Softwarequalitätssicherung - Zusammenfassung

Christoph Redl

Zusammenfassung und Ausformulierung der Unterlagen zu „Softwarequalitätssicherung“ im Sommersemester 2007 an der TU Wien (QSE).

Inhaltsverzeichnis

1 Allgemeines

1.1 Begriffe

1.1.1 Qualität

Es gibt keine allgemein anerkannte Definition von Qualität. Ein Versuch Qualität zu definieren ist „Qualität ist die Eignung eines Produktes zur Erfüllung vordefinierter Anforderungen“.

Qualitätssicherung soll dafür sorgen, dass ein Produkt für den Anwender verwendbar, für den Professionalisten verständlich und änderbar und für den Betreiber effizient und administrierbar ist.

Qualität ist keine Eigenschaft des Produktes, die später hinzugefügt werden muss. Man muss sie bereits während der Entwicklung berücksichtigen.

1.1.2 Qualitätsfaktoren

Qualitätsfaktoren sind bestimmte die Qualität betreffende Forderungen, wie z.B. Effizient, Wartbarkeit, Wiederverwendbarkeit.

1.1.3 Qualitätssicherung

Qualitätssicherung ist eine Stelle im Unternehmen die sich mit allen die Qualität betreffende Maßnahmen beschäftigt. Dazu zählen beispielsweise die Vorbereitung und Durchführung von Reviews, die Vorbereitung und Durchführung von Produkttests, die Unterstützung bei der Personal- und Toolauswahl, lokale Standards bezüglich Qualität herstellen, etc..

Wichtig ist, dass die Qualitätskriterien direkt von der Entwicklungsabteilung stammen. Die Entwickler (bzw. der Auftraggeber) legen also fest welche Anforderungen an die Qualität gestellt weden. Die Qualitätssicherung stellt danach lediglich fest ob diese Vorgaben auch erreicht wurden und gibt (konstruktive) Maßnahmen vor, die den Entwicklungsprozess unterstützen sollen.

1.1.4 Qualitätsmanagement

Dies ist keine Stellt im Unternehmen sondern bezeichnet die Qualitätssicherung als Aktivität. Es sollten dabei Grundprinzipien verfolgt werden, zu dennen beispielsweise zählen:

- Mehraugenkontrolle
- Projektabhängige Planung
- Verwendete Qualitätssicherungmaßnahmen evaluieren (und nicht blind anwenden)
- Integrierte Qualitätssicherung (und nicht am Ende der Entwicklung)

- Rückkopplung der Ergebnisse (siehe Plan-Do-Check-Act-Konzept)
- Frühzeitige Fehlererkennung

1.2 Das Plan-Do-Check-Act-Prinzip

Dieses Prinzip sagt aus, dass die Erkenntnisse aus der Qualitätssicherung nicht nur im aktuellen Projekt genutzt werden sollten, sondern auch in der gesamten Arbeitsgruppe und im gesamten Unternehmen.

Nach dem Planen (Plan) und der Implementierung (Do) werden analytische Qualitätssicherungmaßnehmen (siehe unten) angewandt (Check). Die Erkenntnisse werden rückgekoppelt (Act).

1.3 Einteilung von Qualitätssicherungmaßnahmen

Grundsätzlich lassen sich Qualitätssicherungsmaßnahmen einteilen in analystische und konstruktive Maßnahen.

Analytische sind solche, die selbst nicht zur Qualitätsverbesserung führen, sondern nur den Ist-Stand ermitteln. Dazu zählen etwa Tests oder Reviews. Sie lassen sich weiter unterteilen in statische (kein lauffähiger Code erforderlich, z.B. Design Reviews) und dynamische (lauffähiger Code notwendig, z.B. Black-Box-Test) Maßnahmen.

Konstruktive Maßnahmen sind solche, die unmittelbar zur Qualitätsverbesserung beitragen. Sie lassen sich weiter unterscheiden in:

- Technische Maßnahmen: z.B. Verwendung guter Tools, Debugger, etc.
- Menschliche Maßnahmen: z.B. Mitarbeiter gut einschulen
- Organisatorische Maßnahmen: z.B. ein geeignetes Prozessmodell (RUP, Wasserfallmodell, V-Modell, etc.) anwenden

1.4 Qualitätsmodell nach McCall

Dieses Modell sieht eine stufenweise Verfeinerung von Qualitätsfaktoren vor.

