Das Fehlermodell: Weiterführende Ansätze für Aufwandsschätzung und Planung von Tests

Dietmar Kropfitsch

Uwe Vigenschow

Im Objektspektrum 6/2003 haben wir eine Methode vorgestellt, wie man mit Hilfe des empirischen Ansatzes eines Fehlermodells sowohl die Aufwände für alle Arten von Tests inkl. der Fehlerkorrekturaufwände in den Griff bekommen und auch Gewissheit über die aktuelle Produktqualität gewinnen kann. In diesem Beitrag wollen wir Ihnen Anpassungen und Erweiterungen dieses Ansatzes vorstellen.

Konkret behandeln wir zwei Themen:

- Anpassungen des Basis-Fehlermodells an unterschiedliche Projektrealitäten. Dies betrifft im wesentlichen Anpassungen an inkrementell-iterative Vorgehen.
- Die Einsatzmöglichkeit von Fehlermodellen bei der nachträglichen Einführung automatisierter Entwicklertests in ein Softwareprojekt. Hierbei stehen Effizienzbetrachtungen im Vordergrund. Dazu betrachten wir die zentralen Klassenarten der Entity-Controll-Boundary Design-Leitlinie unter dem Licht on Unit-Tests.

Zuerst möchten wir kurz das Prinzip unseres Fehlermodells zusammenfassen.

Zusammenfassung unseres Fehlermodells

In [Kro03] haben wir unser Fehlermodell für Black-Box-Tests detailliert vorgestellt. Auf der Basis von abzuschätzenden Eingangsparametern können Aussagen zu Aufwänden für die Testfallerstellung, Testdurchführung und Fehlerkorrektur gemacht sowie eine Abschätzung der zu erwartenden Fehleranzahl getroffen werden. Das Rechenmodell basiert auf wenigen einfachen Formeln und kann daher mit einem Tabellenkalkulationsprogramm erstellt werden.

Aus einer zu erwartenden Fehlerdichte und einer Abschätzung der Programmgröße wird die Gesamtfehleranzahl hoch gerechnet. Sie umfasst alle Fehler über den gesamten Entwicklungsprozess, d.h. ein Großteil dieser Fehler wird bereits in der Analyse und Entwicklung gefunden und korrigiert. Je nach Qualität des Softwareentwicklungsprozesses enthält das Testobjekt vor dem Start der Integrations- und Systemtests noch ein Drittel der Gesamtfehler.

Da die verbleibende Fehleranzahl alle Arten von Fehlern umfasst, kann die von der QS zu findende Fehleranzahl über die Testabdeckung häufig noch weiter reduziert werden. Über diese Zielgröße für die Fehleranzahl kann auch ein Testende-Kriterium abgeleitet werden.

Über Annahmen, wie viele Testfälle zum Aufdecken einer Fehlerwirkung notwendig sind und Abschätzungen der Aufwände der einzelnen Tätigkeiten wird der QS-Testaufwand hoch gerechnet. Dabei werden über eine geometrische Reihe auch die Aufwände für Re-Tests korrigierter Fehler sowie fehlerhaft oder unvollständig korrigierter Fehler berücksichtigt.

Modifikationen des Basis-Fehlermodells

Dieses Basis-Fehlermodell kann in verschiedene Richtungen erweitert und ausgebaut werden. So sind einerseits individuelle Anpassungen wie auch grundsätzlich methodische Erweiterungen möglich.

Betrachtung der Fehlerklassen

Eine ökonomisch durchaus interessante und einfache Variante besteht in der Betrachtung einer Fehlerkurve, in der Fehlerklassen Berücksichtigung finden. In diesem Fall kann man das Test-Ende-Kriterium selektiv auf die fatalen und schweren Fehler anwenden. Selbstverständlich sollte auch in diesem Fall die Mindestanzahl an Fehlern gefunden sein, der zeitliche Verlauf der Kurve für die Anzahl der schweren Fehler abflachen und trotz neuer Testfälle auf einem konstanten Wert verharren. Wie Fehlerklassen sinnvoll gebildet werden, sehen Sie im Kasten 1.

Differenziertere, risikomanagement-orientierte Ansätze

Nun mag der Leser einwenden, dass Anwendungen aus Teilsystemen bestehen können, die sowohl unterschiedliche Komplexität als auch unterschiedliches Risiko aufweisen. Für diesen Fall bieten sich drei Alternativen an:

- den Parameter Fehler/KLOC für jedes Teilsystem zu variieren (Komplexität) oder
- den Parameter Testabdeckung für jedes Teilsystem je nach Einschätzung des Risikos zu erhöhen oder verringern (Risikofaktor) oder
- die beiden Parameter Fehler/KLOC und Testabdeckung für jedes Teilsystem separat festzulegen.

