6 Testen von Software

Testen ist eine besonders wesentliche Methode zur Qualitätssicherung in der Software-Entwicklung. Weitläufig wird Testen als „Ausführung des Programms mit dem Ziel, Fehler zu finden“ gesehen (zu einer genaueren Definition siehe Kapitel 1). Es erfüllt im Entwicklungsprozess allerdings gleich zwei wichtige Funktionen:

- Messung der Produktqualität (siehe Kapitel 4) zur Steuerung des Entwicklungsprozesses
- Finden von Fehlern zur Verbesserung der Produktqualität

Die Messung der Produktqualität ist eine notwendige Voraussetzung für die Verbesserung der Produktqualität: um sichergehen zu können, dass eine Verbesserungsmaßnahme Erfolg hatte, muss der Zustand vorher und nachher gemessen und verglichen werden. Aus der Anzahl der gefundenen Fehler lässt sich auf die Produktqualität schließen (siehe Kapitel 1). Dies erklärt den Einfluss des Testens auf den Erfolg von Software-Projekten. Analoges gilt natürlich für jede Art von Produkt, insbesondere Prozesse und ist eine wesentliche Grundlage der Prozessverbesserung (siehe Kapitel 12).

Die geschichtliche Entwicklung auf dem Gebiet der Software-Entwicklung weist einen Übergang von äußerst einfachen Systemen zu Beginn bis zu den heutigen, sehr komplexen Systemen auf. Die frühesten Testmethoden unterscheiden sich daher sehr stark von heutigen: wo früher mit „brute force“, d.h. ohne Methodik oder unter sturer Einhaltung von Methoden, gearbeitet werden konnte, führt heute nur noch intelligente Planung und Strategie zum Erfolg. Dieser Unterschied ist allerdings nicht nur geschichtlich verankert: auch heute ist er zwischen kleinen und mittleren bis großen Projekten, zwischen jungen und erfahrenen Entwicklern, Testern und Unternehmen zu sehen.

Das Kapitel 6 bringt einen kurzen Überblick über Testen und ökonomische Aspekte davon, über die geschichtliche Entwicklung des Testens, sowie zwei Beispiele zu kleinen und mittleren bis großen Systemen, um den Unterschied zwischen fehlender und vorhandener Planung zu verdeutlichen. Einen wesentlichen Einfluss auf die Testplanung hat die Ökonomie, was Kapitel 8 ausführlicher erläutert wird. Die Strukturierung des Testprozesses wird in Kapitel 7 beschrieben, die in der Testdurchführung anwendbaren Testmethoden in Kapitel 9 und 10.

Tests sollten im allgemeinen von ausgebildeten Testern durchgeführt werden. Die in vielen Betrieben aus Kostengründen durchgeführte Praxis, dass Entwickler ihre eigenen Programme testen, ist wegen der eintretenden „Betriebsblindheit“ nur in beschränktem Maße effektiv. Näheres zu den am Testprozess beteiligten Personen ist in Kapitel 7 zu finden.

In vielen Software-Entwicklungsmodellen ist Testen als abschließende Tätigkeit zur Verifikation der Korrektheit des Produkts vorgesehen. Testen ist allerdings ein den gesamten Entwicklungsprozess begleitender Prozess. Die Positionierung von Testen im Entwicklungsprozess wird in Kapitel 7 näher erläutert.

6.1 Motivation und Begriffsdefinitionen

Qualitätsansprüche für Software hängen sehr stark davon ab wie groß die Bedeutung der Verwendung des IT-Systems ist. Je größer die strategische und kommerzielle Relevanz des IT-Systems für eine Firma bzw. Organisation ist und je mehr Schaden bei einem Fehlverhalten verursacht werden kann, desto wichtiger ist es, Fehler noch vor dem Einsatz des Systems zu finden.

Alle Fehler kosten Geld. Nicht gefundene Fehler, sowie Fehler, die erst spät im Entwicklungsprozess gefunden wurden, sind die teuersten Fehler von allen (siehe Abbildung 6.1). Wenn sie nur in einer späteren Phase der Entwicklung gefunden werden, verursachen sie teure Nacharbeiten. Wenn sie nicht gefunden werden, können sie Fehler im Betrieb mit schweren finanziellen und rechtlichen Konsequenzen verursachen, und (bestenfalls) hohe Betriebskosten des Systems [Kit, 1995]. Mit Glück, und wenn der Wartungsvertrag es so besagt, muss der Kunde den Software-Hersteller bezahlen, damit dieser den Fehler korrigiert. Egal welcher Fall eintritt, verliert der Software-Hersteller an Reputation. Die Kosten davon sind nicht mit objektiven Methoden zu schätzen, können aber sehr hoch sein.

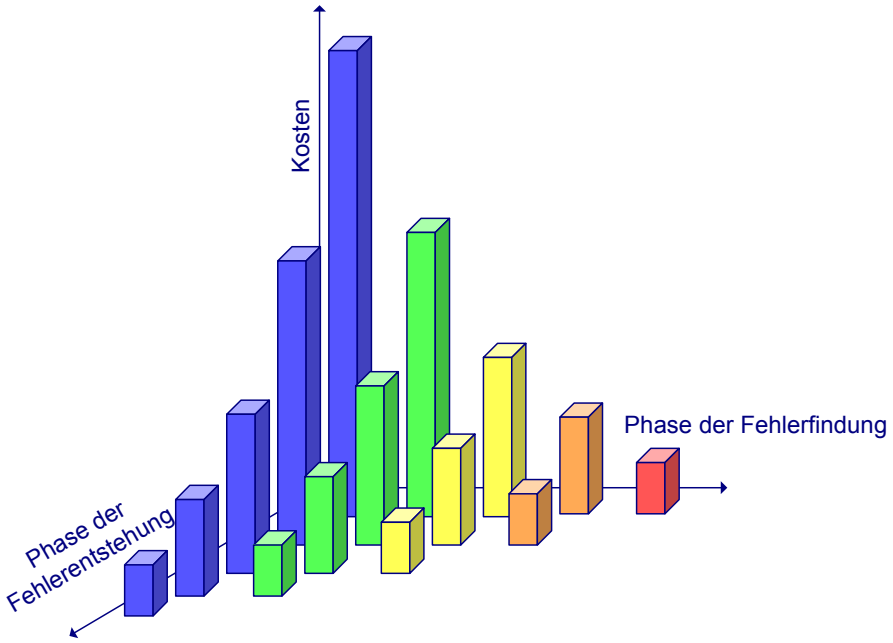

Abbildung 6.1: Kosten eines Fehlers in Abhängigkeit von der Phase, in der er gemacht wurde, und der Phase, in der er entdeckt wird [Boehm, 1982]

Die Kosten, die von Fehlern verursacht werden, hängen ab von der Phase, in der Fehler gemacht wurden, und der Phase, in der sie gefunden wurden, wie in Abbildung 6.1 dargestellt (angepasst von [Boehm, 1982]). So verursachen z. B. Fehler im Software-Design, die während der Implementierung gefunden werden, mehr Korrektur-Aufwand als reine Implementierungsfehler.

Die meisten Fehler werden allerdings früh im Entwicklungsprozess gemacht: mehr als 50 % aller Fehler geraten für gewöhnlich allein während der Anforderungsanalyse ins System [Kit, 1995].

Testen hat zwei Primär-Ziele: das erste ist es, Fehler zu finden, damit die Qualität des Software-Produkts durch Korrektur der Fehler erhöht wird. Je früher wir einen Fehler finden, desto größer ist der Nutzen, den wir durch Ersparnis der Fehlerkosten haben. Das zweite Ziel ist es, die Unsicherheit bezüglich der Qualität des Software-Produkts zu minimieren. Aus der Anzahl der gefundenen Fehler kann die Anzahl der noch vorhandenen Fehler im Produkt, d.h. dessen Qualität, geschätzt werden (siehe Kapitel 1), sowie Kosten, Dauer etc. des Software-Projektes [Pol et al., 2000]. Testen kostet Zeit und Geld, aber es gibt auch ein mehr oder weniger detailliertes Bild der Qualität des Software-Produkts. Man möge sich erinnern: “You can’t control what you can’t measure” [DeMarco, 1982].

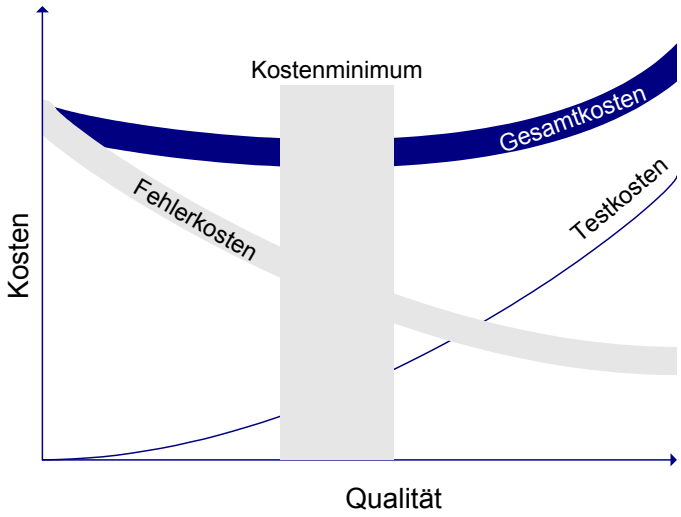

Abbildung 6.2: Beziehung zwischen Fehlerkosten und Qualitäts-Kosten

Die Schwierigkeit besteht darin, den Punkt zu treffen, wo die Summe von Testkosten und geschätzten Fehlerkosten ein Minimum erreicht[1]; Abbildung 6.2 stellt das schematisch dar. Man beachte, dass das Minimum nicht ein einziger Punkt ist, sondern durch einen ganzen Bereich gegeben ist. Dies ergibt sich vor allem daraus, dass die Fehlerkosten nur geschätzt und nicht genau berechnet werden können.

Wenn wir die Qualität des Produkts sowie die bereits angefallenen Kosten für Tests und Fehlerkorrekturen kennen, können wir sagen, wann Tests und weitere Qualitätssteigerungen nicht mehr ökonomisch tragbar werden. Dies erlangt gerade in der heutigen Zeit besondere Bedeutung:

[1] Allerdings ist oft eine gewisse Mindest-Qualität erforderlich, um am Markt konkurrenzfähig zu sein. Wenn diese Qualität noch nicht erreicht wurde, müssen qualitätssichernde Maßnahmen auch weiter als bis zum Minimum der Gesamtkosten durchgeführt werden.

die Qualitätsansprüche steigen, während vom Markt immer kürzere Realisierungszeiträume verlangt werden, bei geringen Kosten und begrenzten zur Verfügung gestellten Arbeitskräften.

