Methoden und Verfahren zur Durchführung von Funktionstests bei Objektorientierter Software

Th. Grechenig, W. Zuser, Ch. Brem

Research Industrial Software Engineering (RISE)
Technische Universität Wien, Karlsplatz 13, 1040 Wien
grechenig@viveka.tuwien.ac.at, zuser@swt.tuwien.ac.at

Abstract: Softwareentwicklung auf der Basis der Objektorientiertung (OO) hat in fast allen Bereichen der Softwaresysteme wesentliche Bedeutung erlangt. Die Tätigkeit des System-Funktionstests wird dadurch naturgemäß tangiert, wiewohl eine detaillierte Überprüfung der Fragestellung klarlegt, daß die überwiegende Menge an bisherigen "Best-Practices" für das Testen beibehalten werden sollte und vor allem ein spezieller Teil des White-Box-Tests des Codes besondere Obacht benötigt.
Sicherheitsrelevante Systeme, die mit Hilfe von OO-Technologie entwickelt werden, erfahren einen angemessenen Funktionstest indem a) das Testen nachhaltig als projekt- und phasenübergreifende Tätigkeit verstanden wird b) diese strategische Sicht im Laufe der Wartung und des configuration management beibehalten wird c) die realweltlichen und physikalischen Risiko- und Streß-Testfälle des konkreten Systems wie bisher vornehmlich auf der Basis der jeweiligen sicherheitstechnischen Historie identifiziert werden d) bei jenen Softwaresystemteilen, die im Geiste der OO entworfen und codiert wurden, das Testverfahren auf folgende Problemzonen speziell achtet:

- Die Grundbaustruktur besteht aus einer Vielzahl autonomer Objektklassenfamilien mit den Vor- und Nachteilen einer elaborierten Vererbungsstruktur. Diese OO-Struktur muß gut sein. Das Design muß intensiv geprüft werden.
- Spezielle Testfälle für Vererbung und Polymorphismus sind vorzusehen.
- Klassentests wenden neben Black-/White-Box-Tests Zustandsmatrizen an.
- Je nach konkreter Sprache und Entwicklungsumgebung sind für das automatische Testen spezielle Vorkehrungen zu treffen.

1. Einleitung

Es ist die Natur von Softwaresystemen in Realdimensionen, daß sie aus prinzipiell theoretischen Gründen sowie aus ökonomischen Überlegungen nicht "vollständig" geprüft werden können. Die kombinatorische Explosion der Möglichkeiten findet sich auf allen Systemebenen wieder: ein mittelgroßes Softwaresystem mag von 5 Entwicklern gebaut worden sein, die jeder im Durchschnitt 20 eigene Code-Files verfaßt haben. Unabhängig von Sprache und Methode ergibt sich daraus schon eine Anzahl von 5000 Testpaaren für einfache Konsistenzprüfungen zwischen den Programm-Files. Ein einzelnes dieser 100 Files mag eine typische Prozedur aus 80 Zeilen C-Code enthalten mit vielleicht 10 Entscheidungen in Form von binären Verzweigungen und Schleifen in

Abfolge: theoretisch ist die Anzahl der Pfade multiplikativ: schon allein aus statischer Sicht ergeben sich hier schnell 1000 Pfade. Daß jede einfache Integer-Tabelle mit 10 Einträgen schon rund 10^{50} Zustände annehmen kann, spielt da beinahe schon eine geringe Rolle: in der Regel läßt sich diese Art von Variantenreichtum durch die Bildung von Äquivalenzklassen nach Maßgabe der Programmsemantik brauchbar handhaben. Aber selbst wenn für jede Varibale nur drei repräsentative Bereiche übrigbleiben: 20 Variablen ergibt $3^{20} > 10^9$... ...und trotzdem funktioniert Software. Sie funktioniert, weil erfahrene Ingenieure funktionierende Systeme bauen können. Wider alle theoretische Wahrscheinlichkeit. Neben fundierter Konstruktion ist der Test dabei das Handlungsmittel des Ingenieurs: kritische Bereiche werden besonders intensiv getestet und das Risiko wird derart eingeschränkt.

