Automatisches Testen von Software

Abstract:
In this article an introduction to automated software testing is given in german language. It covers a discussion on the need to test, some basics and then more details on selected aspects such as formal requirement analysis and some technicalities. A method for implementing automated tests into an organization is presented, the Automated Testing Lifecycle Methodology (ATLM). The article is based on the german translation of 'Automated Software Testing: Introduction, Managment, and Performance' by E. Dustin.

Betreuer des Blockseminars:

Prof. Dr. Wüst,
Prof. Dr. Kneisel

Thema:

Automatisches Testen von Software

Ein Überblick über Softwaretests, ihre Anwendung und Implementierung im Rahmen der ATLM.

Erstellt mit dem Textsatzsystem LaTeX

Inhaltsverzeichnis

1 Einleitung **3**

1.1 Themenwahl . . . 3

2 Grundlagen **4**

2.1 Testziele . . . 4

2.1.1 Kostenfragen . . . 5

2.2 Einteilung von Testverfahren . . . 6

2.3 Testgruppen . . . 7

2.3.1 Regressionstests . . . 7

2.3.2 Smoketests . . . 7

2.4 Testverfahren . . . 7

2.4.1 Black-Box-Testverfahren . . . 7

2.4.2 White-Box-Testverfahren . . . 10

3 ATLM - Automated Test Life-cycle Methodology **13**

3.1 Aufbau der ATLM . . . 14

3.2 ATLM und TMM . . . 15

3.3 Die Phasen der ATLM . . . 15

3.3.1 1. Entscheidung für Testautomatisierung . . . 15

3.3.2 2. Erwerb eines Testwerkzeugs 16

3.3.3 3. Einführung des automatisierten Testens 16

3.3.4 4. Planung, Design und Entwicklung der Tests 17

3.3.5 5. Testdurchführung und -verwaltung 17

3.3.6 6. Bewertung des Testprogramms und -prozesses 18

3.4 Ansatz zum Testen von Anforderungen 19

3.4.1 Anforderungsanalyse 19

3.4.2 Ordnen der Anforderungen 22

3.4.3 Schlussfolgerungen 22

4 Fazit **23**

A Abbildungen **24**

Kapitel 1

Einleitung

Das Testen von Software ist für Projekte beinahe jeder Grössenordnung ein wichtiger Faktor für den Erfolg des Systems. Entsprechend existiert heute eine Vielfalt von Techniken und Werkzeugen, um das Testen zu vereinfachen. Vereinfachen heisst: automatisieren, unterstützen, formalisieren oder wenigstens verwalten helfen - Präferenz in dieser Reihenfolge.

Getestet wird, um verschiedenste Ziele zu verifizieren: Anforderungen, Wirtschaftlichkeit, Benutzbarkeit, Fehlerfreiheit oder andere Qualitäten. Diese grosse Bandbreite an Testzielen resultiert in einer entsprechenden Menge an möglichen Testverfahren, die längst nicht alle automatisierbar sind.

Um die genannten Ziele sinnvoll zu vereinen, muss also Testaufwand und Entwicklungsaufwand vorsichtig gegeneinander abgewogen werden, wobei eine Einführung automatischer Tests bedacht werden sollte. Der Entwicklungszyklus soll also bestimmte Tests umfassen, die regelmässig durchgeführt werden. Dieses Ziel ist ohne eine Automatisierung auf technischer Ebene allerdings nicht zu erreichen[1], daher kommt auch der verwendeten Technik eine entscheidende Rolle zu.

1.1 Themenwahl

Es ist ein Testteam vonnöten, um den eben angesprochenen Grad der Testautomatisierung zu erreichen. Die dadurch entstehenden organisatorischen Fragen werden in [1], der Hauptquelle dieses Seminars, sehr breit behandelt. Aufgrund des eher technischen Rahmens des Seminars werden diese Themen (verglichen mit dem Buch) nur marginal behandelt.

In dem Buch werden diese Themen ausschliesslich in den Kontext der ATLM (siehe 3, Seite 13) gestellt, wobei der Überblick über die Testverfahren etwas verloren geht. Hier soll ein breitgefächerter Überblick entstehen, der an ausgesuchten Stellen vertieft wird.

[1] Selbst mit einer Wiedereinführung der Sklaverei oder dem was sich mancher Politiker unter Computer-Indern vorstellt, könnte man immer noch nicht die *korrekte* Wiederholung der definierten Tests gewährleisten

Kapitel 2

Grundlagen

2.1 Testziele

Das primäre Ziel beim Testen ist die Erhöhung bzw. Sicherstellung der Produktqualität, also die fehlerfreie Ausführung unter allen Bedingungen. Da Software eine stark und vor allem uneinheitlich vernetzte Interna hat, kann nur ein konsequentes Testen die Funktionalität von Teilen oder dem Gesamtsystem gewährleisten.

Um ein sinnvolles Mass an Tests zu erreichen, müssen die Ziele, die man mit den Tests gewährleisten möchte, klar sein. Die Ziele lassen sich teilweise aus den Anforderungen ableiten, da gerade beim Ersetzen eines alten Systems dessen Mängel zu Anforderungen gemacht werden. Diese Ziele können folgende Problembereiche umfassen:

- Stabilität

  kein bekannter Test darf eine unvorhergesehene Fehlermeldung oder gar einen Absturz provozieren. Belastungstests sollten vorgesehen sein. Die Tests müssen jeden Teil der Anwendung abdecken.

- Korrektheit

  ... nennt man das Einhalten der Anforderungen. Anforderungen müssen mit besonderer Sorgfalt getestet werden, um ihre Einhaltung sicherzustellen. Unter anderem die folgenden Punkte müssen zutreffen, damit die Software korrekt ist:

  - Gültige Eingaben liefern die erwartete Ausgabe
  - Ungültige Eingaben werden angemessen Zurückgewiesen
  - Das Programm läuft solange stabil, wie erwartet wird.
  - Das Programm läuft gemäss der Spezifikation

- Softwarequalität/Wartbarkeit

...kann durch konsequentes verfolgen der Änderungen bewertet werden: Pendeln sich die Änderungen gleichmässig in den verschiedenen Modulen ein oder gibt es 'Problemkinder'? Gibt es noch regelmässig Änderungen an Komponenten oder dem Entwurf? Weisen Softwaremetriken auf problematische Stellen hin, die einfacher sein könnten?

- Antwortverhalten

  Oft wird eine gewisse maximale oder mittlere Antwortzeit gefordert oder ergibt sich aus dem Kontext. Zwar kann auch heute noch eine Software nie schnell genug arbeiten, aber aus Rücksicht auf das Budget gibt man sich mit brauchbarem Ergebnis zufrieden.

- Benutzerfreundlichkeit (Usability)

  Die Software muss den Anforderungen der Endbenutzer genügen und idealerweise leicht verstanden werden.

- Laufzeitverhalten

  Braucht die Anwendung zu lange oder belegt zuviel Arbeitsspeicher bei bestimmten Vorgängen? Existieren Speicherlecks oder sind temporäre Dateien übrig nach Beendigung?