Voraussetzung für alle drei Ansätze ist eine eindeutige Abgrenzbarkeit und

© 2005 Dietmar Kropfitsch und Uwe Vigenschow

damit Abschätzbarkeit der Größe jedes Teilsystems.

Diese differenzierten Betrachtungen erhöhen den organisatorischen Aufwand für die Erhebung der Größe der Anwendung, der Zuordnung der Fehler zu den Teilsystemen, Auswertung, Interpretation, usw. Es sollte daher sehr sorgfältig geprüft werden, wie differenziert die Anwendung wirklich betrachtet werden soll oder muss. Meist wird es ausreichen, die Testabdeckung zu variieren. Selbst bei gleich bleibendem Testaufwand kann schon mit dieser Maßnahme das Risiko deutlich minimiert und damit eine wesentlich höhere Qualität der Anwendung erreicht werden.

Anpassung an iterative Verfahren für System- und Integrationstests

Die bisherigen Betrachtungen beruhen auf wasserfallähnlichen Vorgehen wie dem V-Modell. Für ein iteratives Vorgehen müssen ein paar Anpassungen erfolgen

Für die erste Iteration wird das Basis-Fehlermodell ohne Anpassungen eingesetzt. Beachten Sie dabei, nur die tatsächlich in der ersten Iteration zu implementierenden KLOC als Basis Ihrer Berechnungen zu berücksichtigen.

Für alle weiteren Iterationen sind die jeweils zu implementierenden KLOC mit einem Regressionsfaktor von 5-10% zu multiplizieren. Von diesem Wert ausgehend sind alle weiteren Berechnungen entsprechend dem Basis-Fehlermodell durchzuführen.

Durch diesen Regressionsfaktor werden Fehlerpotentiale berücksichtigt, die durch die Implementierung von Schnittstellen zu bereits vorhandenen Funktionen und/oder die Erweiterung von bereits verfügbaren Funktionen entstehen können. Je umfangreicher und/oder komplexer diese Erweiterungen sind, desto höher sollte der Regressionsfaktor gewählt werden.

Ein Fehlermodell für Entwicklertests

Ein Aufwandsschätzmodell im Rahmen eines testgetriebenen Designs analog zum Black-Box-Fehlermodell macht nicht viel Sinn. Unsere Erfahrungen bestätigen die Aussagen bekannter Autoren wie z. B. [Beck00], dass die Erstellung automatisierter Entwicklertests z.B. mit JUnit [JUnit] nicht zu signifikantem Mehraufwand führt. Im Gegenteil werden durch ein testgetriebenes Vorgehen im vergleichbaren Zeitraum neben den Nutzklassen zusätzlich deren Testklassen erstellt und die verbleibende Fehleranzahl spürbar reduziert. Aus Managementsicht ist hier also kein zusätzliches Aufwandsschätzmodell erforderlich. Man kann sich mit einer detaillierten Aufgaben-Zeiterfassung begnügen.

Anders stellt sich die Situation dar, wenn es darum geht, in einem Wartungs- und Weiterentwicklungsprojekt nachträglich automatisierter Unit-Tests und testgetriebenes Vorgehen einzuführen. Welche Aufgaben erwarten uns dabei und wie lange werden sie wohl dauern? Wo fangen wir an und was lassen wir vorerst beiseite liegen? Antworten liefert uns eine Modifikation des Fehlermodells.

Nachträgliche Einführung von Unit-Tests

Wir haben also Software, die bereits beim Kunden im Einsatz ist, und wollen die weitere Wartung und Erweiterung langfristig effizient durchführen. Eine breite Basis automatisierter Entwicklertests soll Ihnen dabei helfen, doch leider wurden bisher keine solchen Tests erstellt. Um mitten drin damit anzufangen, bietet sich ein zweistufiges Verfahren an:

1. Für jede Softwareänderung oder –erweiterung wird ab sofort testgetrieben gearbeitet, d.h. zuerst werden Unittests programmiert und danach die jeweilige Funktionalität.
2. Die bestehende Software soll gezielt und effizient durch zusätzliche Unittests sicherer und stabiler gemacht werden.