Das Testen objektorientierter (OO) Software unterscheidet sich nicht grundsätzlich vom Testen von Software irgendeiner anderen Bauart. Testen in seiner besten industriellen Form ist eine gesamtprojektbegleitende Aktivität, die in jeder Phase der Systemverfeinerung Testfälle sammelt und in jeder analogen Phase der Integration und des Zusammenbaus diese Testfälle anwendet bzw. ausprobiert. Betrachtet man jedoch den realexistierenden Zustand mancher Softwareprojekte, dann macht es sehr oft wenig Sinn über die Besonderheiten des Testens in OO-Systemen zu reflektieren: das kleine Test-1x1 ingenieurgemäßer Softwareentwicklung hat noch gar nicht Einzug gehalten. Dann hält man es mit dem Softwaretesten wie mit den Elektromotoren von der guten alten Fertigungsstraße: der Motor kommt vom Band auf den Teststand und wenn er gut klingt, dann wird er ausgeliefert. Dort funktioniert das auch ganz gut, doch Softwaresysteme sind Maschinen anderer Bauweise und haben keinen Klang...

2. Der Funktionstest als permanente Entwicklungstätigkeit

Im Jargon der Softwaretechnik nennt man den klassischen industriellen Funktionstest *Verifikation*. Die Verifikation prüft die Frage "Wurde das System wie geplant richtig gefertigt?". Im Gegensatz dazu prüft die *Validierung* "Wurde das richtige System gebaut"? Fragen der Validierung sind im wesentlichen vor und nach der Prüfung der hier gegenständlichen Problemstellungen abzuhandeln. Validierung gehört in den Bereich der allgemeinen Projektziele, in das Requirement Engineering bzw. zum Akzeptanztest und zur Zielprüfung. Diese Fragen sind alle ziemlich unabhängig von der konkreten technischen Fertigung. OO ist eine Modellierungs-, Fertigungs- und Implementierungstechnik.

Traditionell kommt die Phase des Testens immer nach den Phasen Analyse, Design und Implementierung. Die in der Literatur häufig genannte "Horrorzahl" von bis zu 50% Anteil der Testphase am Gesamtentwicklungsaufwand (schon 30% erscheint dem Hausverstand sehr hoch) ergibt sich letztlich aus eigentlichen Konstruktionsfehlern, die erst beim Zusammenbauen der Teilkomponenten bemerkt und evident werden. Rekonstruktion in späten Projektphasen ist eben aufwendig und 50% Aufwand in der Phase des Testens steht i.A. für 5% reiner Testarbeit und 45% Rekonstruktion.

Prinzipiell läßt sich dieses Problem bei der Softwareentwicklung wohl nicht völlig vermeiden. Es kann aber deutlich verbessert werden, indem

- Testen von Beginn an als eine projektpermanente Aktivität angesehen wird
- Testfälle so früh wie möglich hergestellt und angewendet werden. Testpläne entstehen schon in den entsprechender Entwicklungsphasen (siehe Abb.1)
- Komponententests, Teilintegrationstest und Tests in Risikobereichen früh durchgeführt werden
- Fragen der Validierung gezielt zu geeigneten Zeitpunkten auch während der Entwicklung geprüft werden.

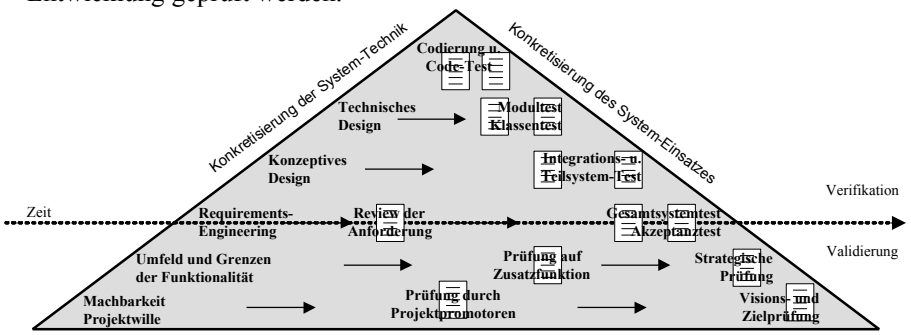

Abb. 1: „A-Symmetrie“- Modell des Software-Testens: Phasen und zugehörige End-Prüfungen liegen auf einer Ebene. Testpläne werden je nach Bedarf für Zwischenprüfungen verwendet.