Eine wichtige Eigenschaft der Testziele ist die Formulierung von definitiven Kriterien zur Erfüllung des Zieles, also einer Möglichkeit, zwischen akzeptablen und ungenügenden Lösungen zu unterscheiden. Auf die Existenz bzw. unmissverständliche Formulierung derselben ist daher grosser Wert zu legen, um den Ausgang eines Tests vergleichbar bewerten zu können.

2.1.1 Kostenfragen

Ebenfalls eine Rolle spielt der Kostenfaktor: Betriebswirtschaftlich lohnt sich Testen nur dann, wenn die eingesetzen Mittel unter den Eingesparten liegen. Da das Einsparpotenzial oft nur sehr gering geschätzt wird, wird der Testaufwand entsprechend angepasst. Das ist oft ein Fehler: Einfache Usability-Tests zum Beispiel, die eine intuitive und bedarfsgerechte Benutzerführung sicherstellen, können den Aufwand für Schulung, Support und Helpdesk senken, Arbeitszeit sparen und die Verkaufbarkeit erhöhen bzw. die Verlängerung von Lizenzen wahrscheinlicher machen. Von diesen Vorzügen kommt je nach Vertragsgestaltung ein beträchtlicher Teil beim Softwarehaus an, sollte nicht ohnehin für den internen Bedarf produziert werden. Folgende Tabelle gibt eine Kostenrelation für die Beseitigung von Fehlern in den jeweiligen Projektphasen an (basierend auf Erfahrungen):

Definition	1
Design	2
detailiertes Design	5
Programmierung	10
Einheitentest	15
Integrationstest	22
Systemtest	50
nach Auslieferung	100+

Gleichzeitig ein Testziel und eine gerne unterschätzte Folge eines erfolgreichen Testprozesses kann die Möglichkeit sein, komplexere Projekte zu realisieren - ein kaum zu überschätzender Wettbewerbsfaktor.

2.2 Einteilung von Testverfahren

Testverfahren können aufgrund ihrer Eigenschaften in verschiedene Kategorien eingeteilt werden, die hier kurz vorgestellt werden sollen.

- Softwaretests

  Diese Tests basieren auf dem Quellcode oder einer laufenden Software (Testsystem) und können daher nur laufen, wenn die Programmierung schon angefangen hat:

  - ▷ Black-Box-Tests

    ... sind Tests, die auf der laufenden Software operieren, z.B. Usability oder Lasttests. In Umgebungen, in denen Testen keine grosse Rolle spielt, sind oft alle Tests Blackbox-Tests.
  - ▷ White-Box-Tests

    ... setzen interne Kenntnisse der Software voraus, in der Regel also Quellcodezugriff. Beispiele sind Code Coverage, Unit-Tests, Lint oder der sehr selten praktikable formale Beweis der Korrektheit.

  Diese Tests sind in aller Regel automatisierbar oder formalisierbar, woraus man aber nicht den Trugschluss ziehen sollte, die Durchführung bräuchte wenig Kenntnisse der Materie.

  Die Einteilung ist mittlerweile etwas problematisch, weil etwa in Java oder .NET erstellte Software in aller Regel auch im Auslieferungszustand genug Metainformationen enthält, um etwa ein Klassen- oder Aufrufdiagramm zu erstellen, wofür in klassisch kompilierenden Umgebungen Quellcodezugriff erforderlich ist. Verwandt ist die Forderung, dass Programmierer nicht auch Tester sein sollten: Dann kommt nämlich implizit beim Test Wissen über die Programmstruktur zum Einsatz und verfälscht das Testergebnis.

- fehlervermeidende Tests

  Diese sollen die Korrektheit von Anforderungen oder Schnittstellendefinitionen belegen und so späte und damit teure Änderungen verhindern, die man auch hätte vermeiden können. Diese Änderungen sollen nur stattfinden, wenn wirklich neue Anforderungen kommen, die aus dem Kontext[1] bisher ausgeschlossen waren. Dies bezieht auch das Überprüfen des Kontextes selbst mit ein, also ob dieser für ein erfolgreiches System ausreichend, zu eng oder zu weit gefasst ist.

  Wenn Testen die Entwicklungskosten senken oder im Zaum halten soll, dann ist diese Testkategorie sicher die wichtigste. Es existieren zwar Automatisierungansätze, doch

[1] Der Kontext ist alles, was korrekte Anforderungen definiert. Anders ausgedrückt ist die Beziehung von Kontext und Anforderungen analog der von Quellcode und Kompilat, nur das Menschen die Überführung erledigen. Darin liegt üblicherweise auch das Hauptproblem.

prinzipbedingt kann Software hier nur unterstützend eingreifen. Ein erfahrenerer Softwareingenieur ist trotz allen Hilfestellungen unabdingbar für diese Aufgabe.

2.3 Testgruppen

Wenn eine Reihe Tests existiert, kommt man irgendwann an den Punkt, wo das durchführen aller Tests für einen neuen Build nicht praktikabel ist. Daher führt man Testgruppen ein, die nach bestimmten Kriterien ausgesucht werden:

2.3.1 Regressionstests

Regressionstests sollen die Erhaltung von bekannt funktionalen Vorgängen sicherstellen, d.h. dass keine Änderung eine funktionale Beeinträchtigung zur Folge hat. Eine beliebte Möglichkeit ist es z.B. passend zu beseitigten Fehlern Regressionstests zu erstellen, die genau diese Funktionalität prüfen.

Regressionstests laufen typischerweise nach zentralen Änderungen, und vor jedem Release. Der Grund liegt in der hohen Wahrscheinlichkeit, mit der Änderungen sich an Stellen auswirken, die nicht bedacht und nicht beabsichtigt waren.

2.3.2 Smoketests

Smoketests sind immer relativ wenige Tests, die die zentralen Funktionen abklopfen. Sie sollen sinnloses Testen an ohnehin defekten Versionen verhindern, indem sie ein breites Spektrum kleiner, schneller Akzeptanztests definieren, ohne dessen Erfolg keine weiteren Tests an einem Build gemacht werden.

Smoketests eignen sich auch zum Anfang der Testerstellung, wenn noch wenig stabil läuft, da Smoketests nur die Grundfunktionalität abdecken.

2.4 Testverfahren

Es gibt eine grosse Anzahl von Testverfahren, die hier mit einer kurzen Beschreibung vorgestellt werden sollen. Zuerst die Black-Box-Tests:

2.4.1 Black-Box-Testverfahren

- Lasttest

Beim Lasttest wird versucht, das System durch Überladung mit Anfragen in eine Situation zu bringen, in der es zusammenbricht. Idealerweise bedeutet Zusammenbruch nicht Ausstieg mit Fehlern, sondern Zusammenbruch der Bearbeitung von Aufgaben, also ein sinnvoller Umgang im der Überlastung. Ein Beispiel währe das Ablehnen von Aufträgen wegen Überlastung. Weniger Sinnvoll hingegen der klassische Absturz mit inkonsistent zurückgelassenen Daten.

Es gibt verschiedene Arten des Belastungtests, man kann auch einzelne Einheiten oder Komponenten einem Belastungstest unterziehen. So kann man sich näher an kritische Punkte herantasten. Auch der korrekte Umgang mit parallelen Zugriffen muss getestet werden, so kann Beispielsweise ein falsches Isolationsniveau beim Datenbankzugriff erkannt werden.