Die Qualitätsfaktoren selbst sind sehr abstrakte Forderungen, die so nicht direkt implementiert werden können, etc. die Effizienz.

Qualitätskriterien sind Aufgliederungen eines Qualitätsfaktors in konkretere Beschreibungen, die während der Implementierung leichter berücksichtigt werden können (z.B. Effizient kann aufgeteilt werden in Speichereffizienz und Ausführungseffizienz).

Jedoch sind Qualitätskriterien immer noch zu abstrakt, weshalb es Metriken gibt. Das sind quantitative Messungen von Softwaremodulen die eine bestimmte Aussage bezüglich Qualität haben. Es gibt die unterschiedlichsten Metriken für verschiedene Qualitätskriterien.

1.5 Einteilung der Qualitätsfaktoren

Qualitätsfaktoren lassen sich grundsätzlich einem Stakeholder (das sind alle, die etwas mit dem Projekt zu tun haben: Entwickler, Betreiber, Benutzer, Auftraggeber, etc.) zuordnen. Kontrolliert werden sollten vor allem solche Qualitätsfaktoren, die für die Entwickler selbst nur eine geringe Bedeutung haben, da diese oft vernachlässigt werden.

Weiters lässt sich eine Unterscheidung zwischen Hygienefaktoren und Mehrwertfaktoren treffen. Hygienefaktoren sind solche, die unbedingt umgesetzt werden müssen da sonst das Gesamtsystem keinen Sinn mehr macht. Dagegen können Mehrwertfaktoren notfalls fehlen, jedoch wird durch deren Umsetzung der Nutzen für den Benutzer größer.

1.6 GQM-Modell

Dieses Modell befasst sich mit der Überprüfung, ob bestimmte Qualitätsfaktoren erfüllt wurden.

(G)oals sind Ziele, die mit man einer bestimmten Messung erreichen will. Z.B. „Man will feststellen, ob das Produkt effizient ist.“ (konzeptuelle Stufe)

(Q)estions sind Fragen, deren Beantwortung zum Erreichen des Ziel führt, z.B. „Wie lange dauert die Abwecklung eines bestimmten Geschäftsfalls?“. Es handelt sich also um eine Konkretisierung eines Ziels in Form einer Frage (operative Stufe).

(M)etriken sind nun Zahlenwerte, deren Kenntnis zur Beantwortung der Frage(n) führt, z.B. Laufzeitmessungen. (Quantitative Stufe)

2 Testen

2.1 Einleitung

Testen ist eine analytische Qualitätssicherungsmethode. Es können dadurch nur Fehler aufgedeckt werden, nicht aber die Fehlerfreiheit nachgewiesen werden. Deswegen sollte es beim Testen auch immer Ziel sein einen weiteren Fehler zu finden und nicht alle Fehler zu finden.

Testen ist eine schwierige Aufgabe, die nicht jeder durchführen kann. Etwa 40% des Gesamtaufwandes bei der Softwareentwicklung werden durch das Testen verursacht.

Man darf Testen weder vernachlässigen noch übertreiben. Irgendwann steigt nämlich bei zusätzlichen Tests der Testaufwand so stark an, dass die Nutzen dadurch übertroffen werden.

Das Lokalisieren von Fehlern ist nicht aufgabe des Testens!

Definition von Testen: „Testen ist das Ausführen eines Programms mit der Absicht Fehler zu finden“.

Definition von Fehlern: „Ein Fehler ist eine Abweichung zwischen dem Verhalten eines Programms und seiner Spezifikation“.

2.2 Begriffe

2.2.1 Testfall

Ein Testfall ist ein Quadrupen, bestehend aus: Vorbedingung(en), Eingabewert(e), Eingabeaktion(en) und erwartetem Ergebnis.