Folgende allgemeine Testtypen spielen aus Ingenieursicht bei Software-Systemen eine wesentliche Rolle:

Normaler Funktionstest: Das System wird über längere Zeit anhand der geplanten Funktionen getestet. Es werden nur „normale“ Fehler geprüft, die durch die Eingaben und Aktionen von typischen Usern auftreten können. Funktionstests können aber auch auf Komponenten oder Klassenebene durchgeführt werden, indem mit geeigneten Testtreibern Funktionen gestartet werden (siehe 3.1).
Volllasttest bzw. Streßtest: Das System wird an seine Grenzen geführt. Alle Parameter werden an die Leistungsobergrenze gestellt, alle Arten von Zusätzen werden angeschlossen, viele User arbeiten gleichzeitig. Streßtests zeigen erfahrungsgemäß oft versteckte Fehler auf.
Überlast-Test: Dabei wird das System bewußt jenseits der geplanten Volllast getestet. Man erwartet dabei nicht, daß es korrekt funktioniert, sondern beobachtet, wie sich das System verhält. Diese Verhalten gibt über Risiken außerhalb der Spezifikation Auskunft.
Negativ-Test: Ähnlich wie beim Überlasttest wird das System in unlogischer und unkorrekter Art und Weise ausprobiert.
Akzeptanz-Test, Usability-Tests: Bei einigen aktuellen Systementwicklungen ist korrekte Funktionalität und Stabilität des Systems allein nicht mehr ausreichend. Handhabbarkeit, Komfort und Zufriedenheit werden daher zunehmend zu einem Prüf-

gegenstand für Softwaresysteme (z.B. Überwachungs- und Steuersysteme, Web-Applikationen)
Regressionstest: Jede (Wartungs-)Änderung im System muß überprüft werden. In komplexen Systemen haben lokale Änderungen oft Auswirkungen auf das Gesamtsystem. Weil daher fast alle bisherigen Testfälle wiederholt werden müssen, sind automatische Tests sehr hilfreich.
Automatischer Test: erfordern fast immer viel Gestehungsaufwand. Die in der Regel sehr hohe Anzahl von durchzuführenden Testfällen bei der Auslieferung einer neuen Version macht eine automatisierte Durchführung der Tests ökonomisch vertretbar. Eingabewerte und Ergebniswerte werden dabei automatisch verglichen; Fehlerfälle werden gesammelt und angezeigt.

Eine ganze Palette von speziellen testaktiven SE-Tätigkeiten werden unter dem Begriff dem Qualitätsmanagement zusammengefaßt (z.B. Reviews und Inspektionen, orthogonale Fehlerklassifikation, interner Beta-Test, ...). Die passende Auswahl ist nach den Erfordernissen der jeweiligen Systembedingungen und des Entwicklungsszenarios als Risiko/Kosten/Nutzen-Entscheidung zu treffen. Die beste Planung wird durch eine unpräzise Umsetzung konterkariert. Mangelnde Testqualität ist nicht unmittelbar spürbar und erfordert ein qualifiziertes und nachhaltiges Projektmanagement.

3. Spezielle Verhaltens- und Vorgehensweisen für das Testen objektorientierter Software

Das Testen von Software-Systemen ist eine Sache der Wahl der passenden Mittel zur konkreten Problemstellungen. Für Systeme, deren innere technische Basis objektorientiert ist, ist das nicht anders: es geht um die Verifikation einer Spezifikation mit angemessenen Mitteln. Für den ausführenden Ingenieur gibt es bei näherer Betrachtung dann doch deutliche Spezialitäten gegenüber traditioneller Technik.

Bei traditioneller Entwicklungsweise gilt der Integrations- und Gesamtsystemtest als eine höchst kritische Phase im Konstruktionsprozeß - ein "wahrer Hammer". Es werden Code-Teile unterschiedlichster Teams und Autoren zusammengefügt und alle Ungereimtheiten, Mehrdeutigkeiten, Inkonsistenzen, Mißinterpretationen, Eigendefinitionen, etc. werden mit einem Schlag sichtbar und virulent. Erfahrene Projektleiter wissen, daß sich die Ursachen dieser Fehlstände nicht nur auf der Implementierungsebene sondern schon in Design-Inkonsistenzen oder verschiedenen Auslegungen der Analysedokumente wiederfinden. Die Phase der Codierung enthält fast immer versteckte Formen der Post-Analyse und des Post-Designs. Kommunikationsfehler im Entwicklungsvorgang werden bei der Integration sichtbar, was häufig zu massivem Re-Coding, Re-Design und Re-Analysen führt. Wie schon bemerkt: nicht der eigentliche Test dauert so lange, sondern das Beheben der entdeckten Fehler.