Aus der Graphik lässt sich auch darauf schließen, wie Tests verbessert werden können: wenn Testen weniger teurer wäre, könnte man mehr testen, bevor der Punkt der minimalen Gesamtkosten erreicht wird. Somit ließe sich die Qualität erhöhen. Ein anderer Weg wäre, die Kosten für Fehlerkorrekturen zu senken. (Prozess-)Optimierung wird in Kapitel 12 kurz behandelt.

Um Testkosten so gering wie möglich zu halten, ist es unbedingt notwendig, den Testprozess zu strukturieren (siehe Kapitel 7), die höchsten Qualitätsrisiken zu identifizieren, die verschiedenen Aufgaben der Qualitätssicherung und des Testens zu priorisieren und zu planen (siehe Kapitel 8) und die richtigen Testmethoden anzuwenden (siehe Kapitel 9 und 10).

Nur kontrolliertes, ziel-orientiertes Testen ist effizient genug um ökonomisch tragbar zu sein. Der Gebrauch von Test-Werkzeugen kann Testkosten weiter verringern. Es gibt viele verschiedene Werkzeuge für verschiedene Phasen im Testprozess, z. B. Testdatengeneratoren, Werkzeuge zur Fehlerverfolgung, oder funktionale Test-Werkzeuge. Für den effizienten Gebrauch solcher Werkzeuge ist es sogar noch wichtiger, einen strukturierten Testprozess zu verfolgen, denn, wie das Sprichwort sagt: „*automatisiertes Chaos führt zu schnellerem Chaos*“. Kapitel 11 geht näher auf den Einsatz von Test-Werkzeugen ein.

6.1.1 Historische Beispiele

Anhand der folgenden Beispielen soll gezeigt werden, dass es Softwarefehler immer geben kann (vor allem aufgrund von Einsparungsmaßnahmen) und welche schwerwiegenden Folgen (Verluste und Schadenersatzforderungen in Milliarden Euro oder USD) diese haben können.

Beispiel 1 – Explosion der Ariane

Das wohl bekannteste Beispiel ist die Explosion der Ariane 5 am 4. Juni 1996: *„... im Verbund mit einer fälschlicherweise vom Rechner als Steuerungsdaten interpretierten Statusmeldung hat dazu geführt, dass die Rakete Ariane 5 über das fehlerhafte Programm mit einer Kurskorrektur bedacht wurde. Diese führte dann innerhalb der Rakete zu Gegenmaßnahmen in Form einer Schubkorrektur zur Gegensteuerung und letztendlich zur Aktivierung der automatischen Selbstzerstörung. (...) Eine nachträgliche Simulation konnte die Ursache exakt reproduzieren. Vor dem Start hatte das ESA-Team von einer solchen Simulation abgesehen – aus Kostengründen.*“ (Zitat aus der Zeitschrift iX, Sept. 1996, S.32) Der Bordcomputer stürzte 36.7 Sekunden nach dem Start ab als er versuchte, den Wert der horizontalen Geschwindigkeit von 64 Bit Gleitkommadarstellung in 16 Bit signed Integer umzuwandeln. Die entsprechende Zahl war größer als 2^{15} = 32768 und erzeugte den Overflow. Das Lenksystem brach ab und gab die Kontrolle an ein Backup-System. Dieses lieferte allerdings den gleichen Fehler, da es die gleiche Software benutzte. Entwicklungskosten betrugen ca. 7 Milliarden Dollar und der Verlust für die Rakete und Satelliten war ca. 500 Millionen Dollar. Folgende Punkte sind in diesem Beispiel wohl besonders hervorzuheben:

- Die Software stammte von der Ariane 4, aber die Ariane 5 flog schneller und daher trat der Fehler erst dann auf. Ein typische Annahme, dass eine bewährte Software immer funktioniert, stellte sich als folgeschweren Fehler heraus.

- Bei der Programmierung war die Umwandlung nicht abgesichert, weil man glaubte, dass die Zahl nie so groß sein könnte. Mögliche Fehlerfälle werden bei der Entwicklung zu selten berücksichtigt, weil sie meist mit einer höheren Entwicklungszeit und damit höheren Kosten verbunden sind. Auch unrealistische Testfälle können zu realen Fällen werden!
- Softwarefehler können nicht durch Backup-Systeme sichergestellt werden – vor allem dann nicht, wenn das selbe Programm darauf läuft).

Beispiel 2 - Intel Pentium Divisionsfehler

Im Spätsommer 1994 fand ein amerikanischer Professor beim Intel Pentium Prozessor einen Divisionsfehler. Beim Chip-Hersteller Intel war der Fehler bereits seit Mitte des Jahres bekannt – das Problem wurde heruntergespielt und als unwichtig beurteilt. Nach Protest-Publikationen im Internet am 30. Oktober 1994 verbreitete sich die Nachricht wie eine Lawine über mehrere Newsgroups hinweg, bis sie zu den Massenmedien im November 1994 fand. Als Stellungnahme kam von Intel, dass man ihnen beweisen müsse, dass der Fehler relevant sei und man nur dann zum Austausch bereit sei. IBM als einer der Kunden von Intel startete daraufhin interne Tests und kam auf das Resultat, dass der Fehler durchschnittlich alle 24 Tage auftreten könnte, im Gegensatz zur Behauptung von Intel: „einmal in 27000 Jahren“. Am 24. Dezember 1994 stellte IBM einen Austauschplan auf und es wurde versichert, dass keine dieser fehlerhaften Chips bei IBM mehr eingebaut wurden. Die Folge war, dass die Aktien von Intel um mehrere Prozentpunkte fielen, und der Handel kurzfristig ausgesetzt wurde. Eine Entschuldigung von Intel war die Folge und es wurde ein kostenloser Austausch „auf Wunsch“ angeboten, um die Kunden zu beruhigen. Der Verlust betrug ungefähr „nur“ 400 Millionen Dollar, da viele vom Angebot nicht Gebrauch machten. In weiterer Folge wurde am Image gearbeitet, und neuen Fehlereinmeldungen kam man mit Kommunikation und Kooperation entgegen anstatt die Meldungen zu ignorieren.

6.1.2 Richtlinien aus praktischer Erfahrung

In diesem Kapitel sind einige Zitate aus der Literatur angeführt, die wichtige Fakten bezüglich Testen auf den Punkt bringen, und die dabei behilflich sind, das Problem Testen aus der richtigen Perspektive zu sehen. Spätestens bei Lektüre des gesamten Buches sollte die Bedeutung aller Zitate klar werden.

Software-Entwicklung und Testen

“*Most software developers don't have such knowledgeable and precise customers. For them, the measure of their products' and services' quality is the satisfaction of their customers, not the match to a specification.*” [Kaner et al., 1999] – Seien Sie flexibel. Setzen Sie einen Software-Entwicklungsprozess und Testprozess ein, der leicht an änderbare Anforderungen anpassbar ist. Versuchen Sie, den Kunden zu befriedigen.

“*Software products are never released – they escape!*” [Kaner et al., 1999] – Die meisten Projekte enden zu einem vorher festgesetzten Termin, nicht erst dann, wenn durch Tests festgestellt wurde, dass das Produkt fertig ist.

"*Testing is an extremely creative and intellectually challenging task*". [Myers, 1979] – Glauben Sie nicht, dass Testen eine einfache, langweilige Arbeit für wenig qualifizierte Leute ist. Setzen Sie professionell arbeitende Menschen mit den richtigen Qualifikationen ein.

"*The objective of testing is to prove that there are bugs and not to prove that the program is bug-free.*" [Kaner et al., 1999] – Verfolgen Sie dieses Ziel. Wenn Sie die Aufgabe bekommen, ein Programm zu testen, versuchen Sie, es zum Scheitern zu bringen.
"*If you want and expect a program to work, you will be more likely to see a working program – you will miss failures. If you expect it to fail, you'll be more likely to see the problems. If you are punished for reporting failures, you will miss failures. You won't only fail to report them – you will not notice them.*" [Kaner et al., 1999] – Ermutigen Sie das Testpersonal, Fehler zu finden. Behandeln Sie sie nicht schlecht, nur weil Sie nicht mögen, was sie finden.

„*Nicht das, was wir nicht wissen, bringt uns zu Fall, sondern das, was wir fälschlicherweise zu wissen glauben.*" [DeMarco, 1998] – Vertrauen Sie nicht Fakten, die Sie bloß annehmen. Bleiben Sie insbesondere durch Tests informiert über die Qualität ihres Produkts.

„*Sie können Ihre Gesamtleistung stärker durch Eindämmen der Misserfolge als durch Optimieren der Erfolge verbessern.*" [DeMarco, 1998] – Testen ist ein Weg, das Risiko von Verlusten zu minimieren.

Testplanung und Testmanagement

"*Do not plan a testing effort under the tacit assumption that no errors will be found.*" [Kaner et al., 1999] – Das Ziel des Testens ist es, Fehler zu finden, und wenn es richtig ausgeführt wird, wird es das auch tun. Planen Sie immer Korrektur- und Nachtest-Zyklen ein.

"*A test plan is a valuable tool to the extent that it helps you manage your testing project and find bugs. Beyond that, it is a diversion of resources.*" [Kaner et al., 1999] – Betrachten Sie den Testplan nicht als heiliges Objekt, das die kleinsten Details beinhalten muss und ständige Aufmerksamkeit benötigt. Es ist ein grobes Werkzeug, nichts weiter.

"*A programmer should avoid attempting to test his or her own program.*" [Beizer, 1984] – Wenn ein Programmierer Fehler im eigenen Programm finden könnte, warum sollte er sie dann gemacht haben? Falls Mangel an spezialisierten Testern es absolut unumgänglich macht, dass Programmierer testen, lassen Sie sie Module von jeweils anderen testen, nicht ihre eigenen.

"*A programming organization should not test its own programs.*" [Beizer, 1984] – Das oben genannte gilt auch für ganze Organisationen.

Teststrategie

"*If you think you can completely test a program once, great. Can you completely test it ten times?*" [Kaner et al., 1999] – Umfassende Tests sind kostspielig. Verwenden sie eine Teststrategie um den Testaufwand auf die wichtigsten Komponenten und Qualitätskriterien des Systems zu konzentrieren.