Es lassen sich Normalfälle (sollten bei korrekter Implementierung funktionieren), Fehlerfälle (sollten bei korrekter Implementierung zu einer Fehlermeldung/Abbruch führen) und Sonderfälle (an der Grenze zwischen Normal- und Fehlerfall).

2.2.2 Testfall-Sepzifikationsmethoden

Testfall-Sepzifikationsmethoden sind systematische Herangehensweisen bei der Erstellung von Testfällen. Im Vergleich zur intuitiven Definition von Testfällen ergeben sich folgende Vorteile:

- Wiederholbarkeit von Testfällen
- Besserere Testfallüberdeckung
- Unabhängigkeit zwischen dem, der spezifiziert und dem, der durchführt
- Effektiver auf bestimmte Fehlertypen anpassbar

2.2.3 Fehlerfindungsrate

Das ist die Wahrscheinlichkeit mit einer bestimmten Methode einen Fehler zu finden.

2.2.4 Fehlerlokalisierung

Einschränkung der Möglichkeiten für den Grund des Fehlers.

2.2.5 Testüberdeckung

Anteil der Use Cases, die von einem bestimmten Set von Testfällen abgedeckt werden.

Die Überdeckung kann durch eine höhere Komplexität oder eine größere Anzahl von Testfällen erhöht werden. Es sollte hierbei ein Tradeoff gewählt werden, weil weder eine zu hohe Komplexität noch eine zu hohe Anzahl an Testfällen vorteilhaft ist.

2.2.6 Generalität

Man spricht von Generalität wenn ein Testfall keine konkreten Werte enthält, mit denen getestet wird, sondern nur Verweise auf zu verwendende Äquivalenzklassen. die Werte werden erst bei der Testdurckführung inkludiert.

2.3 Fehler beim Spezifizieren von Testfällen

- Mehrere Testfälle die das gleiche testen (Redundanz)
- Vermischung von Eingabewerten und Ergebnissen
- Auslassen des erwarteten Ergebnisses
- Test von menschlichem Verhalten statt des Systems

2.4 Arten von Testmethoden

2.4.1 Black-Box-Tests

Die Testfälle werden ohne Berücksichtigung des Wissens über die interne Struktur einer Software spezifiziert. Man baut sie auf Basis der Spezifikation der Software auf.

Die Eingabewerte werden anhand der Use-Cases partitioniert, und nicht anhand der Verzweigungen im Programm. Überprüft wird letztendlich die korrekte Input-Output-Relation.

Daraus ergibt sich, dass die Fehlerstelle kaum bestimmbar ist (was aber ohnehin kein direktes Ziel von Tests ist). Der Test dient nur einer grundsätzlichen Funktionsüberprüfung.

Bezüglich der Überdeckung ist es Ziel, die Anwendungsfälle möglichst vollständig abzudecken.

2.4.2 White-Box-Tests

Hier ist Wissen über die Implementierung notwendig, weil die Testfälle direkt auf Basis des Codes erzeugt werden. Die Testdaten werden daher anhand der Verzweigungen im

Programm partitioniert.

Bezüglich der Überdeckung ist es Ziel, alle Pfade (einer bestimmten Länge) im Programm abzudecken.

2.5 Testfall-Spezifikationsmethoden

Grundsätzlich lässt sich unterscheiden zwischen funktionalen und strukturellen Spezifikationsmethoden. Funktionale Methoden (Äquivalenzklassenzerlegung, Grenzwertanalyse) korrespondieren mit Black-Box-Tests und strukturelle Methoden mit White-Box-Tests.

2.5.1 Äquivalenzklassenzerlegung

Es werden für jede Dimension der Eingabe Klassen von Werten gebildet, die gleiches oder ähnliches Verhalten zur Folge haben. Es gibt sosohl Klassen mit gültigen als auch welche mit ungültigen Werten. Es sollten nicht nur explizite sondern auch implizite (z.B. globale Variablen) Parameter berücksicht werden.

Jede Klassenkombination sollte zumindest einmal getestet werden.

Anmerkung: Die Klassen können sich durchaus überdecken. Es kann beispielsweise eine Klasse für die syntaktische und eine für die sematische Ebene erstellt werden. Bei (syntaktisch und sematisch) korrekten Eingaben, liegt die Eingabe dann in beiden Klassen gleichzeitig.