Der erste Punkt steht hier nicht weiter zur Debatte. Wie aber gehe ich beim zweiten Punkt vor?

Fehlerverteilung

Fehler sind nicht gleichmäßig verteilt, sondern ballen sich an bestimmten Stellen, die sich dadurch auszeichnen, dass sie besonders komplex oder kompliziert sind. Wir können diese Stellen auf zwei Wegen identifizieren, über unsere Versionskontrolle oder über ein Fehlermodell auf Basis von Klassenarten. Der erste Weg ist recht einfach und kann auch parallel als Ergänzung zum Fehlermodell gegangen werden. Wir filtern dazu aus unserer Versionskontrolle die Quellen mit den höchsten Versionsnummern heraus. Diese sind vermutlich am häufigsten geändert worden und dies kann ein Indiz für hohe Komplexität und Fehleranfälligkeit sein.

Fehler ballen sich aber auch in ganz bestimmten Arten von Klassen, wie sie in einer Design-Leitlinie definiert werden. In einer Design-Leitlinie werden Klassen gemäß ihren unterschiedlichen Aufgaben in verschiedene Arten von Klassen unterschieden (s.u.). Die Ausgangsbasis für unser Fehlermodell bilden sowohl die Anzahl der Klassen als auch die Anzahl ihrer Methoden. Das klingt einfach, wir müssen hier aber gemäß einer Design-Leitlinie differenzieren, denn Klasse ist nicht gleich Klasse. Jede Klasse hat unterschiedliche Aufgaben und daran angepasst unterschiedliche Ausprägungen. Am einfachsten lässt sich dieser Sachverhalt an Hand eben dieser Design-Leitlinien darstellen. Es gibt verschiedene dieser abstrakten Design-*Schubladen*. Zwei weit verbreitete sind:

- Entity-Controller-Boundary
- Werkzeug-Aspekt-Material.

Design-Leitlinie

Glücklicherweise sind die Differenzen für unsere Betrachtungen marginal. Am Beispiel *Entity-Controller-Boundary* wollen wir die Konsequenzen für das Fehlermodell erläutern.

- Entitäten sind Fachklassen oder werden häufig auch als Geschäftsobjekte bezeichnet. Sie repräsentieren unsere Materialien, also häufig reale Objekte. Hier finden wir unsere persistenten Klassen, die also auch die Datensicht beinhalten. Typische Beispiele sind *Kunde*, *Vertrag*, *Rechnung* usw. Entitäten zeichnen sich dadurch aus, dass sie viele Attribute haben und damit auch viele einfache Zugriffsmethoden, die sog. Getter und Setter. Wirklich komplexe Methoden finden wir deutlich seltener. Hierbei handelt es

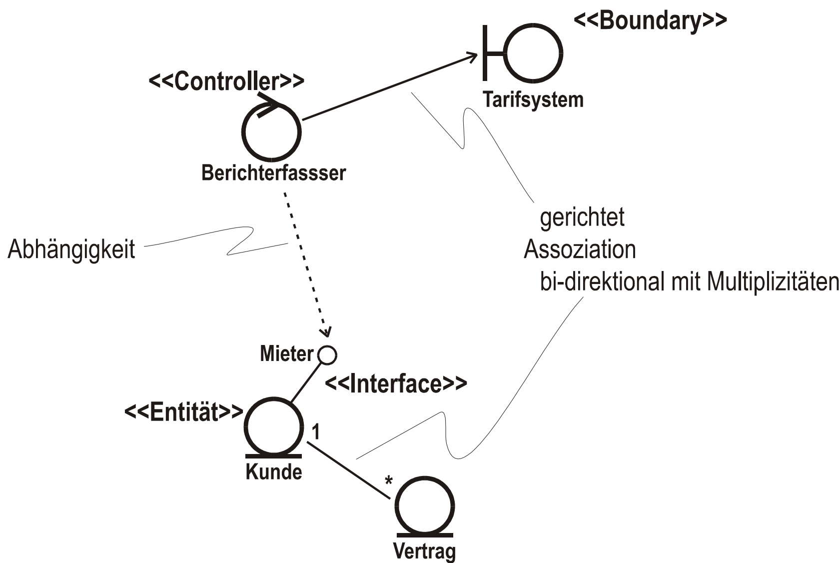


Abb. 1: Schematische Einführung in die Symbolik der UML für Entitäten, Controller, Boundary, Interface, Assoziation und Abhängigkeit.