"*The sooner a bug is found and fixed, the cheaper.*" [Kaner et al., 1999] – Versuchen Sie, Fehler so früh wie möglich zu finden. Verwenden Sie Reviews und andere Verifikationsmethoden früh im Softwareentwicklungsprozess, und verwenden Sie eine Teststrategie.

"*During each testing cycle, be conscious of the tradeoff between depth and breadth of testing.*" [Kaner et al., 1999] – Verlieren Sie sich nicht im Test einer einzigen Komponente unter Vernachlässigung anderer. Versuchen Sie, zuerst schwerwiegende Fehler zu finden, und testen Sie unwichtige Details zum Schluss.

"*The probability of the existence of more errors in a section of a program is proportional to the number of errors already found in that section.*" [Myers, 1979] – Falls eine Person einen Fehler bei der Analyse, im Design oder in der Implementierung gemacht hat, hat das meistens die Ursache darin, dass die Komponente komplex oder schlecht verstanden ist, oder das sich Anforderungen an die Komponenten oft geändert haben. Das macht es wahrscheinlich, dass noch weitere Fehler in dieser Komponente zu finden sind.

Testfalldefinition

"*A good test case is one that has a high probability of detecting an as-yet undiscovered error.*" [Myers, 1979] – Definieren Sie Testfälle so, dass keine zwei dieselben Fehler finden. Definieren Sie keine Testfälle, deren einziger Zweck es ist, zu zeigen, dass ein bestimmter Pfad durch die Applikation richtig funktioniert.

"*A successful test case is one that detects an as-yet undiscovered error.*" [Myers, 1979] – Nennen Sie Testfälle, die keine Fehler gefunden haben, nicht "erfolgreich", es sei denn, Sie wollen, dass Tester viel "Erfolg" haben.

"*Test cases must be written for invalid and unexpected, as well as valid and expected, input conditions.*" [Beizer, 1984] – Der Endbenutzer wird sehr wahrscheinlich viele ungültige und unerwartete Eingaben verwenden, also seien Sie darauf vorbereitet.

"*A necessary part of a test case is a definition of the expected output or result.*" [Beizer, 1984] – Wenn kein Ergebnis spezifiziert ist, tendieren Personen dazu, das zu sehen, was sie sehen wollen, nämlich ein korrekt funktionierendes Programm.

"*Avoid throw-away test cases unless the program is truly a throw-away program.*" [Beizer, 1984] –Wenn Sie explorative Tests verwenden, schreiben Sie die Schritte sowie die Ergebnisse nieder. Diese Testfälle können später wiederverwendet werden.

"*Examining a program to see if it does not do what it is supposed to do is only half of the battle. The other half is seeing whether the program does what it is not supposed to do.*" [Beizer, 1984] – Inkludieren Sie Testfälle, die zusätzliches, ungewolltes Verhalten des Programms aufdeckt.

Test-Vorbereitung

"*Back up data files before attempting to replicate a bug. Never, never, never use the original source of the data. Always use copies.*" [Kaner et al., 1999] – Ansonsten können Änderungen oder korrupte Daten den Fehler nur einmal reproduzierbar machen.

Test-Ergebnisse

"*Thoroughly inspect the results of each test.*" [Beizer, 1984] – Nehmen Sie nicht an, dass ein Ergebnis korrekt ist, nur weil es auf den ersten Blick so aussieht. Ignorieren sie nicht „kleine Unannehmlichkeiten". Schreiben Sie detaillierte Fehlerberichte.

"*A problem tracking system exists in the service of getting the bugs that should be fixed, fixed. Anything that doesn't directly support this purpose is a side issue – especially nice management statistics.*" [Kaner et al., 1999] – Wenn Sie ein Tool zur Problem-Verfolgung verwenden, verschwenden Sie nicht Ihre Ressourcen, indem Sie versuchen, es perfekt und für alles verwendbar zu machen.

Test-Verbesserung

„*Es gibt keine Schnellschüsse zur Verbesserung der Produktivität.*" [DeMarco, 1998] – Solche Maßnahmen sind meist kontraproduktiv.

6.1.3 Begriffsdefinitionen

Einige Begriffe aus dem Themenbereich Testen werden in der Literatur nicht einheitlich verwendet. Um Missverständnissen vorzubeugen, wird deren Bedeutung in diesem Buch hier erläutert.

Testfall

Ein Testfall besteht aus einer Menge von Vorbedingungen (V), einer Menge von Eingabedaten an das zu testende (Teil-)System (E), einer Menge von Aktionen, die zur Eingabe der Daten notwendig sind (A), und einer Menge von erwarteten Ergebnissen (R). Ein Testfall ist also ein Tupel (V, E, A, R).

Typische Vorbedingungen sind z.B. der Zustand des System zum Zeitpunkt der Ausführung des Testfalls oder geforderte Inhalte in einer Datenbank. Die Eingabedaten sind dann eine Menge von Daten, die an die Eingabeschnittstelle des Systems geschickt werden. Die Menge der Aktionen beschreibt dann, in welcher Art und Weise die Schnittstelle angesprochen wird. Dies kann z.B. durch Aufruf einer API-Funktion mit bestimmten Parametern oder durch Eingabe über ein graphisches Interface geschehen. Die erwarteten Ergebnisse sind diejenigen Reaktionen des Systems, die laut Spezifikation aufgrund der Vorbedingungen und der Eingabedaten zu erwarten sind.

Testplan

Der Testplan beschreibt im wesentlichen den organisatorischen Ablauf des Tests. Er kann, muss aber nicht die Testfallspezifikationen beinhalten. Die Inhalte eines Testplans sind in Kapitel 7 näher beschrieben.

Testfallspezifikation

Die Testfallspezifikation das Dokument, welches die Testfälle für ein System beinhält. Eine Testfallspezifikation kann für mehrere Tests verwendbar sein, und jeder Test kann mehrere Testfallspezifikationen umfassen.

Teststrategie

Eine Teststrategie beschreibt die Vorgehensweise für einen möglichst effizienten und ökonomischen Test. Sie wird normalerweise im Testplan festgelegt und umfasst z.B. die Reihenfolge für die Tests, die Testintensitäten je Qualitätsmerkmal oder je Komponente.

Testprotokoll

Im Testprotokoll wird mitgeschrieben, wer wann welche Tests mit welchem Ergebnis durchgeführt hat. Bei Auftreten eines Fehlers wird im Testprotokoll üblicherweise die Fehlernummer festgehalten, welche dem Fehler im Fehlerverfolgungssystem zugewiesen wird.

Testbericht

Testberichte geben einen Überblick über die Ergebnisse eines Tests. Sie werden am Ende eines Tests oder einer Teststufe und/oder in regelmäßigen Intervallen, z.B. wöchentlich, erstellt. Ein Testbericht beinhält Daten wie z.B. Anzahl gefundener/korrigierter/offener Fehler je Fehlerklasse, mögliche Trends etc.

Testautomatisierung

Unter Testautomatisierung versteht man die Verwendung eines Test-Tools (oder Test-Werkzeugs) mit dem Ziel, das Testen zu unterstützen.

6.2 Kurze Geschichte des Testens

Zu Beginn der Software-Entwicklung bestanden Programme aus einfachen Algorithmen, die aus wenigen, oft hartcodierten Eingabedaten ein Ergebnis produzierten. Die „Software“ bestand aus Steckverbindungen oder Lochkarten. Die Ressourcen waren knapp, sodass die Algorithmen oft auf Umwege einschlagen mussten, um Ressourcen zu sparen. Die Ausführung eines Algorithmus dauerte meist Stunden. Die wichtigsten Qualitätsmerkmale waren somit damals die Funktionalität, die Ressourcen-Sparsamkeit und das Zeitverhalten. Dies beeinflusste auch den Test der Programme: aufgrund der langen Ausführungsdauer waren dynamische Tests sehr zeitraubend. Deshalb wurde oft versucht, im vorhinein, also statisch, die Korrektheit des Programms zu beweisen, was bei der überschaubaren Komplexität der Algorithmen zum Teil auch durchaus möglich war. Dies konnte durch formale Beweise geschehen, oder durch Überlegungen zu allen möglichen Eingaben.

Umfassende Tests, d.h. Tests aller möglichen Eingaben an das System, waren dann mit der Entwicklung von Programmiersprachen nicht mehr möglich. Somit mussten Methoden gefunden werden, aus der Menge aller möglichen Testfälle möglichst repräsentative auszuwählen. Bei den eher kleinen, überschaubaren Programmen waren formale, eher auf der Programmlogik basierende Methoden mit hohem Überdeckungsgrad noch durchaus möglich. Mit wachsender Größe und Komplexität wurden formale White-Box-Methoden jedoch weniger brauchbar, die Anstrengungen verschoben sich mehr und mehr darauf, die korrekte Umsetzung von Anforderungen zu zeigen, also auf Black-Box-Methoden. Mittlerweile werden viele Systeme entwickelt, deren Größe nur eine geringe Test-Überdeckung selbst der Anforderungen erlaubt.

Im Laufe der Zeit entwickelten sich unterschiedliche Systemtypen (siehe Kapitel 2), wie Echtzeitsysteme, in Hardware eingebettete Systeme (Firmware), Client/Server-Systeme, Verteilte Systeme, Web-Applikationen und Computerspiele. Jede einzelne dieser Systemtypen stellte unterschiedliche, zum Teil neue Qualitätsanforderungen: Skalierbarkeit, Benutzerfreundlichkeit, Plattformunabhängigkeit, Wartbarkeit, Einhaltung gesetzlicher Bestimmungen, Fehlertoleranz, etc. Manche Anforderungen rückten eher in den Hintergrund, wie z.B. Ressourcen-Sparsamkeit. Um auf dem Gebiet des Qualitätsmanagements die Qualität eines Software-Systems greifbarer zu machen, zerlegten gewisse Standardisierungs-Organisationen den Begriff Qualität in einzelne Komponenten, z.B. IEEE, ISO oder DIN. Dies erleichtert Qualitätsmanagern und Testern heutzutage, die wichtigsten Anforderungen zu identifizieren und zu bewerten.

Eine weitere Entwicklung fand auf dem Gebiet der Testautomatisierung statt: wurde sie zu Beginn rein als automatische Ausführung der Tests gesehen, so fällt heute die Automatisierung jeder Aktivität im Bereich des Testens darunter, z.B. der Testfallspezifikation.

Im selben Zeitraum entwickelte sich auch die Gruppe der Anwender. Am Anfang brachten noch Pioniere ihres Fachbereichs, Wissenschaftler und Ingenieure, noch viel Geduld mit der meist von ihnen selbst erstellten Software auf. Heute ist Software schon in Lebens- und Arbeitsbereiche vorgedrungen, wo der Benutzer wenig Ahnung von den Mühen der Software-Entwicklung hat und überdies hohe Anforderungen stellt.