Die Anwendung der OO Software-Entwicklung (OOSE) in probater Art und Weise kann in der Regel helfen, die Kritikaliät dieser Integrationsphase zu dämpfen, indem der Intensitätsverlauf deutlich abgeflacht wird: Ein OO-System besteht aus einer größeren Anzahl von miteinander kommunizierenden Objekten. Die Aktivierung (der „Aufruf“) eines Objektes erfolgt über Nachrichten (Parameter, Stimuli). Objekte enthalten im Falle guter Struktur und Konstruktion eine angemessene Menge an Daten (Attribute, Zustände) und Prozeduren (Abläufe, Verhalten, Algorithmen). Sie sind daher i.A. semantisch größere, komplexere Einheiten und Komponenten als konventionell strukturierte Systembausteine. Ein effektiv entworfenes Objekt ist ein Hort der konzeptiven Lokalität, Dichte und Kompaktheit von Information und Verhalten.
Solche Objekte verfügen über wohldefinierte Interfaces und jedes Systemverhalten wird nur über die Kommunikation zwischen solchen Objekten wirksam. Nur so kommt es zu Teilergebnissen. Diese Grundsystematik führt implizit zu einer viel frühren Integration. Teile des traditionellen Integrationstest werden vorgezogen. Entwickler sind angehalten kleinere Routinen nur im Rahmen von sinnhaften Objekten zu entwerfen. Die Grundbausteine des Systems werden dadurch größer. Die OO-Systematik erzwingt damit eine Teilintegration als Nebeneffekt. Ein getestetes Objekt ist eine größere getestete Baueinheit. Die Integration mehrerer Objektklassen verschiedener Teams verursacht daher in der Regel "ein kleineres Drama".
Dazu kommt der weitere gewollte Nebeneffekt, daß gute OOSE ähnliche Begriffe, Strukturen und Modellformen in Analyse, Design und Codierung verwendet. Damit werden die Umbrüche an der Wirklichkeit/Modell- und Modell/Code-Schwelle geringer und führen zu einer kleineren Anzahl "stiller" Mehrfachinterpretationen in der Codierung. Abbildung 2 visualisiert die Fortschritte in der Aufwandsverteilung der Software-Entwicklung.

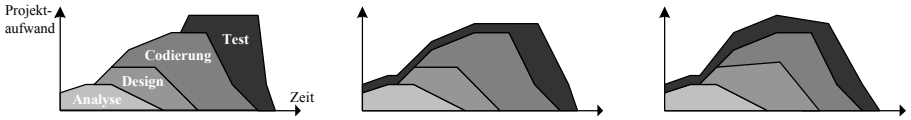

Abb. 2: Schematisierte Darstellung der Aufwandsverläufe in der Software-Entwicklung.
Der Testaufwand wird früher wirksam. OOSE ändert das Verhältnis Design zu Codierung.

Aber natürlich kann auch die OOSE existierende Systemkomplexität nicht zum Verschwinden bringen. Objekte sind zwar mächtige Software-Bausteine, aber ein schlechtes Objektmodell ergibt im günstigsten Fall eine unübersichtliche „Spagetthi-Struktur“. Im schlechten Fall ist sie wie Hausbauen mit Komponenten von der Halde: irgendwie paßt nichts zusammen. Betrachtet man den Gesamtprozeß erkennt man: konventionelle Entwicklung basiert letztlich auf kleineren Grundeinheiten, die im Konstruktionsfinish zu relativ hoher Integrationskomplexität führt. OO-Systematik braucht viel Intensität und Qualität bei der Grundmodellierung. Wenn ein problem-

entsprechender Entwurf gelingt, dann ergibt sich eine deutliche Entspannung bei der Systemintegration.
Der Zusatzaufwand (ja das Risiko) (k)ein effektives OO-Design zu erzielen, ist letztlich dadurch zu motivieren, daß die OO-Systematik spätere Änderungen und Ergänzungen über die Mechanismen der Vererbung von Methoden deutlich erleichtert.