2.5.2 Grenzwertanalyse

Aufbauend auf einer Äquivalenzklassenzerlegung werden besonders solche Werte getestet, die sich an den Rändern einer Äquivalenzklasse befinden. Genau dort sind nämlich besonders oft Fehler zu finden.

2.5.3 Strukturtest

Ausgehend vom Ablaufdiagramm (Kontrollflussgraph) werden alle Pfadstücke einer bestimmten Länge identifiziert. (Anmerkung: Das Ablaufdiagramm sollte bei größeren Programmen methodenweise erstellt werden.) Für die Länge n erhält man dann eine Cn-Überdeckung.

Für jedes der Pfadstücke bestimmt man dann die Eingabedaten, die zum Durchlaufen dieses Pfadstücks notwendig sind und erhält so einen Testfall.

Es existiert für einen bestimmten Kontrollflussgraphen immer eine Minimenge and notwendigen Testfällen, um eine Cn-Überdeckung für ein bestimmtes n zu erhalten.

2.5.4 Datenzyklustest

Es wird die sogenannte CRUD-Matrix (Create, Read, Update, Delete) erstellt. Darin sind für jede Funktion und jeden Entitytyp die Werte C, R, U und D eingetragen, je nachdem welche Datenbankoperationen die entsprechende Funktion am entsprechenden Entitytyp durchführt.

Zur Bestimmung der Testfälle werden nun die Funktionen in solchen Reihenfolgen aufgerufen, dass die Kombinationen (C, R), (U, R) und (D, R) getestet werden, und zwar jeweils mti allen Funktionen und in allen möglichen Reihenfolgen. Damit die Funktionen die entsprechenden Aktionen durchzuführen müssen sie natürlich entsprechend parametrisiert werden.

2.5.5 Nicht-Funktionale Tests

Neben den oben genannten Testfall-Spezifikationsmethoden gibt es noch welche um Nicht-Funktionale Tests vorzubereiten. Diese entsprechen z.B. Tests der Skalierbarkeit (Erweiterbarkeit des Systems, d.h. langsame Steigerung der Last) und Stresstests (Last an einem hohen Level beginnen lassen und testen, ob das System die Anforderungen nach einer Zeit bewältigen kann).

3 Messungen

Messungen können wie folgt kategorisiert werden:

- Direkt vs. indirekt, je nachdem ob die Messung direkt am zu untersuchenden Objekt vorgenommen wird oder ob sie aus anderen (direkten) Messungen abgeleitet wird
- Objektiv vs. subjektiv, je nachdem ob jeder Beobachter zum gleichen Ergebnis kommen würde, oder ob seine persönliche Einstellung in das Ergebnis einfließt
- Qualitativ vs. quantitiv, je nachdem ob das Ergebnis in Zahlen ausdrückbar ist oder nicht (z.B. durch Text, Bilder, Interviews, etc.)

4 Reviews

Reviews Prozesse sind zum Zwecke der objektiven, strukturierten und qualitativen Begutachtung und Bewertung von Produkten. Es kann grundsätzlich alles reviewt werden, das im Verlauf eines Projektes produziert wird, also nicht nur ausführbarer Code, sondern auch bereits Designdokumente.

Das Ziel, das man mit Reviews erreich will ist die möglichst frühe Erkennung von Fehlern, da spät erkannte Fehler viel teurer sind.

4.1 Rollen bei Reviews

- Moderator: Leitet das Review, entscheidet aber nicht unbedingt (je nach Reviewtyp - siehe unten)
- Gutachter: Bringen Vorschläge ein was ihrer Meinung nach verbessert oder geändert werden muss
- Leser: Trägt das Review-Objekt vor
- Schreiber: Führt das Protokoll, das während der Reviewsitzung angefertigt werden soll
- Autor: Beantwortet Fragen des Review-Teams, rechtfertigt aber währenddessen nicht die Lösungen