Die meisten Programme sind in irgendeiner Form überlastbar, und es ist lediglich Ziel diese Grenze zu kennen.

- Performance-Test

  Der Performance-Test ähnelt dem Lasttest, es ist aber nicht die *Über*- sondern die *Be*lastung Ziel des Tests. Man sucht also die Antwort auf die Frage, bei welcher Belastung das System noch ausreichend schnell ist. In Client-Server-Architekturen kann die Performance leicht von kaufentscheidender Bedeutung sein und damit wesentlich den Erfolg des Systems bestimmen.

  Zur Automatisierung von Last-und Performancetests gibt es eine menge Werkzeuge, wie Testdatengeneratoren, GUI-Replay-Programme, Benutzersimulatoren (für Datenbanken oder GUIs).

- GUI-Test Tests von Graphischen Benutzeroberflächen werden meist mit sog. *Capture-Replay-Programmen* automatisiert. Mit ihrer Hilfe kann die Funktion von GUIs getestet werden, indem Benutzeraktionen wie z.B. Mausklicks oder Tastatureingaben aufgezeichnet werden und automatisch wiederholt werden können.

  Viele Programme setzen die Eingaben dazu in Skripten um, die später manuell nachgebessert werden können. Ebenfalls wichtig ist die Umsetzung der Mausklicks in Fensterkoordinaten oder besser in Steuerelemente bzw. deren Nummern, soweit möglich[2].

  So entstandene Skripten müssen natürlich auch den Erfolg des Tests beurteilen können. Dazu kann man z.B. einen Bildvergleich machen, was aber wenig Praktikabel ist, da man bei kleinen Layout- oder Textänderungen schon falsche negative Tests bekommt und nachbessern muss[3]. Besser an dieser Stelle ist eine vom GUI unabhängige Erfolgskontrolle z.B. über eine Datenbankabfrage, die die Tätigkeit des GUI verifiziert, oder das Prüfen von Dateien.

  Zu den GUI-Testverfahren gehören auch *stochastische Tests* (auch Affentest oder 'Test Monkey'). Sie Testen Zufallsbasiert, entweder ohne oder mit minimalen Kenntnissen über das Programm, welche in Form einer Zustandstabelle vorliegen. Diese Verfahren

[2] moderne Layoutsysteme machen die Steuerelementnummer als 'hartverdrahtetes' Identifikationsmerkmal überflüssig, so das sie eher zwischen Builds verändert werden

[3] Auch beliebt ist hier die vorherige Blamage vor der Programmiermannschaft. Spätestens dann gibt es aber *wirklich* Grund zum Nachbessern.

sind recht umstritten, sie sollten auf keinen Fall der einzige GUI-Test sein. In manchen Projekten wurden allerdings etwa 10-20 Prozent der Fehler auf diese Weise aufgedeckt[4]

Die Funktionsweise ist sehr Simpel: Ein Test Monkey generiert Klicks und Eingaben an das Programm nach dem Zufallsprinzip, mit minimalen Vorgaben durch den Benutzer. Die Erfolgskontrolle besteht im Einhalten einer vorgegebenen Zustandstabelle bzw. deren Übergängen oder schlicht im Nichtabsturz des Programms.

- Usability-Test

  Unter Usability-Tests versteht man Tests, in denen die die Bedienbarkeit des Programms durch den Benutzer bewertet wird. Also solches immun gegen jede Art der Automatisierung, machen die Autoren in [1] den interessanten[5] Vorschlag, die Tests mit einem Prototypen und einem Capture-Replay-Programm zu machen, um jederzeit nachvollziehen zu können, wie ein Benutzer das Programm bedienen würde. Da sie praktisch nicht automatisierbar sind, wird hier nicht weiter darauf eingegangen.

- Sicherheitstest

  Sicherheitstests versuchen die Sicherheitsbestimmungen zu unterlaufen. Da man diese dazu kennen muss, ist es je nach Geschmack kein Black-Box-Verfahren. Ein Skript könnte zum Beispiel prüfen, ob temporäre Dateien world-readable oder anderweitig unsicher sind. Die Implementierung von Kryptographie-Modulen muss getestet werden.

- Integrationstest

  Der Integrationstest ist ein Test der Anwendung als Ganzes, oft im Rahmen einer Installation. Dabei wird ein grosses Spektrum an Tests durchgeführt, idealerweise alle verfügbaren Black-Box-Tests. Typischerweise kommen folgende Testarten zum Einsatz:

  - Funktionstest
  - Regressionstests
  - Sicherheitstests
  - Belastungstests
  - Leistungstests
  - Usability
  - Datenintegritätstest
  - Backup und Restore - Test

  Integrationstests sind aufwendig, aber vergleichsweise selten nötig. Ein Akzeptanztest ist oft lediglich ein Integrationstest beim Kunden.

[4]Siehe hier

[5]Interessant findet der Autor vor allem die Konstruktion einer Möglichkeit, Usability-Tests zu einer Liste von *automatischen* Testverfahren hinzuzufügen. Aufgrund der komplett fehlenden Information über das Verhalten des Benutzers selber ist hier wohl mehr eine Videokamera gefragt und der skizzierte Fall ein selten Praktikabler

2.4.2 White-Box-Testverfahren

- Unit-Test

  Unit-Tests zielen auf das automatische Testen von *Units* (Einheiten) des Codes. Ihr Gedanke basiert auf der Tatsache, das ein zuverlässiges System auch aus zuverlässigen und hochwertigen Einzelteilen bestehen muss. Die Einzelteile sollen also besser (getestet) werden.

  Eine Einheit kann zum Beispiel eine Klasse, eine Komponente oder eine API sein. Diese wird in ein Testbett integriert, so dass sie eventuell unabhängig von anderen Komponenten getestet werden kann.

  Unit-Tests sind mit vielen (Open Source-) Werkzeugen automatisierbar, die für praktisch alle Plattformen verfügbar sind. Als Beispiel hier *NUnit* (Abb. 2.1):