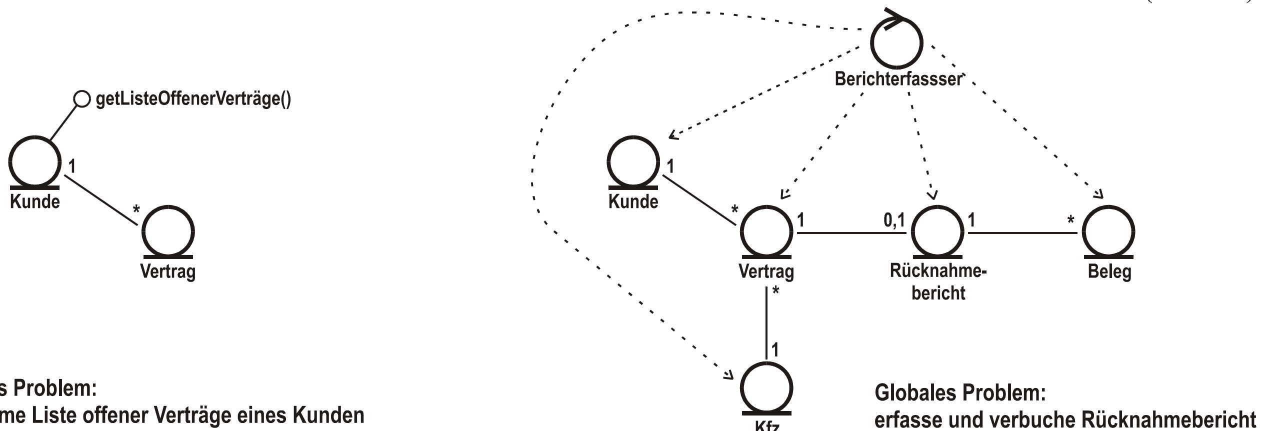


Abb. 2: Das Zusammenspiel von Entitäten und Controllern hängt davon ab, ob es sich um eher lokale (links) oder globale (rechts) Problemstellungen handelt. Für lokale Fragestellungen braucht kein Controller eingeführt zu werden, wenn eine Entität die Verantwortlichkeit mit ihrem eigenen Wissen (Attribute) erfüllen kann. Bei globalen Problemen wird dies kaum möglich sein. Die Verantwortlichkeit wird einem eigenen Controller übertragen.

sich meist um lokale Such- oder Berechnungsmethoden (Abb. 1 und 2).

■ Controller steuern unsere Abläufe. Meist wird pro Anwendungsfall ein Controller implementiert. Daneben gibt es noch zentrale Basis- und Hilfsabläufe, die ebenfalls je in einen eigenen Subcontroller implementiert werden. Controller haben kaum eigene Attribute, da sie sich die notwendigen Daten im Zusammenspiel mit den Entitäten von dort holen bzw. dort manipulieren. Auch finden wir eher wenige Methoden, die haben es aber in sich. Hier sind die Abläufe und Regeln implementiert (Abb. 1 und 2).

■ Boundarys sind Sichten (Views), die an Systemschnittstellen geschaffen werden, um das dahinter liegende System zu kapseln. Das typische Beispiel sind Bildschirmmasken an der Mensch-Maschine-Schnittstelle. Dort wird kaum Geschäftslogik implementiert, sondern über zusammenfassende Methoden mit eher „flachen" Daten auf das dahinter liegende System zugegriffen. Boundaries haben meist keine Attribute und fassen in ihren wenigen Methoden des gekapselten Systems zusammen. Meist werden sie als Singleton implementiert, es gibt also im System nur eine Instanz (Abb. 1).

Daneben gibt es natürlich noch weitere Klassenarten wie Interfaces[1] oder Factories[2], die für unsere Betrachtungen relevant sind. Für den ersten Einstieg in die Fehlermodelle fassen wir diese jedoch der Einfachheit halber unter *Sonstiges* zusammen.

Klassenarten und Testaufwand

Entitäten, Controller und Boundaries gehen unterschiedlich in unser Fehlermodell ein. Entitäten sind zwar relativ große Klassen mit vielen Attributen und noch mehr Methoden. Die meisten sind jedoch simple Zugriffsmethoden, die Getter und Setter. Diese sollten am Besten aus einem Designtool heraus generiert werden. Somit muss der Generator einmal getestet werden, die Getter und Setter fallen uns nur dann zur Testlast, wenn sie modifiziert werden müssen. Deutliche Testaufwände über diverse Testmethoden (Testfälle) fallen nur für die wenigen, komplexen Methoden an, mit denen ein lokaler Ablauf gesteuert wird (Abb. 2). Diese sind dann genau so gewissenhaft zu testen, wie die Controller.