4.2 Phasen einer Review

1. Planung: Ziele des Reviews feststellen (auch z.B. den Typ), Zeit- und Kostenplan aufstellen
2. Vorbesprechung: Den Gutachtern wird das Review-Objekt vorgestellt falls das notwendig ist (je nach Komplexität)
3. Intensive Einzeldurcharbeitung: Jeder Reviewer versucht für sich Fehler zu finden
4. Reviewsitzung: Die einzeln gefundenen Probleme werden zusammengetragen, darüber diskutiert und so eine gemeinsame Problemliste erstellt
5. Nacharbeit: Das reviewte Objekt wird vom Entwicklungsteam gemäß Problemliste überarbeitet
6. Reviewbericht: Abschließend wird ein Bericht über das Review erstellt
7. Nun kann wieder am Beginn angesetzt werden, falls das notwendig ist (nächste Iteration)

4.3 Arten von Reviews

Es gibt mehrere Arten von Reviews mit unterschiedlichen Zielsetzungen. Grundsätzlich lässt sich unterscheiden ob der Kunde beim Review dabei ist oder nicht.

4.3.1 Reviewtypen mit Kundenanwesenheit

- Requirements review: Die Anforderungen werden gemeinsam mit dem Kunden noch einmal durchgegangen um abzusichern dass das umgesetzt wird, das der Kunde auch tatsächlich haben will.

- Preliminary design review: Das Design wird dem Kunden vorgelegt um zu überprüfen ob er damit einverstanden ist.
- Critical design review: Der Kunde soll mitunterschreiben (bei kritischen Projekten) um formal sein Einverständnis einzuholen.
- In-Process Review

4.3.2 Reviewtypen ohne Kundenanwesenheit

- **Management Review**
- Code-Walkthroug Der Autor stellt seine Module ablauforientiert vor. Das eignet sich um gemeinsam mit den Reviewern die Qualität der Implementierung zu begutarchten und Verbesserungsvorschläge einzuholen. Weiters können neue Mitarbeiter dabei zusehen und sich so mit dem Projekt vertraut machen. Letztendlich entscheidet hier der Autor selbst ob bzw. was überarbeitet wird.
- **Technical Review** Dies ist bereits ein formaleres Review, bei dem auch noch ein Moderator sowie ein Protokollführer anwesend ist. Das Produkt wird mit dem Ziel begutachtet, die Stärken und Schwächen zu finden. Das Review-Teams gibt eine Empfehlung an das Management, das letztendlich entscheidet.
- **Inspection** Eine Inspection ist ein formales Review das speziell in frühen Phasen der Entwicklung durchgeführt wird. Dabei wird das zu reviewende Produkt von einem Leser Schritt für Schritt vorgetragen. Das Review-Team versucht schwere Fehler zu finden und ein Protokollführer macht Notizen dazu. Schließlich entscheidet der Moderator wie weiter verfahren wird.

4.4 Lesetechniken

Das zu reviewende Objekt kann auf verschiedene Arten gelesen werden, die den Review-Prozess unterstützen.

- Unstrukturiert: Intuitive Durcharbeitung
- Checklisten-orientiert: Anhand von Checklisten wird beurteilt ob das Objekt bestimmte im Vorfeld festgelegte Kriterien erüllt oder nicht
- Perspektivisches Lesen: Es wird das Objekt aus verschiedenen Sichtweisen betrachtet (z.B. Benutzer, Auftraggeber, etc.) und versucht, aus jeweils einer Sichtweise Fehler zu finden
- Usage-Based Reading: Es wird versucht zuerst in den wichtigen Use-Cases Fehler zu finden und erst später in den wenigen wichtigen (dazu ist aber natürlich vorher eine Priorisierung der Anwendungsfälle notwendig)

5 Automatisierung in der Qualitätssicherung

5.1 Test-Driven Development

Unter Test-Driven Development versteht man das Prinzip, dass Testfälle vor oder spätestens während der Implementierung spezifiziet werden. Sie können dann als formale Spezifikation der Software-Requirements gesehen werden. Somit kann bei der Implementierung darauf Rücksicht genommen werden, dass die Testfälle - und somit auch die Requirements - erfüllt werden.