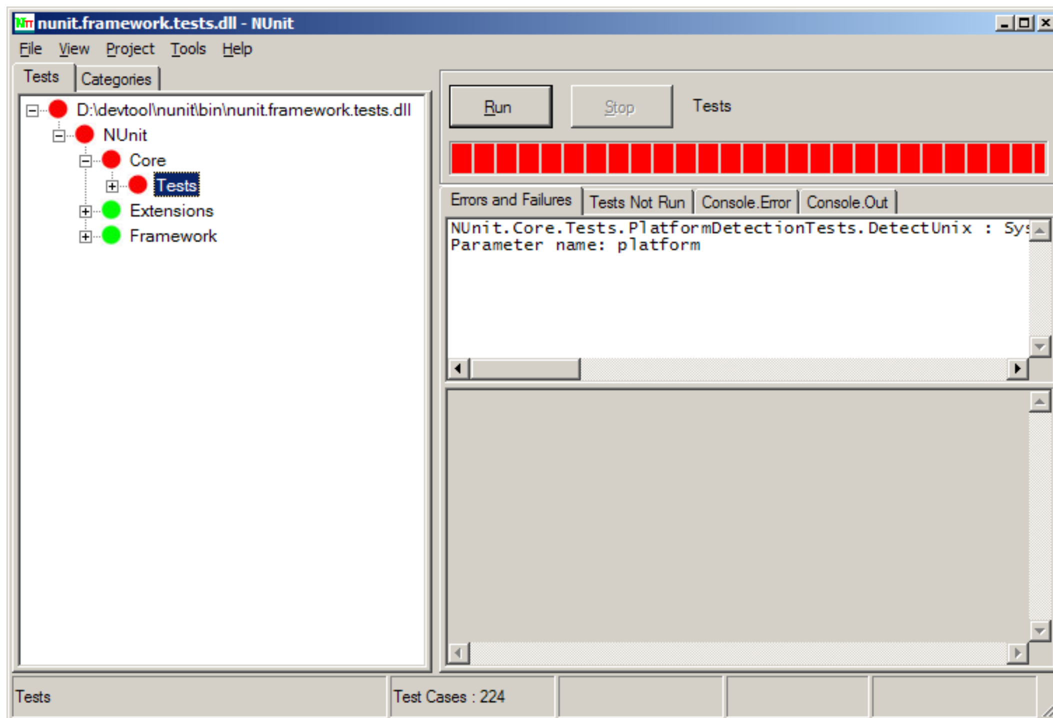

Abbildung 2.1: NUnit 2.2

  Die sehr übersichtliche Gruppierung der Tests erlaubt eine automatisierte Durchführung und GUI-gestützte Auswertung. Die auffallende Markierung mit Rot macht auf fehlgeschlagene Tests aufmerksam, Gelb auf nicht ausgeführte und Grün auf erfolgreiche Tests. Fehlschläge werden in der Hierarchie (die Assemblys, also DLLs, und deren Namensräume umfasst) nach oben propagiert, damit sie auf jeden Fall bemerkt werden.

- Code-Metriken

Code-Metriken gibt es viele, z.B. LOC[6] oder Funktionslänge, aber auch solche die den Namen Metrik eher verdienen, z.B. die Zyklomatische Komplexität (CC) oder die Halstead-Metriken. Die letzteren beiden sind typischerweise in komplexen Quellcodes höher, spiegeln also wirkliche Problemstellen wieder bzw. können zu ihrer Identifizierung beitragen. Diese Techniken werden auch *statische Analyse* genannt, im Gegensatz zu weiter unten beschriebenen Techniken der *dynamischen Analyse*, Code Coverage, Boundchecker und Profiler.

Eine Komponente mit hoher CC (>20) sollte ausführlich getestet werden, da sie mit sehr vielen verschiedenen Komponenten zusammenarbeitet und wahrscheinlich komplex und fehleranfällig ist.

- Lint

  Lint ist mit dem aufkommen von C entwickelt worden und untersucht den Kode auf schlechten Stil. Viele der von Lint entdeckten 'Problemstellen' sind sehr C-spezifisch, z.B. die Zuweisung in einem Vergleich. Lint gibt also Hinweise auf mögliche Fehler aus, die mit Vorsicht zu geniessen sind, da ein theoretisches perfekt fehlerfreies Programm immer noch für Lint wie ein Software-GAU aussehen könnte. Wenngleich es unwahrscheinlich ist.

- Code Coverage

  Wenn Code mit Debugging-Informationen instrumentiert ist, kann unter anderem ermittelt werden, welche Instruktionen ausgeführt wurden. In Kombination mit Testaufrufen kann also ermittelt werden, welcher Code getestet wird und welcher nicht. Das Verhältnis getesteter Code zu Gesamtcode wird als Testabdeckung oder Code Coverage bezeichnet.

  Wer misst, misst Mist: Problematisch sind hier Compilerfehler, vor allem die vergleichsweise häufigen Optimiererfehler. Da optimierter Code oft nicht mehr eindeutig den erzeugenden Ausdrücken zugeordnet werden kann[7], sind die Debugging-Informationen für optimierten Code im praktischen Debugging oft kaum zu gebrauchen. Wenn der Compiler überhaupt Debugging-Informationen für optimierten Code generiert.

  Die praktischen Implikationen: Debugging erfolgt ohne Optimierungen[8]. Speicherlöcher werden schwerer zu fassen wegen dem veränderten Speicherlayout im Vergleich zum Release beziehungsweise dem optimierten Build.

  Für Code Coverage Tools resultiert diese Situation einer Einschränkung ihrer Aussage auf den Debug-Build, was fehlerhafte Optimierungen aussen vor lässt. Zusätzlich kann eine im Debug-Modus vorhandene manuelle Instrumentierung, z.B. über *Assertions*, die Ergebnisse ungewollt beeinflussen.

- Boundchecker

  Wie Code Coverage Tools sind Boundchecker *Laufzeiterweiterungen*, die das korrekte Ansprechen des vom Programm reservierten Speichers sicherstellen. Der Boundchecker

[6] Lines Of Code, Zeilenzahl

[7] Compiler für Pipeline-Architekturen verwenden in der Regel ein sogenanntes *instruction reordering* zur Vermeidung von zeitintensiven *pipeline stalls*

[8] Bei vielen Compilern aus genannten Gründen auch die Vorgabe, wenn Debugging-Informationen gewünscht sind.

prüft also beim konkreten Programmablauf, ob Speicher über ungültige Indizes beim Array-Zugriff angesprochen wird. Normalerweise geschieht das parallel mit einer Suche nach Speicherlöchern, also nicht freigegebenem Speicher zu dem keine Referenzen mehr existieren, und hängenden Zeigern, also Zeiger in bereits freigegebene Speicherbereiche.

  In Umgebungen mit Garbage Collector, also automatischer Speicherverwaltung, muss die Definition von Speicherlöchern ausgeweitet werden auf unnötige Referenzen zu Speicher, der deshalb dann nicht freigegeben werden kann. Hängende Zeiger kann es hier nicht geben[9], und 'Unnötig' ist leider keine automatisiert prüfbare Eigenschaft, weshalb es in Umgebungen wie Java oder .NET solche Tools überflüssig sind.

- (Memory) Profiler

  Profiler existieren seit es Hochsprachen gibt. Sie ermitteln das Laufzeitverhalten eines Programms und machen so nicht oder stark verwendete Programmteile sichtbar. Sie sind eigentlich kein Testverfahren, wenn man von dem Spezialfall absieht, das ein gewisser Speicherverbrauch eingehalten werden soll.

  Manche Tools können auch das verhalten bei begrenztem Speicher simulieren, indem sie Speicherbelegungsversuche des Programms gezielt blockieren. So kann man Testen, wie sich Programme in spärlicher ausgestatteten Umgebungen verhalten.

  Wichtig kann auch der Langzeitbetrieb sein: Steigt der Speicherverbrauch langfristig, ist das ein Indiz für ein Speicherloch. Manche Tools können auch die Speicheranforderungsanweisung identifizieren, die ein Gegenstück vermissen lässt.

- Einfügen von Fehlern

  ... testet das Verhalten der Software in Extremfällen oder schlicht bei der Fehlerbehandlung. Über bedingte Kompilation können die Test-Fehler eingebaut und dann geprüft werden. Mann kommt so auch mangelnden Parameterprüfungen auf die schliche (*Fail-Test*) oder kann die Funktionalität der Test-Suite prüfen - schliesslich sollte man sich auch darauf nicht blind verlassen.