Seit Beginn der Softwareentwicklung hat, im Zusammenhang mit der wachsenden Größe und Komplexität der Software, somit einerseits eine Verschiebung des Qualitätsfokus und andererseits der Testmethoden stattgefunden:

- Der Qualitätsfokus verschob sich von einigen wenigen, relativ leicht messbaren Qualitätsanforderungen zu einer großen Menge, objektiv nur schwer messbaren Anforderungen.
- Die Testmethoden entwickelten sich von formalen, eher auf der Struktur des Programms basierenden Methoden hin zu eher informellen, auf die Anforderungen bezogenen Methoden.

Diese geschichtliche Verschiebung hat auch in den gängigen Modellen zur Software-Entwicklung ihren Eindruck hinterlassen: bei Tests in frühen Phasen der Entwicklung, wo die einzelnen Systemkomponenten noch überschaubare Größe und Komplexität haben, also bei Modul- und Integrationstests, finden eher formale White-Box-Methoden zum Test der wichtigsten, objektiv messbaren Qualitätskriterien (z.B. Funktionalität, Leistung und Ressourcensparsamkeit) Anwendung; in späteren Tests, also Systemtests, Abnahmetests, etc. werden eher informelle Black-Box-Methoden auch für weniger greifbare Anforderungen wie Benutzerfreundlichkeit angewandt.

6.3 Testen in kleinen Projekten

Dieses Kapitel beschreibt die Test-Methodik in kleinen Projekten, d.h. Projekten mit nur wenigen Komponenten und relevanten Qualitätsmerkmalen und erläutert sie anhand eines Beispiels. Im Ausblick auf die mittleren bis großen Projekte wird analysiert, wo die wesentlichen Unterschiede im Test und deren Ursachen liegen.

6.3.1 Vorgehensweise

Bei kleinen Entwicklungsprojekten sind nur wenige, d.h. max. zwei bis drei Personen beteiligt (vgl. Kapitel 2). Das entwickelte (Teil-)System besteht aus wenigen Komponenten, und nur wenige Qualitätskriterien sind relevant. Die Komplexität ist überschaubar, und somit auch die Aufwände. Wenig Planung bzw. Strategie, sowohl bei der Projekt- als auch der Testplanung, ist notwendig. Die Entwicklung besteht aus einem oder nur wenigen Zyklen.

Aber auch bei kleinen Projekten sind Tests überaus wichtig, um brauchbare Qualität zu erzeugen. Der Fokus liegt auf grundlegender, nicht fortgeschrittener Testmethodik, z.B. bloße Durchführung eines Äquivalenzklassentests durch den Entwickler selbst. Für die meisten Praktiker/Entwickler stellt ein kleines System den „Erstkontakt“ mit Testen dar. Einzelne, in sich abgeschlossene Teilprojekte eines größeren Projekts können auch als kleines Projekt angesehen werden.

Tests werden meist von den Entwicklern selbst durchgeführt und bestehen aus einer oder maximal zwei Stufen: entweder nur Systemtest oder Modultest und Systemtest. Allerdings sind Modultests immer zu empfehlen, da sie viel eher Fehler finden, und diese Fehler sind leichter lokalisierbar.

Im wesentlichen besteht die Vorgehensweise zum Test von kleinen Systemen aus einigen wenigen Schritten (wie übrigens jeder Testprozess, siehe Kapitel 7). Im nächsten Kapitel wird der Prozess anhand eines beispielhaften kleinen Projekts näher erläutert.

Planung der Tests

Dieser Schritt wird bestimmt durch die Fragestellung „Was ist überhaupt zu testen? Was nicht? Wie soll getestet werden?“.

- Welche physisch und logisch eigenständige Komponenten (Programmmodule, Dokumente, etc.) müssen getestet werden?
- Welche Anforderungen sollen beim Test überprüft werden?
- Wie sollen die Anforderungen getestet werden?

Im allgemeinen steht aus der Analyse bzw. Design-Phase bereits ein mehr oder weniger detaillierter Überblick über die Struktur des Systems (Architektur, Entwurf), bestenfalls in graphischer Form, zur Verfügung. Diese kann als Grundlage zur Ermittlung der zu testenden Komponenten herangezogen werden. Wichtig ist, dass „Nebenprodukte“, die nicht explizit oder nur am Rande erwähnt werden, beim Test nicht vergessen werden. Zusätzlich ist zu empfehlen, das System auch einmal selbst zu durchleuchten, da oft, v.a. in schlecht strukturierten Entwicklungsprozessen, die Entwicklung zu diesem Zeitpunkt schon weit fortgeschritten ist, mit Änderungen an der Struktur, die nirgends dokumentiert sind.

Zur Ermittlung der zu testenden Qualitätsmerkmale sollten die Benutzeranforderungen herangezogen werden. Es gilt abzuwägen, welche Qualitätsmerkmale relevant genug sind, einen eigenen Test dafür aufzuziehen. Denn der Test eines jeden Qualitätsmerkmals erfordert seine eigenen Vorbereitungen mit entsprechenden Aufwänden. Auch hier ist wichtig, Anforderungen mit einzubeziehen, welche nicht explizit oder nur am Rande erwähnt wurden. Um keine Qualitätsmerkmale zu vergessen, empfiehlt es sich, Standardzerlegungen von Software-Qualität zu

Rate zu ziehen, z.B. ISO 9126 oder TMap. Weiters ist zu beachten, dass für jede Komponente unterschiedliche Qualitätsmerkmale relevant sein können.

Stehen die oben erwähnten Dokumente nicht zur Verfügung, bleibt dem Tester nichts anderes über, als im Dialog mit Entwicklern, Auftraggeber und künftigen Anwendern die Systemkomponenten und Qualitätsanforderungen selbst zu ermitteln.

Sobald feststeht, welche Komponenten und welche Qualitätsmerkmale zu testen sind, gilt es, konkret festzulegen, wie dies zu geschehen hat. Dazu wird nur je Qualitätsmerkmal und Komponente grob skizziert, welche Methoden wie Anwendung finden und welche Testendekriterien gelten. Testendekriterien dienen der Beantwortung der Frage, wann das Produkt „gut genug“ ist. Sie sind aus dem Grund wichtig, dass selbst kleine Projekte im allgemeinen nicht vollständig getestet werden können, und dass zudem die Korrektur mancher Fehler unökonomisch ist.

Bei der Festlegung der Testmethodik sollte gleich der Aufwand geschätzt werden: die Planung muss so erfolgen, dass bei Eintreten von Problemen noch genügend Freiräume für rechtzeitige (!) Änderungen im Plan bleiben. Eine genauere Spezifikation der Testfälle erfolgt im nächsten Schritt.

Spezifikation der Testfälle

Die Spezifikation der Testfälle sollte im allgemeinen zwar vor Durchführung der Tests erfolgen, bei kleinen Projekten erfolgt sie dennoch oft erst während der Tests. Dabei dürfen sich Tester nur nicht dazu verleiten lassen, vollkommen unmethodisch zu testen. Es sollten daher zumindest die Testmethoden (siehe Kapitel 9 und 10) im vorhinein festgelegt werden; die Ableitung der Testfälle kann dann während des Tests erfolgen.

Kleine Projekte sind im allgemeinen vom Umfang her noch so überschaubar, dass das Gesamtsystem ohne nennenswertes Risiko und ohne explodierende Aufwände im selben Umfang getestet werden kann. Es empfiehlt sich daher, für alle Komponenten eine durchgängige Testmethodik anzuwenden. Dies erleichtert die Planung im Vergleich mit mittleren bis großen Projekten sehr (siehe nächstes Kapitel).

Durchführung

Die Durchführung der Tests erfolgt nach den zuvor geplanten Methoden. Die Tests werden in Protokollen dokumentiert. Die Protokolle können einfache Listen, z.B. in Excel-Format, sein, oder Einträge in einer eigenen Datenbank. Für kleine Projekte sind die einfachen Listen jedenfalls ausreichend. Die aufgetretenen Fehler werden gesammelt, am besten in einer Datenbank, auch bei kleinen Projekten. So wird gewährleistet, dass Fehler nicht verloren gehen, mehrfach korrigiert werden etc.

Zur Durchführung der Tests sollte auch bei kleinen Projekten eine eigene Testumgebung eingerichtet werden. In den Protokollen und vor allem den Fehlermeldungen wird sodann immer vermerkt, wer den Fehler wann, unter welchen Vorbedingungen, mit welchen Aktionen und in welcher Programmversion gefunden hat.

Abschluss

Nach Erreichen der Testendekriterien sind die Tests abgeschlossen. Jetzt gilt es nur noch, die gesammelten Materialien (Protokolle, Datenbanken, Testumgebungen) für eventuelle spätere Verwendung oder Nachvollziehbarkeit, z.B. in späteren Projekten oder aus rechtlichen Gründen, zu sichern und aus dem Test und dem Projekt im allgemeinen für spätere Projekte zu lernen.

Das Lernen aus dem Prozess ist gerade bei kleinen Projekten sinnvoll: das hier Gelernte findet auch in Teilprojekten von größeren Projekten Anwendung. Dazu werden Daten über die Entwicklung des Produkts, z.B. Fehler je Entwicklungszyklus, je Komponente etc. gesammelt und daraus Rückschlüsse auf die Effizienz des Testprozesses geschlossen, was die Grundlage für eine Verbesserung ist (siehe Kapitel 12).

6.3.2 Beispiel: Konversion XML-Daten in HTML

Zur Veranschaulichung des Testvorgehens in kleinen Projekten betrachten wir ein Beispiel. Es soll ein Webseiten-System zum Online-Verkauf von Produkten entwickelt werden. Daten über die Produkte werden als XML zur Verfügung gestellt und sollen in HTML-Format übersichtlich dargestellt werden. Die HTML-Dateien sollen Navigation durch die Produkte und deren Bestellung ermöglichen; für die Bestellung muss die Eingabe von weiteren beschreibenden Daten zu den bestellten Produkten (z.B. Anzahl), die bei der Bestellung mitgesendet werden, möglich sein.