Obwohl die Controller nur wenige Attribute und wenige Methoden enthalten, fallen dort die hohen Testaufwände an. In diesen Methoden sind die Abläufe, Berechnungen und Regeln sowie Ausnahmen implementiert. Im Gegensatz zu den Entitäten wird ein Vielfaches an Testmethoden benötigt.

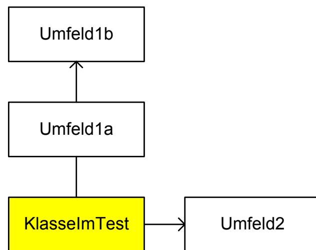
Boundaries sind unsere Systemschnittstellen. Obwohl Boundaryklassen recht

[1] Abstrakte, funktionale Schnittstellenbeschreibungen.

[2] Fabrikklassen zum Erzeugen von Objekten einer oder ähnlicher Klassen.

Klassentest

Kettentest



Modultest

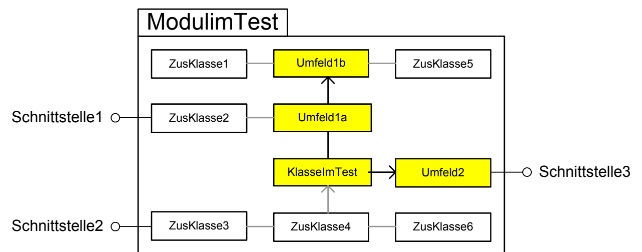


Abb. 3: Die verschiedenen Arten von Unittests. Der eigenständige Klassentest (links), der Test entlang von Objektketten im direkten Umfeld (mitte) und der Test einer kohärenten Klassengruppe über die Modulinterfaces (rechts).

kompakt und meist gut überschaubar sind, entstehen im Vergleich zu den wesentlich umfangreicheren Entitäten recht große Testaufwände, da eine Systemschnittstelle vollständig mit alle ihren Regeln und Ausnahmebedingungen zu testen ist. Die Größe einer Klasse sagt also nicht direkt etwas über ihren Testaufwand aus. Die obigen Betrachtungen werden um so wichtiger, je knapper unsere Zeit wird, da wir anhand der Klassenarten gezielter auf die Fehlersuche gehen können.

Arten von Unittests

Was sind eigentlich Unittests? Diesen Sammelbegriff haben wir jetzt schon mehrfach verwendet, aber was meinen wir damit? Genauer sollten wir eigentlich von Klassen-, Ketten- und Modultests sprechen. Was ist nun damit gemeint? Ein Klassentest betrifft nur die zu testende Klasse unter Nutzung der Methoden, die die Klasse nach außen bereit stellt. Betrachten wir danach auch das direkte Umfeld, so können wir z.B. einen vollständigen Ablauf entlang einer Assoziationskette testen. Eine Gruppe zusammen arbeitender, kohärenter Klassen wird gerne als Modul zusammen gefasst. Die Zugriffe auf ein solches Modul erfolgt häufig über Interfaces. Jetzt kann ein vollständiger Ablauf innerhalb des Moduls über seine Schnittstellen getestet werden[3]. Die Zusammenhänge sind in Abb. 3 zusammen gefasst. Alle Arten von Unittests lassen sich über Tools wie JUnit gut automatisieren.

Die dreistufige Einteilung ist deshalb sinnvoll, weil auf jeder Stufe andere Fehler gefunden werden. Die Reihenfolge vom Klassentest über den Kettentest zum Modultest stellt sicher, dass die jeweiligen Grundlagen in Ordnung sind und wir uns auf die bisher noch nicht entdeckbaren Fehler in lokalen bzw. globalen Zusammenspiel der Klassen konzentrieren können.

Das Unit-Test-Fehlermodell

Um zu einer Planungstabelle für die Klassen-, Ketten- und Modultests zu kommen, schätzen wir die unterschiedlichen Klassenarten bzgl. ihres Anteils zu testender Methoden ab. Daraus ergibt sich die Anzahl der für die Testklassen zu programmierenden Testmethoden.