[9] Im wesentlichen darauf begründete sich die am Anfang des Java-Hypes gerne geäusserte Hoffnung, Java mache Programmfehlern ein Ende - was naturgemäss in der Fachpresse nie wirklich Thema war.

Kapitel 3

ATLM - Automated Test Life-cycle Methodology

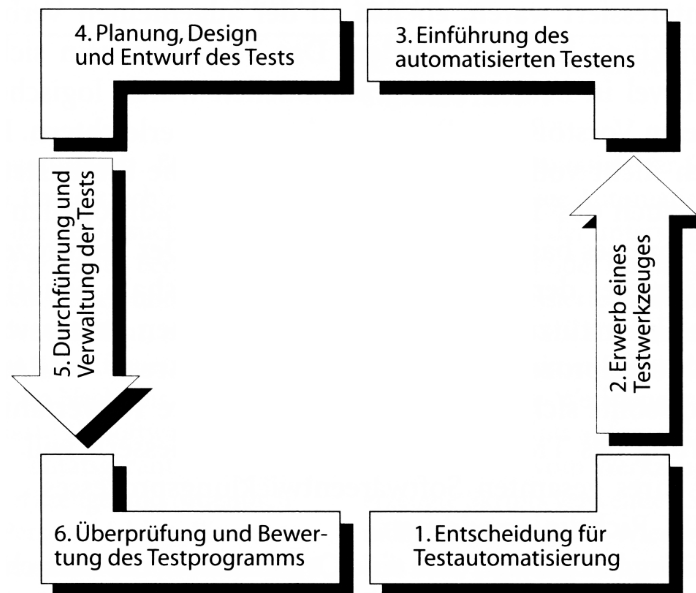

Abbildung 3.1: Die Phasen der ATLM

Die ATLM ist 'eine strukturierte Methode, die auf die erfolgreiche Implementierung automatisierter Tests abzielt' [1]. Sie ist als solches zur Ergänzung des Entwicklungprozesses gedacht, ist also nicht stark an ein spezifisches Softwareentwicklungsmodell gekoppelt. Sie ist natürlich nicht gänzlich unabhängig: Die ATLM zielt auf moderne Ansätze, die den Endbenutzer früh aktiv in die Entwicklung einbeziehen. Genauer gesagt in der Analysephase, dem Entwurf, und dem Test.

Die ATLM zielt auf eine Implementierung in der *Organisation*, nicht unbedingt in einem spezifischen *Projekt*. Wobei natürlich ein Projekt das erste sein muss, und es wird daher eindringlich gewarnt, ein Projekt zu wählen dass von der Einführung auch profitiert. Andernfalls wird es hinterher umso schwerer die praktisch unbewiesenen, bei einem Scheitern 'praktisch widerlegten' Vorzüge zu vermitteln. Auch zu hohe Erwartungen bei den Beteiligten können für eine spätere Akzeptanz hinderlich sein.

Das Testen soll also ein eigener Bereich oder wenigstens ein eigener Zyklus werden, in dem die Tests weiterentwickelt werden. Dabei beschreibt die Methode z.B. Möglichkeiten, die Geschäftsführung von den Vorteilen eines Testprozesses und der einhergehenden Einführung des automatisierten Testens zu überzeugen, oder die richtige Wahl von Testwerkzeugen, wobei die diskutierten Details den technischen Rahmen weit hinter sich lassen.

3.1 Aufbau der ATLM

Die ATLM ist 6-Phasen-Modell, von der Entscheidung zum Testen bis zur abschliessenden Bewertung und Verbesserung des Testprozesses. Die Phasen sind an Entwicklungschritte angelehnt, die sich in eigentlich allen Entwicklungsmodellen identifizieren lassen.

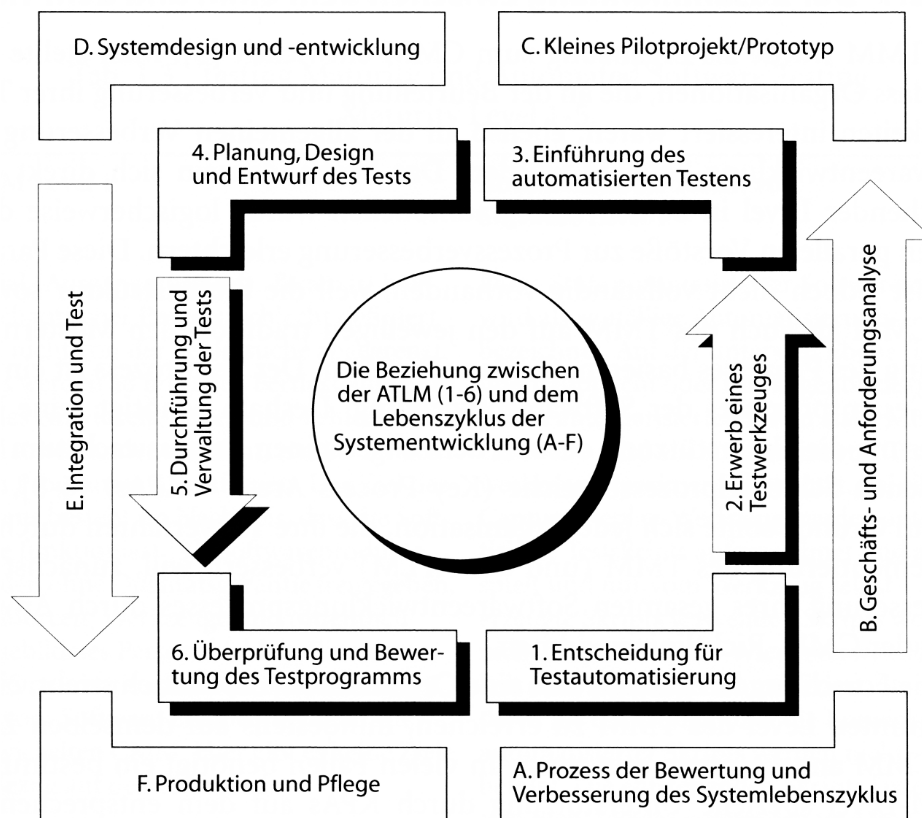

Abbildung 3.2: ATLM und Entwicklungsmodell

3.2 ATLM und TMM

Das Testing Maturity Model (TMM) spiegelt in den vom Capability Maturity Model[1] (CMM) bekannten Entwicklungsstufen[2] die Ausgereiftheit eines Testprozesses. Da die ATLM einen Einführungs- oder Verbesserungsprozess beschreibt, gibt es keine Beziehung zwischen den ATLM-und TMM-Stufen.

Vielmehr ist die ATLM ein Weg zu höheren TMM-Reifegraden. Der entsprechende CMM-Reifegrad sollte zu diesem Zeitpunkt schon beherrscht sein. Die ATLM nimmt explizit hierauf Bezug und schlägt Strategien zu jedem Übergang in eine der höheren Stufen vor.

Ausserdem wird ein *Automated Software Testing Maturity Level* definiert, der die TMM-Reifegrade aus der Perspektive des automatisierten Testens spiegelt.