Projekt-Eckdaten

Bei den XML-Daten können ca. zehn verschiedene Typen von unterschiedlicher Komplexität unterschieden werden. Deren Struktur ist in einer Spezifikation beschrieben. Als Referenz für das Layout und die (teilweise) Funktionalität steht ein statischer Prototyp zur Verfügung. Ein Dokument zur Spezifikation der HTML-Funktionalität steht leider nicht zur Verfügung. Die Funktionalität im Detail bleibt den Entwicklern überlassen, es muss nur gewährleistet sein, dass der Geschäftsfall „Verkauf eines Produkts mit in XML beschriebenen Merkmalen via Browser“ abgedeckt ist.

Zur Umsetzung des Projekts wird XSLT verwendet, mit dem XML-Dokumente in beliebige andere Dokumente in XML-Format, u.a. HTML, transformiert werden kann (näheres zu XSLT siehe http://www.w3.org/TR/xsl). Abbildung 6.3 zeigt schematisch die Transformation eines XML-Eingabebaumes in einen HTML-Ausgabebaum. Die Zielplattform ist der Internet Explorer 6.0, es sollen aber Umschaltmechanismen für andere Browser gleich als leere Stubs eingebaut werden. Die Funktionalität der HTML-Seiten wird in Java Script umgesetzt.

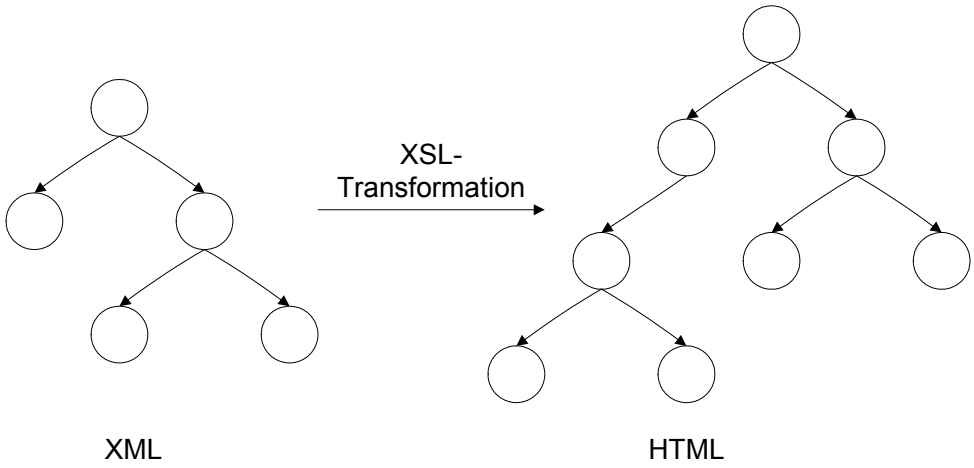

Abbildung 6.3: Transformation XML - HTML über XSLT

Der Aufwand für das gesamte Projekt wird auf ca. 10 Personenmonate geschätzt. Den Auftrag erhält ein Zweier-Team, wodurch sich unter Berücksichtigung ihrer Urlaubsplanung und möglichen dringenden Einsätzen in anderen Projekten eine Durchlaufzeit von ca. 6 Monaten ergibt. Davon ist ein Monat, also zwei Personenmonate Aufwand, für den Test eingeplant.

Entwicklung

Damit das Risiko schwerer konzeptioneller Fehler möglichst gering bleibt, verwendet das Team Methoden des Extreme Programming (XP, siehe Kapitel 12): Programmieren in Paaren (Pair Programming) und kurze Iterationen. Die Entwickler entwickeln jede Komponente gemeinsam und testen sie sogleich intensiv. Somit können sie aus frühen Iterationen für spätere lernen. Aus dem selben Grund wollen sie zusätzliche qualitätssichernde Maßnahmen (z.B. Reviews, siehe Kapitel 5) möglichst früh einbringen. Sie beginnen auch bei den einfacheren Komponenten, um bei den komplexeren Fällen die ersten Hürden schon überwunden zu haben.

Testplanung

Als ersten Schritt für den Test müssen die beiden Teammitglieder nun die Komponenten und Schnittstellen im System identifizieren. Das zu entwickelnde System besteht aus mehreren XSL-Dateien, die im Aufbau im wesentlichen gleich sind. Als Input erhalten sie XML-Daten einer gegebenen Struktur, und der Output besteht aus HTML-Dateien, die in Internet Explorer 6 lauffähig sein sollen. Zusätzlich existieren Java-Dateien (.js), welche in den HTML-Dateien gemeinsam verwendete Funktionalität beinhalten. Somit sind zu testende Komponenten die XSL-Dateien, die gemeinsam verwendeten Java-Dateien sowie die generierten HTML-Dateien.

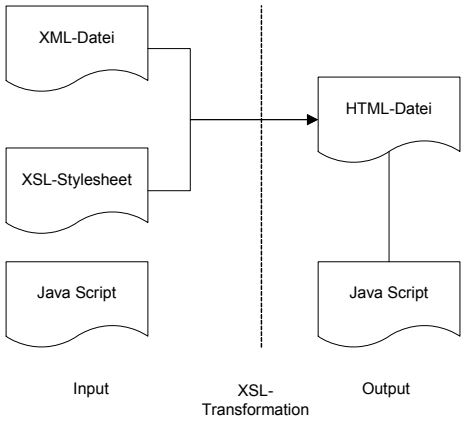

Abbildung 6.4: Komponenten bei der XSL-Transformation XML - HTML über XSLT

Als nächstes gilt es für das Entwicklerteam, die relevanten Qualitätskriterien zu identifizieren. Für jede Komponente können unterschiedliche Kriterien wichtig sein. Sie ergeben sich hauptsächlich aus den Anforderungen der Spezifikation, können aber auch implizit angenommen sein. Deshalb ist es gut, alle relevanten Kriterien für die Testplanung explizit zu nennen. Um keine Qualitätsmerkmale auszulassen, ist es ratsam, Standardzerlegungen der Qualität zu Hilfe zu nehmen. Das Entwicklerteam nimmt den Standard ISO 9126 als Grundlage für ihre Analyse der relevanten Merkmale.

- Bei den XSL-Dateien sind hauptsächlich die Qualitätsmerkmale Funktionalität (korrekte Übersetzung in HTML) und Änderbarkeit (modularer Aufbau, Inline-Dokumentation, Fehlerfindung) wichtig.
- Für die HTML-Dateien, d.h. die generierten Webseiten, sind die Qualitätsmerkmale Funktionalität (Navigation durch Produktstruktur, Bestellung, ...) und Bedienbarkeit (Layout, Vergleichbarkeit mit Prototyp) von Relevanz.
- Die für die Funktionalität der Webseite zuständigen, in den HTML-Dateien eingebundenen .js-Dateien müssen die richtige Funktionalität (Collapse/Expand der Produktstruktur, Auswählen/Kopieren/Löschen eines Produkts, Merken der eingegebenen Daten bei Weiternavigieren, ...), aufweisen und leicht änderbar (modularer Aufbau, Inline-Dokumentation, Fehlerfindung, Wiederverwendbarkeit) sein.
- Andere Qualitätsmerkmale haben in diesem Projekt nur geringe Bedeutung und werden daher gar nicht getestet.

Somit ergeben sich drei unterschiedliche Arten von Tests, nämlich je zu testendem Qualitätskriterium eine: Tests der Funktionalität, der Bedienbarkeit und der Änderbarkeit. Jede dieser Arten von Tests erfordert eigene Vorbereitungen und eine eigene Testmethodik. Das Entwicklerteam entschließt sich zu folgender Vorgehensweise zum Test:

Qualitäts-merkmal	**Komponente**	**Vorgehensweise**

Funktionalität	XSL-Dateien	Die Funktionalität der XSL-Dateien wird nach (vorläufiger) Fertigstellung jeder XSL-Datei getestet. Als Testfälle dienen alle möglichen Abfolgen von Tags in den Eingabe-XML-Dateien.
	HTML-Dateien	Da keine Spezifikation der HTML-Funktionalität im Detail vorliegt, sollen grundlegende Entscheidungen darüber mit dem Auftraggeber gemeinsam getroffen werden. Die Vorgehensweise dafür ist gleich wie zum Test der Bedienbarkeit. Danach erfolgt der Test der Funktionalität der generierten HTML-Dateien im Internet Explorer nach erfolgreichem Test der jeweiligen XSL-Datei. Es sollen alle Funktionen (aus Anwendersicht) mit gültigen und (sofern möglich) ungültigen Eingaben mindestens einmal durchgeführt werden. Außerdem soll jede Codezeile der integrierten Java-Scripts mindestens einmal ausgeführt werden.
	.js-Dateien	Da die Java-Dateien wiederverwendet werden, reicht der Test derer Funktionalität in einer einzelnen HTML-Datei aus. Auch hier gilt, dass jede Funktion mindestens einmal aufgerufen und jede Codezeile zumindest einmal ausgeführt werden soll.
Bedienbarkeit	HTML-Dateien	Zum Test der Bedienbarkeit stellt das Entwicklerteam dem Auftraggeber nach der Fertigstellung jeder XSL-Datei dem Auftraggeber die generierte HTML-Datei zur Anschauung bereit.
Änderbarkeit	XSL-Dateien, .js-Dateien	Die Änderbarkeit lässt sich mit vertretbarem Aufwand nur statisch testen; also entschließt sich das Entwicklerteam, eine Review mit zwei externen, in der Technologie erfahrenen Entwicklern durchzuführen. Die Review soll so früh wie möglich, also nach Fertigstellung der ersten Module, stattfinden, damit für die folgenden Entwicklungen möglichst großer Nutzen daraus gezogen werden kann.

Tabelle 6.1: Testvorgehen je Qualitätsmerkmal und Komponente

Testfälle werden zwar nicht im vorhinein definiert, aber während der Tests in einer Protokolldatei (Excel) mitprotokolliert. Bei Auftreten eines Fehlers wird dieser in der Fehlerdatenbank erfasst, wo er einen typischen Lebenszyklus (siehe Kapitel 7) durchläuft.

Testaufwände

Die Aufwände für die Tests schätzt das Entwicklerteam folgendermaßen:

- Die Vorbereitung des Tests für jede XSL-, HTML und .js-Datei benötigt im Durchschnitt ca. 2 Personenstunden. Danach produziert jeder Testfall ca. 15 Personenminuten Aufwand. Es ergeben sich zum Test der XSL-Dateien für alle XML-Eingabedateien im Durchschnitt ca. 15 Tag-Kombinationen, zum Test der HTML-Dateien ca. 20 Testfälle je Datei, zum Test der .js-Dateien ca. 30 Testfälle je Datei. Bei ca. 10 XML-Eingabedateien und ungefähr 5 .js-Dateien ergibt das (10+10+5)*2h + (10*15 + 10*20 + 5*30)*10min = 25 h + 1500 min = ca. 50 h. Bei Berücksichtigung von Nachtests kann dieser Aufwand verdoppelt werden, also ca. 100 h.