Zusätzlich werden Testklassen für Ketten- und Modultests benötigt. Hier programmieren wir die Testfälle, die das Zusammenspiel über die einzelnen Klassentests hinaus prüfen.

[3] Genau genommen handelt es sich beim Modultest um einen Grey-Box-Test.

Für unsere Unit-Tests brauchen wir entsprechende Stellvertreterobjekte, also mehr oder weniger komplexe Dummy-Klassen, die uns das notwendige Umfeld simulieren. Die notwendigen Aufwände können in der Tabelle abgeschätzt werden (s. Abb. 4).

Was können wir dem Rechenmodell auf den ersten Blick entnehmen? Die nachträglichen Aufwände für einen Klassen-, Ketten oder Modultest differieren erheblich. Ein Modultest, der sich nur auf die Boundaries konzentriert, ist danach bei einer ersten Testabdeckung von 30% innerhalb weniger Tage realisierbar. Selbst die deutlich aufwändigeren Kettentests sind bei gleicher Testabdeckung innerhalb weniger Wochen machbar. Für beide Testarten werden keine zusätzlichen Dummys benötigt.

Bei der nachträglichen Implementierung automatisierter Tests schlagen wir daher vor, mit Modultests zu beginnen, ggf. deren Testabdeckung weiter zu erhöhen, und dann Kettentests zu programmieren. Der Aufwand für breit angelegte Klassentests ist nachträglich enorm hoch.

Details zum Rechenmodell

Die Klassenarten sind nur nach Entity, Controller und Boundary differenziert. Andere, wie Fabriken usw. sind unter *Sonstige* zusammen gefasst. Aus diesen Klassen wird die durchschnittliche

© 2005 Dietmar Kropfitsch und Uwe Vigenschow

Anzahl konkreter Methoden ermittelt bzw. approximiert. Für Controller und Boundaries sind alle Methoden zu testen. Für Entitäten und andere beschränken wir uns auf die komplexeren Methoden, die wir mit 10% bzw. 30% angenommen haben. Daraus ergibt sich eine Anzahl in der Spalte *Anzahl zu testender Methoden*.

Zu jeder Methode einer zu testenden Klasse gibt es mindestens zwei Testmethoden innerhalb seiner Unittestklasse: je eine für die positiven und negativen Tests. Bei komplexeren Klassen wie Controllern sind es häufig mehr. Die mittlere Anzahl je Klassenart wird in der Spalte *Testmethoden pro zu testende Methode* eingetragen. Daraus ergeben sich durch Multiplikation mit der geplanten Testabdeckung (im Beispiel 30%) die Anzahl der zu programmierenden Testmethoden.

Für den unabhängigen Klassentest sind je nach Klassenart mehr oder weniger viele Dummys zu erstellen. Bei nachträglich implementierten Testfällen wird die Anzahl der Dummys meist gering ausfallen. Auch diese Werte werden geschätzt und in das Rechenmodell übertragen. Für die Erstellung der Testmethoden, Dummyklassen und die Korrekturen fehlerhafter Tests werden die mittleren Aufwände in Minuten geschätzt und in die entsprechenden Felder eingetragen.

Fehlermodell Berechnungsblatt Unit-Tests

Parameter

Klassenart	Anzahl Klassen	konkrete Methoden pro Klasse im Mittel	direkt zu testende Methoden	Anzahl zu testender Methoden	Testmethoden pro zu testender Methode	geplante Testabdeckung	Anzahl der Testmethoden	je Klasse zu entwickelnde Dummys	Aufwand je Test* [min]	Aufwand je Testfall-Korrektur [min]	Anteil fehlerhafter Tests	Aufwand Dummy-Erstell. [min]	1 PT hat Stunden
Enität	300	30	10%	900	2	30%	540	1	30	15	20%	30	8
Controller	100	7	100%	700	3		630	2					
Boundary	10	3	100%	30	3		27	1					
Sonstige	100	10	30%	300	2		180	1					
Summen				1930			1377						

Berechnungen

Unittest-Art	Anzahl zu testender Methoden	Anzahl der Testmethoden	Anzahl der Dummys	Testaufwand [PT]	Aufwand Fehlerfixes [PT]	Aufwand Dummy-Erst. [PT]	Aufwand je Testart [PT]	Faktor aus fehlerhaften Testfällen
Klassentest	1930	1377	183	86	11	11	108	1,25
Kettentest	700	630		39	5		44	
Modultest	30	27		2	1		3	
Summen		2034		127	17	11	155	

PT: *Personentag*

* *Aufwand je Testfall für Testfall-Findung, Implementierung, Durchführung und Auswertung*

Abb. 4: Beispiel eines Fehlermodells für Unit-Tests in Form eines einfachen Berechnungsblatts einer Tabellenkalkulation. Die Aufwände nachträglich zu erstellender Unit-Tests kann darüber abgeschätzt werden.