3.3 Die Phasen der ATLM

Hier soll kurz jede Phase erläutert werden, mit ihren Vorbedingungen, Aktivitäten und Stolperfallen.

3.3.1 1. Entscheidung für Testautomatisierung

Es wurde festgestellt, das der bisherige (manuelle?) Testansatz nicht genügt. Aufgrund von Recherchen wird die Einführung automatisierter Tests beschlossen und nach einem Pilotprojekt ausschau gehalten.

Der mit der Einführung Beauftragte muss ein geeignetes Projekt identifizieren, dessen Testanforderungen kennenlernen und falsche Erwartungen ausräumen. Dazu gehören z.B. Ängste der Testingenieure, sich selbst überflüssig zu machen, oder entsprechende Erwartungen auf der Seite der Geschäftsleitung. Schliesslich ist Testen ja teuer. Automatisiertes Testen ist auch kein Patentrezept gegen Bugs oder Qualitätsprobleme im Allgemeinen. Gerade die in der Regel technisch ungebildete Geschäftsleitung, z.B. die den Kauf von Werkzeugen autorisieren muss, kann falsche Erwartungen haben. Es ist wichtig, diese als Fortschritt gegenüber manuellen Tests zu vermitteln, die das Testen vereinfachen kann und sich nur langfristig auszahlen kann.

Aufgrund der mit dem automatisierten Testen einhergehenden Anforderungen an die Tester sollte in dieser Phase wenn möglich schon die Kommunikation mit der Programmierung verbessert werden, z.B. durch teils geteilte Räumlichkeiten.

[1] Das CMM ist mittlerweile durch das CMMI (I wie Integrated) ersetzt worden und soll auslaufen

[2] initial(1), repeatable(2), defined(3), managed(4), optimized(5)

3.3.2 2. Erwerb eines Testwerkzeugs

Nachdem die Geschäftsführung grünes Licht gegeben hat, kann der Testingenieur die benötigten Werkzeuge aussuchen. Die wichtigsten Kriterien sind die Entwicklungs- bzw. Testumgebung und die Art der durchzuführenden Tests. Danach richtet sich im wesentlichen die Auswahl an verfügbaren Werkzeugen. Für bestimmte Tests ist das aber wiederum relativ unwichtig, denn Datenbankabfragen können z.B. von beinahe jedem Betriebssystem aus durchgeführt und ihre Performance gemessen werden.

Wenn man sich eine Marktübersicht verschafft hat, muss man sie in der Regel weiter einengen mittels spezifischeren Kriterien, die vom Projekt oder dem Entwicklungsystem abhängen. Ist das Werkzeug mit dem Compiler/der Datenbank kompatibel, unterstützt es Dritthersteller-Widgets, wird es vielleicht von im Projekt verwendeten Konstrukten durcheinander gebracht? Solche Fragen lassen sich oft nur durch Testen beantworten, weshalb man mit Demoversionen der in Frage kommenden Tools experimentieren sollte, bevor man etwas einsetzt.

Natürlich spielt auch der Preis eine Rolle, schliesslich muss die Geschäftsführung einen Kauf absegnen. Die Vorteile eines teureren Werkzeugs gegenüber der günstigeren Alternative müssen also entsprechend kommuniziert werden.

3.3.3 3. Einführung des automatisierten Testens

Wenn passende Werkzeuge vorhanden sind, kann ihre Einführung begonnen werden. Hier ist Vorsicht geboten, denn möglicherweise noch vorhandene Vorurteile können zu Akzeptanzproblemen führen, die eine erfolgreiche Einführung verhindern.

Wichtig ist der richtige Zeitpunkt, damit die Aktivität nicht als unpassende, schlecht koordinierte Massnahme aufgefasst wird. Zu früh ist ebenso problematisch wie zu spät. Zu früh kann z.B. die Abhänigkeit der Testskripts von ungefestigtem Code fördern, eine Tatsache die den Entwicklern natürlich klar ist und damit die Glaubwürdigkeit der Massnahme untergräbt. Zu späte Einführung kann zu Verspätungen wegen des Testens führen, von den Auswirkungen auf die Moral muss nicht noch geredet werden.

Wichtig ist auch eine Strategie, um die Integration reibungslos verlaufen zu lassen. Schliesslich sollen die Werkzeuge in den Prozess eingegliedert werden, nicht umgekehrt. Die ATLM gibt hier einen recht konkreten Leitfaden, er umfasst unter anderem:

- Testprozessanalyse: Ziele, Strategien, Masstäbe
- Schulung der Mitarbeiter
- Prüfung von Werkzeug und dessen Kompatibilität
- Rollen und Verantwortlichkeiten

3.3.4 4. Planung, Design und Entwicklung der Tests

Ein gutes Testprogramm muss auch geplant werden. Das Beginnt mit der Prüfung der erforderlichen Aktivitäten, um Sicherzustellen, das alle Ressourcen, Methoden und Prozesse Vorhanden und Einsatzbereit sind. Für jedes Testprogramm müssen Ziele und Anforderungen definiert werden. Teststrategien müssen Entwickelt werden, die die Anforderungen unterstützen.

Meilensteine und ein Zeitplan sollten entwickelt werden. In dieser Phase ist auch die beste Gelegenheit, den Testprozess zu dokumentieren. Das Umfasst die Planungsaktivitäten, die Testanforderungen, die Tests selber, die Testziele und die Integration mit Programm, Testumgebung und Werkzeug. Die ATLM empfiehlt, eine Vorlage so anzupassen, das sie den Testansatz vollständig dokumentiert.

Beim Design der Tests kann man sich an die besprochenen Verfahren halten. Nur wenige Tests werden ein völlig anderes Grundmuster aufweisen.

Beim Entwurf der Tests muss, wie bei jeder Software, auf Wiederverwendbarkeit, Skalierbarkeit, und Qualität geachtet werden. So können die Tests nach Änderungen in der Software schneller an die neuen Bedingungen angepasst werden. Auch für eine eventuelle Nachfolgeversion ist es wünschenswert, über *wartbare* Tests zu verfügen. Eventuell vorhandene Programmierrichtlinien für die verwendeten Skriptsprachen sollten ausgesucht und eingehalten werden.

3.3.5 5. Testdurchführung und -verwaltung

Die Testumgebung wurde nach dem Testplan eingerichtet, die entwickelten Testreihen können nun verwendet werden, um das Programm zu prüfen. Die Pläne für die Tests werden in der vom Testplan vorgesehenen Reihenfolge abgearbeitet. Eventuell wird ein Nutzerakzeptanztest durchgeführt.

Nach der Durchführung müssen die Testresultate ausgewertet und dokumentiert werden. Beispielsweise wir es oft zu der Situation kommen, das Testergebnisse nicht den erwarteten Ergebnissen entsprechen. Diese sind aber nicht immer auch Problemindikatoren, sondern können mit dem Testverfahren bzw. seiner Implementierung zusammenhängen. Wenn es sich also um einen Fehler im Test oder einen anderen Nicht-Softwarefehler handelt, muss das in den Testergebnissen dokumentiert sein und eventuell behoben werden. Andernfalls muss natürlich ein Bugreport erstellt werden.