- Der Test der Bedienbarkeit und Entscheidungen über Funktionalitäten erfordern nur die Bereitstellung der HTML-Dateien in lauffähiger Form an den Auftraggeber. Das wird alles in allem (inklusive Nachtests) auf 16 h geschätzt.
- Die Review für die Änderbarkeit erfordert ca. je 3 Stunden je Entwickler und externen Review-Teilnehmern an Vorbereitung, ca. eine Personenstunde in der Durchführung und ca. 1 Stunde Nacharbeit (Dokumentation, Berichte, ...) je Entwickler. Es soll keine Review zur Überprüfung von Korrekturen stattfinden. Somit ergeben sich 3*(2+2) + 1*4 + 1*2 = 18 h.
- Die Protokollierung der Tests erfolgt in einem Excel-Sheet; Fehler werden in einer bereits vorhandenen und fürs Projekt (vom Configuration Management) aufgesetzten Datenbank erfasst. Das Entwicklerteam rechnet mit ca. 16 h Aufwand für Protokollierung der Tests und Erfassen bzw. Bearbeiten von Fehlermeldungen.

Den Gesamtaufwand für die Tests schätzt das Entwicklerteam somit auf 100 + 16 + 18 + 16 = 150 Personenstunden. Bei einem Gesamtbudget für die Tests von 2 Personenmonaten = 2*160 Personenstunden bleibt mehr als genug Puffer für Unvorhergesehenes (Verzögerungen/Schwierigkeiten in der Entwicklung oder beim Test).

Testdurchführung

Bei der Entwicklung stößt das Team immer wieder auf unvorhergesehene Probleme, welche Verzögerungen verursachen, beispielsweise beim Zusammenspiel der HTML-Dateien. Die Reviews werden erst nach Fertigstellung der ersten großen Komponenten durchgeführt und liefern Ergebnisse, die zur Überarbeitung von einigem Code führen. Zudem müssen die Entwickler nach Absprachen den Auftraggebern neue Komponenten einführen, die auch getestet werden müssen. Dadurch verschieben sich die Tests und erhöhen sich die Testaufwände. Allerdings ist der eingeplante Puffer groß genug, dass das Produkt fertig getestet eine Woche vor dem geplanten Endtermin fertiggestellt ist.

Schlussfolgerungen

Zum Beispielprojekt sind einige Punkte hervorzuheben, die in kleinen Projekten zwar vernünftig sind, bei mittleren bis großen Projekten jedoch zu Problemen führen können und verbessert werden sollten:

- Die Entwickler führten die Tests selbst durch.
- Es gab keine Unterscheidung von Teststufen (Komponenten-, Integrations-, Systemtest). Allerdings wurde eine Art Abnahmetest durch den Auftraggeber einzeln für jede XML-Struktur durchgeführt.
- Eine Einteilung des Tests in Phasen (Planung, Vorbereitung, Durchführung, Abschluss) war nur ansatzweise vorhanden.
- Die Entwickler verzichteten auf umfangreiche bzw. detaillierte Planung der Tests. Das ist bei kleinen Projekten ökonomischer, da zu umfangreiche Planung nur zu unnötigem Overhead führt.

- Die Entwickler stellten alle Komponenten in ihrer Bedeutung gleich: sie legten die Reihenfolge der Entwicklung nicht aufgrund der Bedeutung, sondern aufgrund der Größe der Komponenten fest.
- Die Entwickler legten die Testendekriterien nicht explizit fest. Sie waren nur implizit gegeben als: „Der Test ist zu Ende wenn alle Tests durchgeführt werden können, ohne dass neue Fehler auftreten, oder wenn der Auftraggeber das Produkt abnimmt, oder wenn das Projekt vorzeitig abgebrochen wird.“ Eine solche Vorgehensweise kann in größeren Projekten zu Komplikationen führen, da dort unter Umständen in jedem Test-/Korrektur-Zyklus mit neuen Fehlern gerechnet werden muss, und zudem die Korrektur mancher Fehler zu teuer ist. In einem solchen Fall wäre der Test nie zu Ende. Somit ist es besser, explizite und vernünftige Kriterien festzulegen.

Das Beispiel kann auch als Teilprojekt eines größeren Projekts gesehen werden (siehe nächstes Kapitel). Die beschriebenen Tests können dann nur als „Private Tests“ gesehen werden, d.h. Tests, welche der Entwickler selbst durchführt, bevor er das Produkt für den Komponententest freigibt. Der Abnahmetest durch einen Dritten entspricht dann dem Komponententest. Darauf folgen dann Integrations- und Systemtests im Verbund mit anderen Komponenten sowie ein Abnahmetest für das Gesamtsystem.

6.4 Testen in mittleren bis großen Projekten

Im Vergleich zu kleinen Projekten befassen sich mittlere bis große Projekte in den meisten Fällen mit der Entwicklung eines Systems mit vielen Komponenten und relevanten Qualitätsmerkmalen. Der Test solcher Systeme ist daher weniger leicht überschaubar und aufwandsmäßig abschätzbar. Dieser Abschnitt beschreibt die Vorgehensweise für Tests in mittleren bis großen Projekten und arbeitet anhand eines konkreten Beispiels die Unterschiede zur Vorgehensweise bei kleinen Projekten heraus.

6.4.1 Vorgehensweise

Mittlere bis große Projekte (siehe Kapitel 2), bei denen das zu entwickelnde Produkt aus vielen Komponenten besteht, sind zu umfangreich und komplex für einfache Ansätze. Die Komplexität des Gesamtsystems erlaubt es nicht, die zu erwartenden Aufwände von Anfang an genau festzulegen. Zum einen sind mehrere die Entwicklung begleitende Teststufen notwendig, um Fehler so früh wie möglich aufzufinden: Modultests, Integrationstests, Systemtests und eventuelle Abnahmetests. Zum anderen besteht jede Teststufe aus Testzyklen, und erst nach mehreren Testzyklen lassen sich die Aufwände genauer abschätzen.

Genauso wie bei kleinen Projekten besteht der Test bei mittleren bis großen Projekten aus mehreren Schritten. Im kleinen Rahmen, also auf Modul- oder Teilprojektebene, gestaltet sich der Test ähnlich wie bei kleinen Projekten. Auf der übergeordneten Ebene, also für die Integrations-, System- und Abnahmetests ist eine sorgfältige, iterative Planung notwendig, um eine kontinuierliche Kontrolle über den Testprozess zu bewahren. Bei sturer Anwendung von Testmethoden ergeben sich zu viele Testfälle für „brute force“. Daher liegt hier der Fokus auf Strategie, Planung und Ökonomie.

Planung der Tests

Bei mittleren bis großen Projekten ist vor allem die Planung auf höherer Ebene wichtig. Die Fragestellung bleibt zwar dieselbe wie bei kleinen Projekten, es kommen aber noch zusätzliche Punkte hinzu:

- Welche Teststrategie soll verwendet werden, um die wichtigsten Fehler möglichst früh und kostengünstig aufzuspüren?
- Wie soll der Test gestaltet werden, um größere Abweichungen vom Plan rechtzeitig erkennen und effizient reagieren zu können?

Ein wesentlicher Unterschied zu den kleinen Projekten liegt in der Integration der vielen Komponenten. Die größere Komponentenzahl macht eine strengere Unterteilung in Teststufen notwendig. Die Integration des Gesamtsystems kann nicht mehr auf einmal (Big-Bang) geschehen, sondern muss in schrittweise (Top-Down oder Bottom-Up) erfolgen. Zur Integration dürfen nur fertig komponentengetestete Systemteile gelangen. Integrationstests werden in Kapitel 7 näher erläutert. Sobald das komplette System erfolgreich integriert ist, kann der Systemtest erfolgen. Eine Beschreibung der Abfolge der Teststufen, eine genaue Definition der Integrationsschritte und der dafür verwendeten Hilfsmittel (v.a. Stubs und Driver, siehe Kapitel 7 und 11) ist ein wesentlicher Inhalt des Testplans. Die weiteren Planungsschritte sind je Teststufe durchzuführen.

Genauso wie bei kleinen Projekten werden auch hier die Komponenten und die relevanten Qualitätskriterien ermittelt. Nun aber kann nicht mehr so einfach festgelegt werden, auf welche Art und Weise die Anforderungen getestet werden sollen. Es gilt vielmehr, eine ökonomische Teststrategie zu erarbeiten, wenn möglich mit Rückfall-Varianten für eventuell eintretende Risiken.

Hierzu müssen unter Berücksichtigung von Abhängigkeiten zwischen Komponenten diese relativ zueinander gewichtet werden, genauso die Qualitätskriterien. In Abhängigkeit von der jeweiligen Relevanz werden dann die Testmethoden entsprechend dem zur Verfügung stehenden Budget so gewählt, dass die wichtigsten Komponenten und Kriterien am intensivsten getestet werden. Die Tests werden in iterativen Test-/Korrekturzyklen geplant, wobei nach Abschluss jedes Zyklus die Planung mit neuen Kennzahlen angepasst wird. Die iterative Testplanung inklusive Aufwandsschätzungen ist in Kapitel 8 genauer beschrieben.

Spezifikation der Testfälle

Die Spezifikation der Testfälle erfolgt sodann nach den jeweiligen den Komponenten zugeordneten Methoden. Bei mittleren bis großen Projekten ist es wichtig, dass dies vor der Testdurchführung zu geschieht, damit möglicherweise in der Planung zu niedrig oder zu hoch geschätzte Aufwände schon während der Spezifikation erkannt werden können.

Als Faustregel für die Spezifikation der Testfälle gilt, dass sie so allgemein wie möglich und so detailliert wie nötig gehalten werden sollten. Im Testfall soll also nur das festgelegt sein, was die Eindeutigkeit des Testfalls ausmacht, alles andere soll vom Tester frei wählbar sein. Mit solchen Testfällen tun sich zwar unerfahrene Tester am Anfang schwer, aber erfahrene Tester haben es viel leichter, und mit der Zeit wird – hoffentlich – aus jedem unerfahrenen Tester ein erfahrener.

Üblicherweise sind Testfälle in Dokumentform beschrieben. Im Idealfall werden Testfälle in etwas formalerer Weise in einer Datenbank festgehalten, die dann während der Durchführung auch zur Protokollierung der Tests verwendet werden kann.