Die Aufwandsberechnungen im unteren Teil des Rechenmodells aus Abb. 4 differenzieren folgendermaßen zwischen den Testarten:

- Klassentest: Einzeltests der zu testenden Methoden aller Klassen mit den dazu notwendigen Dummys.
- Kettentest: Tests über die Controllerklassen mit den dazu benötigten Entitäten bzw. weiteren Klassen. Hierfür werden meist keine Dummys benötigt.
- Modultest: Test über die Boundaries der Module. Auch hierfür werden kaum Dummys benötigt werden.

Aus der jeweiligen Anzahl zu erstellender Testmethoden kann deren Aufwand abgeschätzt werden. Für die Klassentests kommt der Aufwand für die Dummyprogrammierung hinzu.

Die einzelnen Testarten werden dabei als unabhängig betrachtet, da es hier um eine nachträgliche Implementierung der Tests geht. Bei einem rein testgetriebene Design können die Klassentestfälle in den Kettentests wieder verwendet werden.

Ansatz für nachträglich entwickelte Unit-Tests

Nach den bisherigen Erkenntnissen bietet sich folgendes Vorgehen bei der nachträglichen Einführung automatisierter Unit-Tests an.

1. Eine geeignete Unit-Test-Umgebung wird allen Entwicklern bereit gestellt.

2. Für alle neu entwickelten bzw. veränderten Methoden bestehender Klassen werden nach testgetriebenem Vorgehen zuerst automatisierte Tests geschrieben, bevor der Nutzcode erstellt bzw. modifiziert wird. Mehraufwände für die Testautomatisierung sind dabei höchstens beim erstmaligen Einsatz als Eingewöhnung anzunehmen. Danach sollten sich die Aufwände durch die schnellere und mit geringeren Fehlern erstellte Nutzcode-Erstellung ungefähr aufheben.

3. Die fehleranfälligsten bzw. problematischsten Bereiche in der bestehenden Software werden identifiziert. Einen Teil dieser Stellen kennen die Entwickler bereits aus den bisherigen Erfahrungen mit dem Code. Weitere Indizien können aus der Versionskontrolle entnommen werden. Dies sind Quellen mit hohen Versionsnummern und generell Quellen, die erst in jüngster Vergangenheit hinzu gekommen sind. Meist sind dies Klassen, die entweder Controller oder aber eine starke Controller-Funktionalität haben.

4. Zusätzlich werden die Boundaries bzw. die Klasssen mit Boundary-Funktionalität identifiziert.

5. Für die Boundaries werden Modultests entwickelt und automatisiert.

6. Für die Controller werden Kettentests entwickelt und automatisiert.

7. Für die als kritisch identifizierten Klassen, die noch keinen direkten Unit-Test haben, werden entsprechende Klassentests und die dafür notwendigen Dummys entwickelt.

Die Aufwände für die nachträglich zu erstellenden Unit-Tests können über das Unit-Test-Fehlermodell nach den individuellen Anpassungen abgeschätzt werden.

Unit-Tests und Refactoring

Stehen umfangreiche Umstrukturierungen im Code an (Refactoring, [Fowl00]), so werden sehr detaillierte Klassentests als Voraussetzung benötigt. Damit wird sicher gestellt, dass nach einer Designänderung die Funktionalität unverändert weiter gegeben ist.

Vor einem Refactoring sollten daher fehlende Unit-Tests ergänzt werden. Die dafür notwendigen Aufwände können mit einer lokalen Betrachtung des Unit-Test-Fehlermodells unter Beschränkung auf umzustrukturierende Module abgeschätzt werden. Die Umstrukturierungsmaßnahme kann so besser geplant werden.

Fazit

Das in [Kro03] vorgestellte Fehlermodell kann für verschiedene Umgebungen und Aufgaben angepasst werden. Diese Modelle leisten ihren Beitrag zur Aufwandsabschätzung und damit zur Planung von QS-Maßnahmen einerseits und der nachträglichen Testautomatisierung in der Entwicklung andererseits.