Um jederzeit in der Lage zu sein, einen Fortschrittsbericht zu geben, sollte der Status der Testdurchführung erfasst werden. Dazu muss jeder Testlauf mit Ergebnissen dokumentiert werden, aufgeschlüsselt nach den eingesetzten Tests.

Um den Testfortschritt bewerten zu können, müssen ferner verschiedene Indikatoren gesammelt werden, Beispielsweise das Verhältnis von durchgeführten Tests zu gefundenen Fehlern.

Die spezifischen Indikatoren müssen auf das Projekt abgestimmt sein und seine Qualität sicherstellen können. Beispielsweise macht es in einem sehr grossen Projekt durchaus Sinn, geringe (bekannte) Fehler in Nebenfunktionen zuzulassen.

Manche Indikatoren können auch die Qualität des Testprozesses beurteilen, den Fortschritt der Anwendung oder ihre Qualität. Diese sind für die Klärung der Frage wichtig, wann das Testen beendet ist. Eine vollständig fehlerfreier Testdurchlauf ist leider oft kein realisierbares Ziel - ausserdem bedeutet das ja noch keine fehlerfreie Anwendung.

Als Indikatoren bieten sich an:

- Testabdeckung (Testverfahren / Testanforderungen, sollte > 1 sein)
- Ausführungsstatus (Ausgeführte- / Gesamttests)
- Fehlererkennungsrate (erkannte Fehler / durchgeführte Tests)
- Fehlerlaufzeit (vom Entdecken zum Beheben)
- Fehlertrendanalyse (Es sollten später weniger Fehler gefunden werden, weil sie Qualität steigt)
- Qualitätsrate (Anteil der erfolgreichen Tests)
- Korrekturqualität (Anteil der wiedereröffneten Fehler)
- Fehlerdichte (wird z.B. pro Modul oder Test ermittelt, um Problemkandidaten zu erkennen)
- Testeffektivität (wird im Nachhinein ermittlet, da sie die später gefundenen Fehler zählt)

Es wird in [1] auch noch eine *Testautomatisierungsmetrik* vorgeschlagen, die den Nutzen der Testautomatisierung beziffern soll[3]. Allerdings lässt sich dieser eigentlich nicht messen, und eine versuchte Messung wird sicherlich kein reales Bild abgeben können, sie soll daher hier nicht weiter erwähnt werden.

3.3.6 6. Bewertung des Testprogramms und -prozesses

Nach dem Abschluss des Testens muss die Leistungsfähigkeit des Testprogramms überprüft werden, um den Prozess verbessern zu können. Es gilt also, aus den Fehlern zu lernen, um im nächsten Projekt oder der nächsten Testphase (Nachfolgeprodukt, nächste Version, anderes Projekt) davon profitieren zu können. Das soll natürlich nicht die Möglichkeit ausschliessen, in der laufenden Testphase solche Verbesserungen einzuführen, wenn sie noch sinnvoll sind.

In dieser Phase sind die im vorigen Verlauf, hauptsächlich aber Phase 5 gesammelten Metrikdaten sinnvoll, um weniger offensichtliche Probleme zu identifizieren. Die Testeffektivität

[3] Trotz der betonten Wichtigkeit einer solchen Metrik haben die Autoren sich mit 8 Zeilen Erklärung begnügt, jedoch fast eine Seite die (Definitions-)Schwierigkeiten einer solchen Metrik besprochen. Siehe [1], S.438

kann möglicherweise jetzt ebenfalls ermittelt werden. Andere Probleme, die Teammitgliedern aufgefallen sind, müssen ebenfalls identifiziert werden. Vielleicht gab es nachträglich grössere Verfahrensumstellungen, oder währen sinnvoll gewesen, wurden aber ausgelassen. Herauszufinden ist ebenfalls, ob oder wie weit man den Testplan eingehalten hat, ob bestimmte Phasen über- oder unterschätzt wurden.

Es macht wenig Sinn, mit dem Dokumentieren dieser Umstände bis in diese Phase zu warten, vielmehr sollten die Teammitglieder solche Dinge frühzeitig Dokumentieren, solange man noch reagieren kann. Vielleicht muss man sich dann auch keine Gedanken über die zukünftige Vermeidung des Misstände machen, weil sie bereits erfolgreich behoben wurden. Die Verbesserung in dieser Phase ist immer die 2. Wahl.

Jetzt müssen die erkannten Probleme reflektiert werden, Lösungsvorschläge und -strategien entwickelt werden. Diese sollten für jeden leicht zugänglich dokumentiert sein - natürlich auch die während des Testzyklus entstandenen Erkenntnisse.

3.4 Ansatz zum Testen von Anforderungen

Das Testen von Anforderungen, wie es hier kurz vorgestellt wird, ist ein Beispiel für einen formalisierten Test. Für Details siehe [1], Anhang A.

Softwareprojekte, die auf ungenauen Anforderungen basieren, können leicht scheitern. Tests wie die Vorgestellten werden daran nur äusserst selten etwas ändern, da sie ebenfalls auf den ungenauen Anforderungen basieren. Das hier vorgestellte Verfahren versucht die ungenauen Anforderungen zu erkennen etwa wie Falschaussagen in einer Ermittlungssache erkannt werden: Durch ihre immanenten Widersprüche. Zum Glück scheinen Anforderungen nicht über die Komplexität eines Kriminalfalles zu verfügen, denn dort kommt man mit formalen Methoden nicht so weit.

Anforderungen unterliegen einem gewissen Eigenleben. Sie werden in Projekte eingebracht und gestrichen, man erkennt sie nicht rechtzeitig, sie 'wahren nie vorhanden'. Um solche Umstände zu minimieren, müssen Kontext (siehe Fussnote 1, Seite 6) und Anforderungen systematisch auf Schwachstellen geprüft werden.

3.4.1 Anforderungsanalyse

Zuerst müssen möglichst alle Anforderungen erkannt werden: Mittels Prototypen, Interviews, Datenanalysen u.ä. wird versucht, alle Anforderungen 'einzufangen'. Dabei hilft es, die Anforderung zusammen mit Zweck, Priorität, Besitzer (Aufsteller) und Verweisen zu verwandten Anforderungen aufzuzeichnen.

Jetzt wird ein Qualitätstor eingeführt, das jede Anforderung passieren muss, um in die Spezifikation zu gelangen. Dieses ist durch einige Tests definiert, die im folgenden kurz angerissen

werden. Sie sollen die Integrität und Genauigkeit der Anforderungen sicherstellen und sind so Entworfen, dass sie meistens auf alle Anforderungen gleichzeitig angewendet werden können.

Qualitätsmass

Das Qualitätsmass ist eine Möglichkeit, eine konforme Lösung eindeutig von einer Nichtkonformen abzugrenzen. Ein Solches muss also vorhanden sein, damit die Anforderung den Test besteht.

Nicht quantifizierbare Anforderungen sind dabei problematisch, etwa 'leicht zu erlernen'. Hier muss man sich auf ein Qualitätsmass einigen, wobei bei der resultierenden Reflektion oft auch die Anforderung konkretisiert werden kann.

Kommt keine Einigung oder Konkretisierung zustande, kann man Versuchen die Anforderung durch mehrere zu ersetzen, die über Qualitätsmasse[4] verfügen.

Das Qualitätsmass sollte ebenfalls mit der Anforderung aufgezeichnet werden, damit es von allen Seiten verifiziert werden kann.