Ähnlich wie beim Model-Driven-Development, bei dem man von Modellen ausgeht und daraus Code generiert, wird beim Test-Driven-Development ebenfalls von Modellen zum Testen ausgegangen. Hier sollte jedoch zur Einhaltung der kurzen Implementierungs- und Testphasen zwischen Modellentwicklung und Durchführung hin- und hergewechselt werden.

5.2 Separation of concerns

Verschiedene Aufgaben sollten so weit wie möglich getrennt werden damit sie von unterschiedlichen Personen durchgeführt werden können, die sich dann jeweils auf ihrem Gebiet spezialisieren können. Es ist nämlich unmöglich, dass sich jeder überall auskennt.

Bei funktionalen Features einer Software ist diese Trennung spätestens mit der Objektorientierung sehr gut möglich geworden: man achtet einfach beim Architektur-Entwurf daruaf, dass Funktionalität schön in Klassen aufgeteilt werden, die dann von verschiedenen Mitarbeitern implementiert werden können.

Bei nicht-funktionalen Features, wie etwa Security-Checks, ist das aber problematisch, da solche Features Aufrufe und Verzweigungen erfordern, die im ganzen Programm verstreut sind. Etwa erfordert ein Security-Check, dass in allen Methoden die vor unbefugtem Aufruf geschützt werden sollen, an erster Stelle eine Abfrage eingefügt wird.

Durch aspect orientated programming wird nun auch in solchen Fällen die Aufteilung ermöglicht. Über Konfigurationsfiles kann man Code bei der Übersetzung automatisch an verschiedenen Punkten (so genannte „jointpoints“) eingefügt werden. Die erwähnten Abfragen lassen sich somit in eigenen Klassen realisieren und bei entsprechender Konfiguration an allen relevanten Stellen einfügen. Die einzelnen Aspekte (auch „concerns“), die nun in separaten Klassen implementiert sind, lassen sich nun natürlich auch separat testen.

5.3 Granularitäten beim Testen

Jedes Feature das implementiert wird sollte auch testbar sein. Vom klassischen Konzept „Erst implementieren, dann testen“ ist man inzwischen abgekommen und bevorzugt

nun abwechselnd kurze Implementierungs- und Testphasen. Tests laufen dabei auf unterschiedlichen Granularitätsebenen ab.

Einerseits werden Unit-Tests durchgeführt, die einzelne Klassen testen. Dazu werden eigene Testklassen geschrieben, in denen sich neben Initialisierungs- und Finalisierungscode (zum Vorbereiten bzw. Abschließen eines Testfalls) eine oder mehrere Testmethoden gibt, die jeweils das Verhalten einer oder mehrerer anderer Klassen testen, indem sie deren Funktionalität nutzen und die Ergebnisse dann mit erwarteten Ergebnissen vergleichen.

Darauf bauen Modultest (Komponententests) auf. Schließlich folgen Integrations- und Systemtests, und schließlich folgt der Acceptance-Test (End To End Tests).

Man führt dabei die Testfälle von eher abstrakten Codetests immer mehr an die echte Benutzung des Systems heran. Keine der Granularitätsebenen kann dabei durch eine andere ersetzt werden, weil „Low-Level-Tests“ zu wenig mit der tatsächlichen Benutzung zu tun haben und umgekehrt „High-Level-Tests“ alleine Fehler zu spät erkennen würden (und außerdem keine Lokalisierung der Ursache ermöglichen würden).

Alle Tests können dabei sowohl funktionale als auch nicht-funktionale (z.B. Lasttest, Stresstest, etc.) Tests sein.

5.4 Hilfsobjekte beim Testen

Führt man Tests auf niedriger Ebene durch (z.B. Unit-Tests), dann möchte man zu viele Abhängigkeiten von anderen Systemteilen vermeiden. Einerseits aus Performancegründen, andererseits aber natürlich auch um Fehlerursachen besser lokalisieren zu können. Außerdem will man oft die „echten“ Systemteile nicht mittesten, weil sie einer Produktivumgebung angehören, die man natürlich nicht gefährden will.