Durchführung

Vor der eigentlichen Durchführung der Tests muss eine geeignete Testumgebung eingerichtet werden. Dies gestaltet sich vor allem bei der Integration schwierig, da viele Komponenten, die bis zu diesem Zeitpunkt unter Umständen auf komplett unterschiedlichen Plattformen entwickelt wurden, nun zu einem System zusammengefügt werden sollen.

Es empfiehlt sich, einen Vorbereitungstest durchzuführen, der feststellen soll, ob das (Teil-)System überhaupt schon reif genug für einen vollständigen Test ist. Dazu kann eine Untermenge der tatsächlichen Testfälle herangezogen werden.

Die Protokollierung der Tests kann in Listenform oder über eine Datenbank erfolgen, die Fehlererfassung sollte in jedem Fall über eine zentrale Datenbank mit Zugriff über Clients geschehen.

Da die Tests iterativ ablaufen und die Testplanung dementsprechend ständig aktualisiert werden muss, müssen die für die Planung relevanten Kennzahlen des Tests (z.B. Testaufwand je Testfall, siehe Kapitel 8) während der Durchführung erfasst werden.

Abschluss

Der Abschluss der Tests nach Erreichen der Testendekriterien umfasst genauso wie bei den kleinen Projekten die Sicherung von Materialien und die Auswertung von gesammelten Testkennzahlen zum Zweck der Prozessverbesserung in späteren Projekten.

6.4.2 Beispiel: Produktverkauf via Internet

Zur Veranschaulichung des Testvorgangs in mittleren bis großen Projekten betrachten wir wieder ein Beispiel-Projekt. Dabei soll ein bestehendes Produkt zur Verwaltung und zum Verkauf von Produkten erweitert werden, sodass die Produkte über das Internet angeboten und verkauft werden können.

Projekt-Eckdaten

Das bestehende Produkt besteht aus einer Datenbank, einem Backend für die Geschäftslogik und einer graphischen Oberfläche. In der Datenbank sind sowohl Metadaten über die Produktstruktur als auch Daten über tatsächlich erfolgte Verkäufe (Verträge) enthalten. Bei Abwicklung eines Verkaufs werden die Metadaten herangezogen, um Verträge mit konkreten Ausprägungen zu erzeugen.

Das GUI soll im Zuge der Entwicklung durch ein Internet-fähiges Front-End ersetzt werden. Änderungen an den Metadaten sollen „auf Knopfdruck“ für den Endbenutzer im Internet wirksam werden, ohne dass dafür an den Webseiten gebastelt werden muss. Zusätzlich soll das System mindestens 100 gleichzeitige Kauf-Anfragen verkraften.

Entwicklung

Die Entwicklung wird von mehreren eigenständigen Teams durchgeführt. Jedes Team entwickelt eine oder mehrere Komponenten (siehe Abbildung 1.4□1). Jedes Team ist für die eigene(n) Komponente(n), vor allem deren Qualität, verantwortlich. Daher muss zwischen den Teams viel Kommunikation (Telefonate, persönliche Gespräche) stattfinden, um die Schnittstellen aufeinander abzustimmen. Diese sind zwar in der Spezifikation enthalten, welche das Gesamtsystem beschreibt, doch während der Entwicklung gibt es immer wieder notwendige Änderungen.

Es gibt einen technischen und einen organisatorischen Projektleiter für das Gesamtprojekt. Der technische Projektleiter übernimmt die Rolle des Testmanagers und entscheidet bei technischen Problemen. Der organisatorische Projektleiter entscheidet bei unerwarteten zusätzlichen Aufwänden, Terminverzögerungen etc.

Insgesamt sind an der Entwicklung 15 Personen beteiligt, plus zwei Personen vom projektübergreifenden CM, die nicht zu 100% ins Projekt eingebunden sind. Der Gesamtaufwand für das Projekt wird auf ca. 100 Personenmonate geschätzt, mit einer Durchlaufzeit von ca. 8 Monaten. Für die Entwicklung der Komponenten inklusive Modultests werden 5 Monate veranschlagt, danach werden drei Viertel der Leute schon für andere Projekte eingeplant. Der Integrationstest und Systemtest soll mit ca. 4 beteiligten Personen 3 Monate in Anspruch nehmen.

Testplanung

Als erstes identifiziert der Testmanager wiederum die Komponenten und Schnittstellen im System. Grob können die Komponenten aus Abbildung 1.4□1 herangezogen werden, von denen die meisten neu zu entwickeln sind:

- das Verbindungspooling am Unix-Server der Applikationslogik, welches einmal aufgebaute Verbindungen wiederverwendet; geschrieben in C
- eine Dialogkomponente am Web-Server, welche die Kommunikation mit dem Pooling übernimmt; geschrieben in Visual C++
- ein Kommunikations-Arbiter, der Anfragen entweder an die Datenbank oder via die Dialogkomponente an das Backend-Programm weiterleitet; geschrieben in Visual Basic
- ein XML-Übersetzer, der Daten aus der Datenbank in XML umwandelt und XML-Daten in eine vom Arbiter verwendbare Form; geschrieben in Visual Basic
- ein Web-Server; dazu werden Active Server Pages des Microsoft Internet Information Servers verwendet
- ein Generator, der Daten über die Produktstruktur aus der Datenbank ausliest und die Generierung von XML-Dateien anstößt; geschrieben in Visual Basic
- ein Transformator, der die Transformation von XML-Dateien über XSL-Stylesheets in HTML-Dateien durchführt; geschrieben in Visual Basic

- XSL-Dateien, welche Anweisungen für die korrekte Transformation beinhalten; entwickelt mit MSXML bzw. eingebettetem Java-Script; diese Komponente kennen wir schon vom Beispiel für kleine Projekte
- An der Datenbank, der Applikationslogik und dem Verbindungauf-/abbau sind nur geringfügige Änderungen durchzuführen.

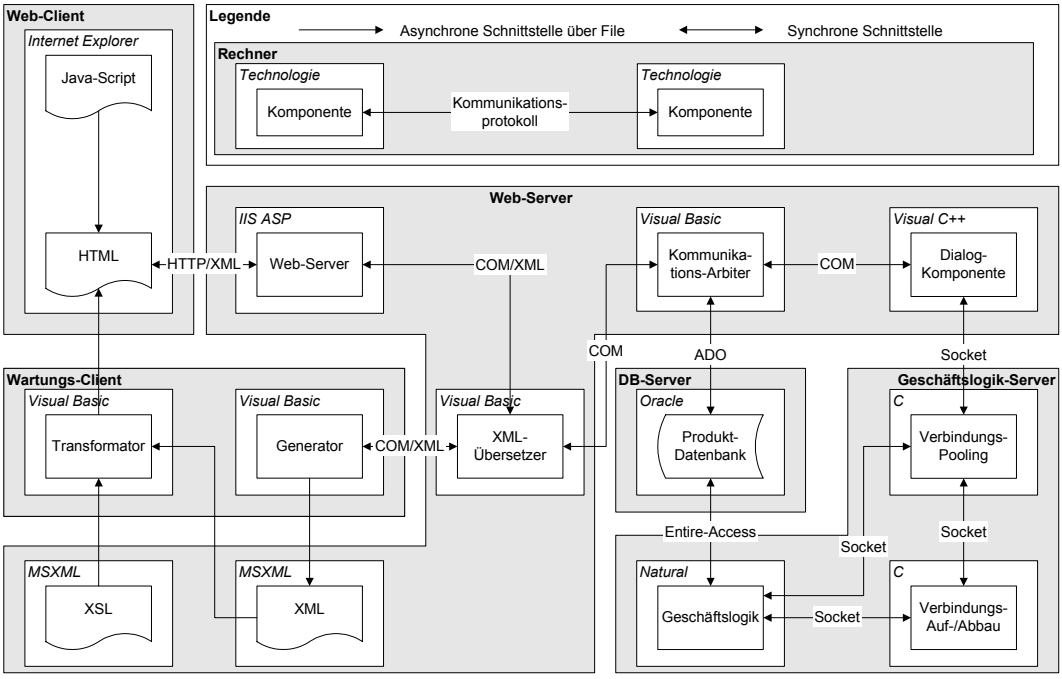

Abbildung 6.5: Struktur des Produktverkaufs via Internet

In diesem Projekt sind mehrere Teststufen zu planen: der Modultest, der Integrationstest und der Systemtest. Ein Abnahmetest wird nicht durchgeführt, da es sich um die Entwicklung eines Standardprodukts ohne konkreten Kunden handelt. Jede dieser Teststufen soll in Test-Korrektur-Zyklen solange stattfinden, bis keine gravierenden Fehler (Fehler der Priorität 1 im verwendeten Fehlerverwaltungssystem) mehr im System enthalten sind.

Die Modultests werden von den Entwicklerteams jeweils selbst geplant, durchgeführt und dokumentiert. Für die Durchführung der Modultests gibt der Testmanager den Teams nur grob vor, wie diese dokumentiert werden sollen und welche Infrastruktur verwendet wird.

Danach wird eine stufenweise Integration der einzelnen Komponenten durchgeführt. Hierfür müssen die Entwicklerteams Stubs und Treiber erstellen, welche ein Zusammenspiel von bestimmten, vom Testmanager festgelegten Komponenten ermöglichen sollen. In frühen Schritten ersetzen Treiber und Stubs Systemteile, die noch nicht integriert wurden. Die Integrationsstufen werden vom Testmanager in Absprache mit den Entwicklerteams festgelegt.

Der Systemtest betrifft sodann das Zusammenspiel aller synchron arbeitender Komponenten. Die Einbindung der asynchron arbeitenden Komponenten sollte bereits im Integrationstest getestet worden sein.