Verständlichkeit und Konsistenz

Anforderungen müssen von allen Beteiligten auf die gleiche Weise interpretiert werden. Die darin verwendeten wichtigen Formulierungen (Objekte, Vorgänge, ...) müssen *konsistent in allen Anforderungen* verwendet werden. Es ist also wichtig, eine präzise Sprache zu wählen und einen Glossar zu erstellen.

Vollständigkeit

Der Kontext definiert das zu lösende Problem, und viele Anforderungen, die im Nachhinein gestellt werden, kommen aus dem Kontext. Natürlich kann es auch sein, das der Kontext nachträglich erweitert wird, aber auch das könnte nur notwendig geworden sein, weil auf einem fehlerhaften Kontext aufgebaut wurde. Diese Fehler sollen eingeschränkt werden, indem der Kontext auf Vollständigkeit untersucht wird.

Der häufigste Fehler ist die Begrenzung des Kontextes auf den Teil, der automatisiert werden soll. Ohne Verständnis für Kultur und Arbeitsweise wird fast automatisch eine Software entstehen, die nicht den Bedürfnissen entspricht.

Man sollte also auch die Arbeit und die Arbeitsweise der mit der Software arbeitenden Personen verstehen und auf nötige Kontexterweiterungen untersuchen.

[4]Dieses schöne Wort ist natürlich die Mehrzahl von Qualitätsmass

Die eigentlich Frage ist: Ist der Kontext gross genug, um alles abzudecken, was verstanden werden muss? Das wichtigste Indiz ist, das auch Vorgänge berührt werden, die ausserhalb der Software ablaufen.

Weiter kann nach fehlenden Anforderungen gesucht werden. Dazu sind die beteiligten auf *unbewusste* Anforderungen zu befragen, die vom aktuellen System schon gelöst sind und deshalb leicht vergessen werden. Rechtliche oder Kulturelle Rahmenbedingungen sind oft auch eine Quelle unbewusster Anforderungen. Dann gibt es noch *vernachlässigte* Anforderungen, die nur nicht gestellt wurden, weil sie nicht für Lösbar gehalten werden.

Weiter kann nach Features ähnlicher Systeme gefragt werden oder es können Brainstormings durchgeführt werden, um vernachlässigte Anforderungen aufzudecken.

Relevanz

Nachdem soviel dafür getan wurde, alle Anforderungen aufzudecken, müssen die irrelevanten wieder über Bord. Manche Leute machen Anforderungen, weil man 'das ja brauchen könnte', oder wegen persönlicher Vorlieben.

Um auf Relevanz zu prüfen, kann man auch die vereinbarten Ziele bemühen, um festzustellen, ob die Anforderung zu den Zielen beiträgt oder nur Schmuckwerk ist. Die Priorisierung der Anforderung kann ebenfalls behilflich sein.

Anforderung oder Lösung?

Einige der irrelevanten Anforderungen sind eigentlich Lösungen, z.B. ist ein GUI eine Lösung eines Benutzerinterfaces (wenn auch wahrscheinlich die einzig Akzeptable). Wenn eine Lösung anstatt einer Anforderung gestellt wird, fehlt oft die Anforderung.

Manchmal ist eine scheinbare Lösung aber auch eine Anforderung, z.B. weil in keine andere Datenbank investiert werden soll und die Vorhandene entsprechend zu nutzen ist.

Bewertung durch Beteiligte

Hat jede Anforderung eine Priorisierung, mit der die Anforderungen beim Design gegeneinander abgewägt werden können? Wenn weniger wichtige Anforderungen nicht erfüllt werden, kann immer noch ein gutes System entstehen.

Verfolgung

Um die Anforderung im System verfolgen zu können, gibt man ihr eine eindeutige Nummer, die keine weiteren Eigenschaften der Anforderung abbildet (also schlicht Fortlaufend).

3.4.2 Ordnen der Anforderungen

Jetzt werden die Anforderungen, die im vorigen Abschnitt als separate Einheit behandelt wurden, gruppiert. Eine Gruppe zeichnet sich durch starke interne Verbindungen aus, die Gruppen sollten untereinander deutlich schwächer verbunden sein. Ereignisse oder Anwendungsfälle bieten eine gute Basis zum identifizieren der Gruppen.

Ziel ist es, die von einer Änderung in den Anforderungen betroffenen Gruppen und Systemteile einfach über die Verbindung zu Ereignissen bzw. Anwendungsfällen zu kennen. Dazu muss sichergestellt sein, dass jede Anforderung mit allen von ihr betroffenen Systemteilen verknüpft ist. Man muss also zu jeder hypothetischen Änderung in den Anforderungen die betroffenen Systemteile benennen können.

3.4.3 Schlussfolgerungen

Die hier vorgestellten Hilfsmittel können beim Zusammenstellen einer guten Anforderungsspezifikation helfen, ersetzen aber kein ausgeprägtes Kommunikationsvermögen. Wichtiger ist eigentlich die Erkenntnis, dass der Testbeginn mit dem Projektbeginn erfolgt, nicht irgendwann später. Sobald eine Anforderung bekannt ist, kann mit Tests angefangen werden.

Kapitel 4

Fazit

Die hier Vorgestellten Methoden bergen das Potenzial, Software sicherer und mit höherer Güte zu realisieren. Für moderne Projekte ist automatisiertes Testen ein muss, und ein gewisser Grad an Automatisierung des Testens ist in beinahe jedem Projekt vorhanden. Open-Source-Programme kommen oft mit einer Testsuite, zu fast allen modernen Standards existieren Möglichkeiten, eine Implementierung zu verifizieren. Testen ist also akzeptierter Teil der Entwicklung.

Für automatisiertes Testen gilt das nicht so umfassend, aber wer es wirklich braucht kommt ohnehin nicht darum herum[1]. Es kann die Qualität der Software erhöhen, die Produktionskosten senken, und die Fehlschlagswahrscheinlichkeit verringern. Aber natürlich nur bei konsequenter Anwendung.

Diese Möglichkeit soll durch die ATLM gegeben werden: Ein erfolgreiches Testprogramm auf Anhieb, das klingt zu schön um wahr zu sein. Trotz gelegentlich eingestreuter Kritik möchte der Autor hier seiner Einschätzung Ausdruck verleihen, das die ATLM durch ihren hohen Praxisbezug und die kompakten Erfahrungswerte tatsächlich in der Lage ist, die Wahrscheinlichkeit für eine erfolgreiche Einführung stark zu erhöhen.

[1] Andererseits kenne ich eine Organisation mit mehr als 500 Mitarbeitern, in der das Testteam aus einem Abteilungleiter besteht, der kurz vor der Endabnahme in die Produktion nochmal das Test-Thema auspackt, seine Tragweite zu realisieren beginnt und dann sagt: 'Ok, ich Unterschreibe'. Diese Organisation müsste nach allen vernünftigen Kriterien längst automatisiert Testen, hat es aber trotzdem bisher ohne geschafft.

Anhang A

Abbildungen

Abb. 3, 3.1 aus [1]

Literaturverzeichnis

[1] Dustin, E., Rashka, J., Paul, J., Software automatisch testen, Springer 2000