegen TDD meist nicht haltbar ist (Müller & Padberg, 2003). Obwohl der Initialaufwand für TDD grösser ist, da zusätzlicher Testcode geschrieben wird, wirkt sich auf längere Sicht Test-First positiv auf die Produktivität aus. TDD resultiert in besser strukturiertem Code mit weniger Abhängigkeiten, und Code mit extensiven Unit-Tests führt zu weniger und kürzerem, da lokalem Fehlersuchen.

8.4 DEPENDENCY-INJECTION

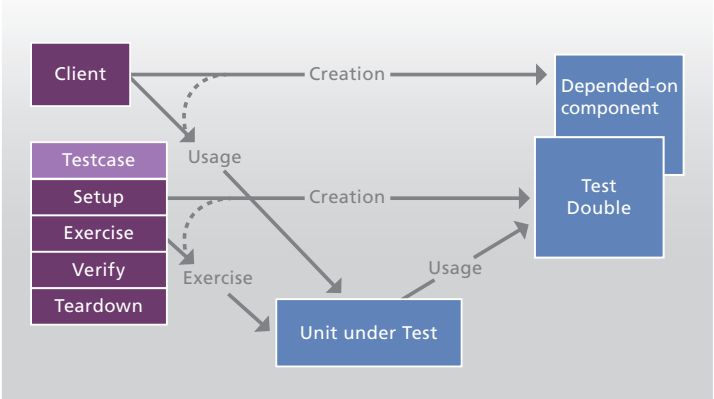


Abbildung 4: Dependency-Injection in Unit-Tests. Quelle: (Meszaros, 2007)

Dependency-Injection (DI) als Anwendung des Inversion-of-Control-Prinzips (Fowler, 2004) beschreibt eine Technik, die eine lose Kopplung zwischen Objekten erlaubt. Loose Coupling bedeutet, dass Objekte möglichst wenige und nur so viele Abhängigkeiten wie nötig haben sollten. Weiter sollte ein Objekt wenn möglich von Interfaces und nicht konkreten Objekten abhängen (McCafferty, 2006).

Beim Dependency-Injection-Pattern nutzt ein Objekt Komponenten, die extern erzeugt wurden. Zum Einspeisen der Objekte wird oft ein DI-Container-Framework benutzt, z. B. Spring (Java, .NET). DI ist aber allgemein anwendbar, um den Code eines Objektes von der konkreten Umsetzung der Klassen, die es benötigt, unabhängig zu machen. In diesem Fall agiert ein Parent-Objekt als Injector und

erzeugt die Objekte. Es gibt grundsätzlich zwei Ansätze, DI zu implementieren: Constructor-Injection und Setter-Injection.

Bei der Constructor-Injection werden die Abhängigkeiten als Parameter im Konstruktor übergeben. Bei der Setter-Injection werden Setter-Methoden benutzt, um die Abhängigkeiten zu setzen.

Dependency-Injection erleichtert Unit-Testing, weil es erlaubt, Test-Doubles anstelle der «echten» Objekte einzuspeisen (Abbildung 4).

8.5 VIER-AUGEN-PRINZIP

Wenn man stark auf eine Aufgabe fokussiert ist, kann es leicht passieren, dass man Offensichtliches übersieht. Kleine Fehler bemerkt man nicht. Es ist deshalb hilfreich, wenn man seinen Code mit einem Kollegen nochmals durchgeht. Bei Code-Reviews werden nicht nur offensichtliche Fehler entdeckt, sondern es wird auch Know-how ausgetauscht und die Kommunikation gefördert. Die Reviewpartner häufig zu wechseln, fördert im Team den Konsens bezüglich Programmierstil und verbreitet das Know-how. Beim Erklären der Überlegungen und durch geschickte Fragen des Reviewpartners werden Schwachstellen und Ungereimtheiten aufgedeckt und Alternativen aufgezeigt. Nicht selten entdeckt der Entwickler seine Fehler, bevor der Reviewer interveniert.

Die extreme Form der Kooperation ist das Pair-Programming, bei dem zwei Entwickler zusammen programmieren (Beck, 1999). Eine abgeschwächte Form ist das Pair-Check-in, bei dem der Code vor jedem Check-in mit jemandem aus dem Team angeschaut wird. Studien haben gezeigt, dass Pair-Programming eine effiziente Art ist, um die Qualität zu erhöhen und somit die Test- und Debug-Kosten zu reduzieren (Cockburn & Williams, 2000).