QS-Aufwände, die gerne unterschätzt werden, können so realistischer bewertet werden. Eine nachträgliche Testautomatisierung in der Entwicklung kann in die verschiedenen Unit-Test-Ebenen zerlegt werden. So können die Aufwände für Klassen, Ketten- und Modultests individuell ermittelt werden. Je nach Aufgabenstellung besteht die Möglichkeit, Einsatzschwerpunkte zu setzen, wie wir an den Beispielen zur nachträglichen Testautomatisierung in sieben Schritten bzw. am Refactoring dargelegt haben.

Bibliografie

[Beck00] Kent Beck: Extreme Programming. Addison-Wesley 2000

[Fowl00] Martin Fowler: Refactoring. Addison-Wesley 2000

[Junit] www.junit.org

[Kro03] Dietmar Kropfitsch, Uwe Vigenschow: *Das Fehlermodell: Aufwandsschätzung und Planung von Tests*. OBJEKTspektrum Nr. 6 2003, Nr. 6

Die Autoren

Dietmar Kropfitsch ist als selbständiger Berater für Qualitätsmanagement, Konfigurationsmanagement, Projektmanagement und Software-Test mit über 15 Jahren Berufserfahrung in leitenden Positionen internationaler technischen und kaufmännischer Großprojekten für Finanzdienstleister und die Raumfahrtindustrie tätig.

Uwe Vigenschow ist Berater und Trainer bei der oose.de GmbH in Hamburg. Er blickt auf über 15 Jahre Berufserfahrung in der technischen und kaufmännischen Softwareentwicklung zurück, wobei er sich seit 1991 mit Objektorientierung und seit 1997 mit Qualitätsmanagement befasst.

© 2005 Dietmar Kropfitsch und Uwe Vigenschow

Fehlerklassen

Wie teilen wir die gefundenen Fehler in sinnvolle Klassen ein? Wie viele Klassen sind notwendig? Aus unseren Erfahrungen reichen dazu vier Klassen aus.

Klasse 1 - Fataler Fehler: Der Fehler legt einen Bereich oder eine Komponente lahm und macht so die Benutzung des Produkts (und auch das Testen selber) unmöglich. Klasse 1 - Fehler führen dazu, dass der System- oder Integrationstest nicht bestanden wird, sie sind so schnell wie möglich zu beheben, idealer weise innerhalb von 24 h.
Beispiele: eine Komponente kann nicht gestartet werden; der Geschäftsprozess kann nicht instanziert werden; Datenbank-Inhalte werden zerstört oder können nicht weiter verwendet werden.

Klasse 2 - Schwerer Fehler ohne Work-around: Die Benutzung des Produkts ist erschwert. Es kann aber mit schweren Beeinträchtigungen benutzt (und getestet) werden. Das Problem kann über einen Work-around nicht umgangen werden. Klasse 2 - Fehler sollten, falls von der QS angefordert und begründet (Weitertesten nur sehr eingeschränkt möglich), innerhalb von 2 Tagen behoben werden.
Beispiele: der Workflow kann nicht weiter geleitet werden; Personen bzw. deren Berechtigungen können nicht gelöscht werden; das Benutzerkennwort kann nicht geändert werden; Ergebnisse einer Berechnung sind falsch, weitere Testergebnisse können nicht mehr auf Richtigkeit überprüft werden; Daten werden nicht gespeichert (je nach Umfang kann dies auch ein Klasse 1 - Fehler sein!).

Klasse 3 - Schwerer Fehler mit Work-around: das Fehlerbild entspricht dem von Klasse 2, jedoch kann der Fehler über einen Work-around umgangen werden.
Beispiele: falsche, irreführende oder unvollständige GUI; sehr schwere Fehler in der Dokumentation, die zu erheblichen negativen Folgen führen können; Personen können nicht angelegt werden, außer es wurde vorher noch keine Rolle angelegt; Datenfelder, die in einer bestimmten Reihenfolge eingegeben werden müssen, um keine falschen Berechnungen zu erzeugen usw.

Klasse 4 - Fehler: Beeinträchtigung des Produkts, die aber nicht sonderlich erschwerend sind, sondern ärgerlich oder irritierend.
Beispiele: unschöner Aufbau der GUI, unpassend definierte Default-Werte, Tippfehler, fehlerhafte oder unergonomische Tabbing-Order, Short Cuts fehlen, funktionieren nicht oder sind doppelt belegt usw.

© 2005 Dietmar Kropfitsch und Uwe Vigenschow