Der Testmanager unterscheidet drei grundlegende Anforderungen für das Gesamtsystem aus Anwendersicht (Internetclient):

- Funktionalität: das System hat nur einen Geschäftsfall, nämlich Kauf eines Produkts, mit all seinen Variationen; alle anderen Funktionalitäten unterliegen der Verantwortung der einzelnen Teams
- Leistungsfähigkeit: das System muss 100 gleichzeitige Kaufanfragen bearbeiten können
- Benutzerfreundlichkeit: der Benutzer muss sich in der Produktstruktur leicht zurechtfinden und in wenigen Schritten einen Einkauf tätigen können

Zum Test dieser Anforderungen des Gesamtsystems legt der Testmanager folgende Vorgehensweise fest:

- Der Test der Funktionalität des Gesamtsystems erfolgt rein über den Internetclient. Die durchzuführenden Testfälle können vom Modultest der HTML-Seiten übernommen werden, mit dem einzigen Unterschied, dass jetzt das gesamte System mitarbeitet.
- Der Test der Leistungsfähigkeit besteht aus drei Teilen: dem Test der Antwortzeiten im Idealfall, dem Test der Skalierbarkeit und dem Test des Verhaltens bei plötzlicher Höchstlast.
    a. Ersterer Test wird über das HTML-Frontend durchgeführt, wobei die Antwortzeiten mittels Code-Instrumentierung gemessen werden.
    b. Der Test der Skalierung geschieht über eigene Treiberprogramme (Typ A und B), welche anstelle des HTML-Frontends eingesetzt werden. Das Treiberprogramm vom Typ A startet eine aus einem Pool von Kaufanfragen zufällig ausgewählte Anfrage, wartet auf eine Antwort vom Server, protokolliert diese und startet eine neue Kaufanfrage. Das Treiberprogramm vom Typ B startet in regelmäßigen Abständen Programme vom Typ A, womit eine steigende Last simuliert wird (nach Messungen wird angenommen, dass das Netzwerk keinen Engpass darstellt und daher die Anfragen von einem einzigen Rechner gestellt werden können). Aus dem Anstieg der Antwortzeiten bei steigender Last wird dann ersichtlich, ob das System eine Last von 100 gleichzeitigen Anfragen verkraftet, bis zu welcher Last das System unter einer bestimmten Antwortzeit bleibt und bis zu welcher Last das System korrekte Ergebnisse liefert oder überhaupt terminiert.
    c. Zum Test des Verhaltens bei plötzlicher Höchstlast startet ein Programm 100 Anfragen gleichzeitig und protokolliert das Antwortverhalten des Systems je Anfrage. Hier wird wiederum angenommen, dass das Netzwerk keinen Engpass darstellt, und dass die sequentiellen Anfragen von einem einzigen Rechner für den Test „gleichzeitig genug“ sind.
- Die Benutzerfreundlichkeit wird in einem Workshop mit Personen überprüft, die bisher wenig bis gar keinen Kontakt mit dem Produkt hatten, und auch wenig Ahnung im Umgang mit ähnlichen Produkten haben, z.B. Mitarbeitern aus der Marketing-Abteilung.

Testaufwände

Die Testaufwände schätzt der Testmanager folgendermaßen auf ca. 27 Personenmonate:

- Jeder Entwickler benötigt ca. einen Monat zum Komponenten-Test seiner Komponente(n). Dies mach 15 Personenmonate aus.
- Für die Durchführung der Integrationstests samt Korrekturzyklen werden ca. vier Tester und Entwickler zwei Monate lang beschäftigt sein, das macht 8 Personenmonate.
- Für den Systemtest samt Korrekturzyklen brauchen die vier Tester und Entwickler einen Monat, was 4 Personenmonate ausmacht.

Testdurchführung

Die Dokumentation der Tests erfolgt über eine Datenbank, die von Mitarbeitern des projektübergreifenden Configuration Managements eigens für das Projekt eingerichtet wird. Darin werden sowohl die Testfälle samt Ergebnissen festgehalten als auch die gefundenen Fehler.

Die Entwicklung der einzelnen Module und deren Komponententests verläuft ohne Probleme. Die ersten größeren Probleme ergeben sich bei der Integration: die Schnittstellen mancher Komponenten passen nicht zusammen. Die Einrichtung der Software-Infrastruktur (Installation der Einzelkomponenten und der dafür benötigten Softwarekomponenten) gestaltet sich als schwieriger und aufwändiger als angenommen, vor allem durch die nicht oder schlecht durchdachte Vorgehensweise bei der erneuten Einrichtung des Gesamtsystems nach jedem Testzyklus.

Ein weiteres Problem ergibt sich aus dem Abzug der meisten Entwickler: manche der im Integrationstest aufgefundenen Fehler sind zu aufwändig für die verbliebenen Entwickler, manche Fehler können sie gar nicht beheben, weil sie bei der Entwicklung der betroffenen Komponenten nicht beteiligt waren. Somit bleibt nichts anderes über, als einige der bereits abgezogenen Entwickler zu reaktivieren. Diese sind aber bereits in andere Projekte eingebunden, haben damit ihre eigenen Termine einzuhalten und sind somit wenig daran interessiert, an einem für sie bereits gelaufenen Projekt noch zusätzliche Arbeit durchzuführen.

Der Integrationstest verzögert sich durch die auftretenden Probleme um einen ganzen Monat, und ist auch um einiges aufwändiger als geplant, nämlich um ca. 6 Personenmonate.

Im Systemtest zeigt sich dann, dass zwar einerseits die Funktionalität und die Benutzerfreundlichkeit einwandfrei sind, dass allerdings auf dem Gebiet der Leistung wesentliche Mängel vorliegen. Ab einer Belastung von ca. 12 gleichzeitigen Anfragen terminiert das System nicht mehr, was weit unter der Anforderung für die Leistung liegt. Eine plötzliche Belastung führt schon bei 10 gleichzeitigen Anfragen zu falschen Ergebnissen und abgestürzten Prozessen. Auch die Antwortzeit im Idealfall liegt deutlich über den gewünschten Werten.

Die folgenden Tätigkeiten zur Analyse und Korrektur der Fehler laufen mit deutlichen Zeichen der Panik ab, sind unkoordiniert und ineffizient. Es ergibt sich zwar zum Glück, dass die mangelnde Leistung nur auf einzelne Komponenten zurückzuführen sind und nicht auf Fehler in der Architektur, aber dennoch verzögert sich der Abschluss des Projekts dadurch um weitere drei (!) Monate mit wesentlichen Mehraufwänden, nämlich zusätzlichen ca. 15 Personenmonaten.

Das Projekt wird somit mit 4 Monaten Verspätung, mit 21 Personenmonaten Mehraufwand und mit deutlicher Erleichterung seitens der Projektleitung abgeschlossen. Hätte nämlich die mangelnde Leistung an einer schlecht durchdachten Architektur gelegen, wäre nichts anderes übriggeblieben, als diese neu zu konzipieren, was einem Neustart des Projekts gleichgekommen wäre. So liegt das Projekt mit ca. 50 % Verzögerung und ca. 20 % Mehraufwand sogar eher im Durchschnitt der „erfolgreichen" Software-Projekte, d.h. derjenigen, die nach Abschluss mit einem brauchbaren Produkt aufwarten können.

Schlussfolgerungen

Der Test dieses Beispiel-Projekts war zwar zum Teil sehr gut durchdacht, wies aber andererseits so manche schwerwiegende Mängel auf. Wir können mehrere Schlussfolgerungen ziehen:

- Der Test war einerseits erfolgreich, da er Mängel noch vor der Produktion aufzeigte. Andererseits kam dies zu einem viel zu späten Zeitpunkt. Der Testmanager identifizierte zwar korrekt die wesentlichen Anforderungen an das Gesamtsystem, vergaß aber darauf, diese so früh wie möglich zu überprüfen, d.h. noch während der Komponententests oder spätestens während der Integrationstests. So wurden die relevanten Qualitätsmerkmale erst im Systemtest getestet, was viel zu spät für rechtzeitige und kostengünstige Korrekturen ist.
- Die Entwickler wurden zu früh abgezogen. Offensichtlich unterlag der Projektleiter der falschen Annahme, dass die Tests nur noch dem Nachweis der korrekten Funktion des Programms dienen, und nicht der Fehlerfindung.
- Die Tests fanden ohne Berücksichtigung der relativen Wichtigkeit von Qualitätsmerkmalen oder Komponenten statt. Bei geeigneter Priorisierung der Tests (und auch der Entwicklung) wären die wesentlichen Mängel viel früher in Erscheinung getreten.
- Die ersten Probleme im Projekt traten plötzlich auf: es war zwar jedem bewusst, dass die Integration kritisch ist, aber niemand dachte, dass es so schlimm wird. Somit verfiel mit einem Mal die Planung und es traten Panikreaktionen auf, was auf mangelndes Risikomanagement und zu wenig Spielräume im Testplan (und Projektplan) zurückzuführen ist. Ein Projekt dieser Komplexität erfordert auch eine flexiblere, nämlich iterative (Test-)Planung (siehe Kapitel 8 und auch die agilen Ansätze in Kapitel 12) und ein vernünftiges Risikomanagement (siehe dazu Kapitel 14).

Das Beispiel zeigte, dass bei mittleren bis großen Projekten die Komplexität nicht aus dem größeren Aufwand resultiert, sondern aus dem Zusammenspiel der Komponenten. Bis zum Abschluss der Komponenten-Tests lief alles gut. Erst bei wachsender Anzahl an interagierenden Komponenten zeigten sich die Probleme, allerdings schon in ausgewachsener Form. Ein guter Test berücksichtigt also die Beziehungen zwischen Komponenten und Qualitätsmerkmalen sowie deren relative Bedeutung und zielt darauf ab, Fehler so früh wie möglich zu finden, solange sie noch klein und leicht lokalisierbar sind. Außerdem wird er so geplant, dass er dennoch auftretende gröbere Probleme verkraften kann.

6.5 Zusammenfassung

6.6 Literaturreferenzen

[Beizer, 1984] Boris Beizer: "System Testing and Quality Assurance", van Nostrand, 1984, ISBN 0-442-21306-9

[Boehm, 1982] Barry W. Boehm, "Software Engineering Economics", Prentice Hall PTR, 1982, ISBN 0-138221-22-7

[DeMarco, 1982] DeMarco, Tom: “Controlling Software Projects”, Yourdon Press, 1982

[DeMarco, 1998] DeMarco, Tom: “Der Termin: Ein Roman über Projektmanagement”, Carl Hanser Verlag, 1998, ISBN 3-446-19432-0

[Kaner et al., 1999] Cem Kaner, Jack Falk, Hung Quoc Nguyen: “*Testing Computer Software*", Wiley, 1999, ISBN 0-471-35846-0

[Kit, 1995] Edward Kit: "*Software Testing in the Real World*", Addison-Wesley, 1995, ISBN 0-201-87756-2

[Myers, 1979] Glenford J. Myers: "*The Art of Software Testing*", Wiley, 1979, ISBN 0-471-04328-1

[Pol et al., 2000] Martin Pol, Tim Koomen, Andreas Spillner: “Management und Optimierung des Testprozesses: ein praktischer Leitfaden für Testen von Software, mit TPI und TMap”, dpunkt.verlag, 2000, ISBN 3-932588-65-

7

6.7 Übungen und Fragen