8.6 KOLLEKTIVE VERANTWORTUNG

Man stellt fest, dass Legacy-Code oft entsteht, weil sich Entwickler nicht verantwortlich fühlen, Code-Smells zu beseitigen. Wenn Komponenten in der Verantwortung von einzelnen Entwicklern liegen (Code-Ownership), ist die Hemmung der Teamkollegen oft gross, Code in «fremden» Komponenten zu ändern. Förderlich für die Softwarequalität ist deshalb eine gemeinsame Codeverantwortung. Entwickler sind motivierter, sauberen Code zu schreiben, weil sie wissen, dass jemand anders ihn auch verstehen muss. Die unterschiedlichen Erfahrungen und Fähigkeiten der Entwickler ergänzen sich und tragen zu qualitativ besserem Code bei.

8.7 CLEAN-CODE-BOY-SCOUT-REGEL

Wenn jeder Entwickler zudem der Clean-Code-Pfadfinderregel «leave the code a little cleaner than you found it» (Martin, 2009) folgt und zur Verbesserung des Codes beiträgt, steigt die Qualität allmählich, und es entsteht ein gemeinsamer Konsens in Bezug auf den Programmierstil und die Vorstellung von sauberem Code.

9 FAZIT

«The greatest mistake you can make in life is to be continually fearing you will make one.» Elbert Hubbard

Um eine Software auch auf lange Sicht am Leben zu erhalten und mit vertretbarem Aufwand weiterzuentwickeln, ist es nötig, den Fokus auf die interne Qualität zu legen.

Damit die Qualität einer Software nicht abnimmt, ist stetiges Refactoring unumgänglich. Automatisierte Unit-Tests sind ein unverzichtbares Mittel, um das Risiko von Fehlern durch Codeänderungen zu minimieren. Wenn Qualität durch gute Unit-Tests abgedeckt ist, diese Tests jederzeit mit geringem Aufwand ausgeführt werden können und in kurzer Zeit Ergebnisse liefern, ist das Fehlerrisiko durch Änderungen stark vermindert worden.

Unit-Tests für Legacy-Code zu schreiben, kann aufwendig sein und lohnt sich nicht in jedem Fall. Muss für die Software noch mehrere Jahre Support geleistet werden und soll die Entwicklung noch einige Zeit weiterlaufen, führt aber kein Weg daran vorbei[13].

Qualitätsgetriebene Entwicklung erfordert von den Entwicklern Disziplin und Professionalität. Sie müssen sich für die Qualität des gesamten Codes gemeinsam verantwortlich fühlen und ihn stetig verbessern. Schwachstellen und Code-Smells sollen ohne Ausreden beseitigt werden. Jedem einzelnen Entwickler muss klar sein, dass in Sachen Qualität keine Kompromisse eingegangen werden.

Versionsverwaltung, Continuous Integration und automatisierte Tests unterstützen und entlasten die Entwickler bei ihrer Arbeit und sind unverzichtbare Hilfsmittel.

Qualitätsgetriebene Entwicklung liefert nicht fehlerfreie Software. Unit-Tests und kurze Feedbackzyklen helfen aber, Fehler schnell und einfach zu lokalisieren. Testbares Design und Clean Code ist einfacher zu verstehen und zu modifizieren.

[13] Ausser vielleicht, wie schon oft passiert, die ganze Software neu zu schreiben.

Projektverantwortliche und Manager müssen verstehen, dass Qualität Aufwand bedeutet. Der Aufwand ist aber abschätzbar. Schlechte Qualität und ungenügende Testabdeckung führen früher oder später zu unkalkulierbarem Mehraufwand. Qualität, und somit Unit-Tests, müssen ausdrücklich gefordert und es muss die nötige Zeit eingeplant werden. Die Qualität darf nicht für andere Projektziele geopfert werden.

10 ANHANG

10.1 AUTOR

Michel Estermann hat einen M. Sc. in Informatik der ETH Zürich und ist seit 2008 bei der bbv Software Services AG als Senior Software Engineer im Embedded Bereich tätig.

10.1 LITERATURVERZEICHNIS

Adzic, G. (2011). Specification by Example: How Successful Teams Deliver the Right Software. Manning Publications.

Atwood, J. (18. September 2006). When Understanding means Rewriting. Von Coding Horror: http://www.codinghorror.com/blog/2006/09/when-understanding-means-rewriting.html abgerufen

Beck, K. (1999). Extreme Programming Explained. Addison-Wesley.
Beck, K. (2002). Test Driven Development: By Example. Addison-Wesley Professional.

Beck, K. (kein Datum). Simple Smalltalk Testing: With Patterns. Von XProgramming.com: http://www.xprogramming.com/testfram.htm abgerufen

Belcham, D. (2009). Working with Brownfield Code. Von devshaped.com: http://devshaped.com/2009/01/working-with-brownfield-code/ abgerufen

Büchi, D. (2012). Continuous Integration für Embedded Systeme. bbv Software Services AG.