Als Hilfsobjekte verwendet man daher Dummy-Objekte (als Füllwerte für nie benutzte Parameter, z.B. einfach NULL), Fake Objects (realsieren echte Funktionalität, jedoch vereinfacht, z.B. ohne Zugriffskontrolle), Stubs (simulieren nicht vorhandene Systemteile auf unterer Ebene) und Mocks (so wie Stubs, jedoch etwas komplexer).

5.5 Assertions

Assertions sind Zusicherungen im Programmcode, dass an einer Stelle eine bestimmte Bedingung erfüllt sein sollte. Ist sie das nicht, so weiß man, dass etwas nicht stimmt.

Man kann solche Assertions zum Beispiel in Unit-Tests nutzen. In der Initialisierungsmethode eines Tests werden Daten vorbereitet und in einem Testfall abgefragt. Weil man die Daten ja speziell für den Testfall vorbereitet hat weiß man welches Ergebnis die Abfrage bei einer korrekten Implementierung liefern sollte: das kann man mit einer

Assertion zusichern. Ist die Implementierung fehlerhaft, so wird die Assertion zu einer Exception führen und der Testfall daher scheitern.

5.6 Bibliotheken zur Unterstützung

5.6.1 JUnit

Ermöglicht das Schreiben von Unit-Testfällen.

5.6.2 HTTP/HTML-Unit

Simulieren einen HTTP- bzw. HTML-Client. Man kann dadurch z.B. bequem auf Websites zugreifen, was aber eher für Testzwecke gedacht ist und nicht für Produktivumgebungen. Vor allem dienen diese Bibliotheken dem Test von Webapplikationen.

5.6.3 JMeter

Diese Bibliothek kann verwendet werden um Performancetests durchzuführen. Es kann damit beispielsweise ein Webserver mit vielen Anfragen konfrontiert und das Verhalten beobachtet und gemessen werden. Diese Tools muss man aber verstehen bevor man sie einsetzt! Bei falscher Anwendung haben die Ergebnisse keine Aussagekraft.

5.7 Automatisierte Code-Quality-Checks

Auch die Qualität des Quelltextes selbst kann überprüft werden. Quellcode-Qualität zeichnet sich durch Aspekte wie ausreichende Dokumentation, Einhaltung von Namens- und Formatierungskonventionen, Vermeidung von dupliziertem Code und ähnlichem aus.

Natürlich kann man solche Messungen stören indem man einfach „Dummy-Kommentare“ einfügt. Es ist daher nicht sinnvoll von den Mitarbeitern einen bestimmten Prozentsatz and Dokumentation zu fordern, da es stark auf den Kontext ankommt ob das sinnvoll ist oder nicht.

5.8 Profiling

Bei lauffähigen Anwendungen lässt sich ein Profil erstellen. Das sind Berichte darüber wie oft eine Funktion aufgerufen wird, wie lange ein Durchlauf dauert, etc.. Entsprechende Tools hängen sich dazu in die Java-VM ein. Man kann diese Informationen nutzen um etwa die Performance zu verbessern (um gute Ergebnisse erzielen zu können muss man wissen wo die meiste Zeit verbraucht wird).

5.9 Continuous Integration

Darunter versteht man die Zusammenschaltung vieler Enwicklungstools zu einem ganzen Entwicklungssystem, das Routineaufgaben periodisch und automatisiert durchführen

kann. Damit werden tägliche Builds erstellt, automatische Testläufe durchgeführt (Unit-Tests) und Benachrichtigung an die Teammitglieder versendet.

Wichtig dabei ist, dass auch das Repository, in das der Sourcecode eingecheckt wird, gekoppelt ist. Nach einem Check-In sollte automatisch ein Build erzeugt werden. Wenn man Maven manuell anstoßen muss, dann ist das noch nicht „Continuous Integration“. „Cruise Control“ und „Apache Continuum“ sind entsprechende Tools, die den ganzen Vorgang unterstützen.

Auch Bug-Tracking-Tools (mit denen alle Stakeholder gefundene Bugs bekanntgeben können) sollten berücksichtigt werden.

6 Quellen

- Folien zur Vorlesung „Software Qualitätssicherung“ im Sommersemester 2007 - QSE, TU Wien

- Deutsche und englische Wikipedia