Cockburn, A., & Williams, L. (2000). The Costs and Benefits of Pair Programming. Proceedings of the First International Conference on Extreme Programming and Flexible Processes in Software Engineering.

Cohn, M. (2010). Succeeding with Agile. Addison-Wesley.

Dolder, P. (2012). Introduction to Continuous Integration. bbv Software Services AG.

Ellnestam, O. & Brolund, D. (2014). The Mikado Method. Manning Publications

Erdogmus, H., Morisio, M., & Torchiano, M. (January 2005). On the Effectiveness of the Test-First Approach to Programming. IEEE Transactions on Software Engineering, 31(1).

Feathers, M. C. (2005). Working Effectively with Legacy Code. Pearson Education.

Fowler, M. (1999). Refactoring: Improving the Design of Existing Code. Addison-Wesley.

Fowler, M. (23. January 2004). Inversion of Control Containers and the Dependency Injection Pattern. Von Martin Fowler: http://martinfowler.com/articles/injection.html abgerufen

Hunt, A., & Thomas, D. (2007). Pragmatic Unit Testing In C# with NUnit. The Pragmatic Programmers.

Martin, R. C. (11. May 2005). Principles Of OOD. Von butunclebob.com: http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod abgerufen

Martin, R. C. (2009). Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall.

McCafferty, B. (18. April 2006). Dependency Injection for Loose Coupling. Von The Code Project: http://www.codeproject.com/KB/architecture/DependencyInjection.aspx abgerufen

Meszaros, G. (2007). xUnit Test Patterns: Refactoring Test Code. Addison-Wesley.

Müller, M. M., & Padberg, F. (2003). About the Return on Investment of Test-Driven Development. In International Workshop on Economics-Driven Software Engineering Research EDSER-5, (S. 2631).

Müller, U. (2010). Agiles Testen – Qualitätssicherung in agilen Projekten. bbv Software Services AG.

Nagappan, N., Maximilien, E. M., Bhat, T., & Williams, L. (June 2008). Realizing quality improvement through test driven development: results and experiences of four industrial teams. Empirical Softw. Engg., 13(3), 289–302.

Peter, S., & Ehrke, S. (2009). Methods & Tools. Von Refactoring Large Software Systems: http://www.methodsandtools.com/PDF/mt200904.pdf abgerufen

Richardson, J. (4. August 2010). Defect Driven Testing: Your Ticket Out the Door at Five O'Clock. Von Agile Zone: http://agile.dzone.com/articles/defect-driven-testing-your abgerufen

Savoia, A. (9. March 2007). Working Effectively With Characterization Tests. (artima developer) Von Alberto Savoia's Weblog: http://www.artima.com/weblogs/viewpost.jsp?thread=198296 abgerufen

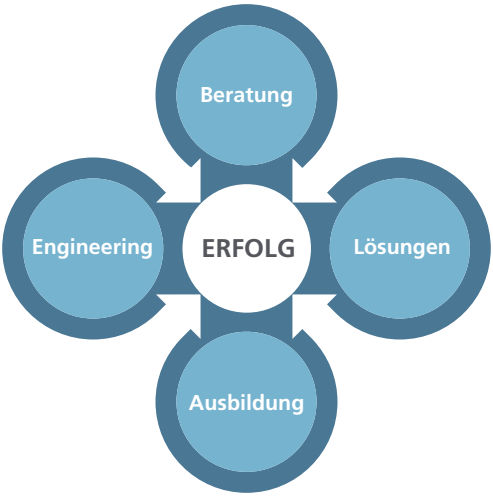
Sierra, K. (7. September 2006). Why „duh"... isn't. Von PASSIONATE Creating passionate users: http:\\headrush.typepad.com/creating_passionate_users/2006/09/why_duh_isnt.html abgerufen

Westphal, F. (2006). Testgetriebene Entwicklung mit JUnit & FIT: Wie Software änderbar bleibt. Dpunkt.

Wilson, J. Q., & George, K. (1982). Broken windows: The police and neighborhood safety.

bbv
MAKING VISIONS WORK.

bbv Software Services AG ist ein Schweizer Software- und Beratungsunternehmen, das Kunden bei der Realisierung ihrer Visionen und Projekte unterstützt. Wir entwickeln individuelle Softwarelösungen und begleiten Kunden mit fundierter Beratung, erstklassigem Software Engineering und langjähriger Branchenerfahrung auf dem Weg zur erfolgreichen Lösung.



Unsere Booklets und vieles mehr finden Sie unter
www.bbv.ch/publikationen

MAKING VISIONS WORK.

www.bbv.ch · info@bbv.ch