Allerdings ist es gerade die Vererbung, die den Vorgang des Testens schwieriger macht als in konventionellen Systemen. Je nach Programmiersprache erben Klassen Operationen von Vorgängerklassen und zwar oft über Mechanismen der dynamischen Bindung. Manchmal ist dann der Test einer Operation in der Hierarchie weiter oben ohne Test in den Folgeklassen einfach nicht ausreichend. Die Modifikation einer Operation in einem Vorgänger wird den Test dieser Operation in allen Nachfolgern erfordern. Ebenso muß man in der Regel nach dem Hinzufügen einer neuen Nachfolgeklasse alle Operationen testen, die diese Klasse erbt. Regressionstesten und Automatisches Testen sind hier in der Praxis offenbar sehr wichtig.

Im Bereich des White-Box-Testens gilt es darüber hinaus zu beachten: nicht alle Testdaten können direkt übernommen werden, wenn z.B. Operationen in einer neu hinzugefügten Nachfolgeklasse überschrieben werden. Vererbungshierarchien können somit recht testintensiv werden. Weitere Test-Komplexität ergibt sich auch durch die Möglichkeit des Polymorphismus: da in vielen OO-Sprachen der Sender einer Nachricht den Typ des Empfängers nicht kennt, braucht ein qualifizierter White-Box-Test spezielle Test-Konzepte für verschiedene Typen von Nachrichten.

3.1 Kern des OO-Testens: Klassen und Klassenhierarchien

Der Test kleinerer Programmteile besteht traditionell im Testen von Prozeduren, Funktionen und Subroutinen. Die Basiseinheit ist im vorliegenden Fall das Objekt bzw. die Klasse. Ein Objekt hat seine Daten, sein Verhalten und seine Zustände. Neben den klassischen Strukturtests und Spezifikationstests ist es dabei sehr zweckmäßig, den Zustandstest als dritte Säule für den Basiskomponenten-Test vorzusehen. Objekte haben einen Zustand und einen Zustandsverlauf, dessen Prüfung ein exzellentes Testmittel darstellt.

a) Der *Spezifikations-Test (Black-Box-Test)* von Klassen befaßt sich mit der Prüfung der Input/Output-Beziehungen der Klasse. Diese Testform unterscheidet sich nur durch die Nomenklatur vom konventionellen BB-Test. Testfälle gewinnt man durch die Bildung geeigneter Äquivalenzklassen der Parameter. Ein simpler aber häufiger Fehler in der Praxis: Testfalldefinition ohne entsprechende Ergebnisdefinition und -prüfung.

b) Der *Struktur-Test (White-Box-Test)* von Klassen hat das Ziel der Prüfung der inneren Programmstruktur, des Codes innerhalb der Klasse. Es geht dabei um eine effektive Kombination von Bereichen der Parametern, Variablenwerten sowie den Grad der Deckung der Programmpfade. Auch hier bleibt es vorerst beim Althergebrachten:

Reduktion der kombinatorischen Vielfalt auf repräsentative Testfälle für Variablen und Parameterwerte sowie Programmpfade. Kritischen Bereichen begegnet man mit mehr Test-Qualität (detaillierterer Testplan, spezifischere Menge an Testfallgruppen). Polymorphismus ist ein für den Entwickler angenehmer Freiheitsgrad führt aber u.a. auch dazu, daß man im Code weniger gut sieht, was exakt passiert. Jede Nachricht in einem polymorphischen Code-Stück entspricht einem CASE-Statement im prozeduralen Code. Das führt im Einzelfall schon zu einer deutlich aufwendigeren Testplanung (siehe 3.2)

Ein weiterer Punkt betrifft die Vererbung. Diese ist prinzipiell ein „testfreundlicher" Mechanismus, weil sie hilft, mit weniger Code auszukommen, indem in Nachfolgeklassen geerbter Code wiederverwendet wird. Allerdings braucht man jedenfalls Testfälle für die Erzeugung von Instanzen selbst sowie den Test dieser Instanzen. Wenn eine Klasse eine Operation von einem Vorgänger erbt, dann muß sie trotzdem „in der neuen Umgebung" getestet werden. Auch das Überschreiben von Operationen im Rahmen der Vererbungshierarchie erfordert eigene Testfälle. Daraus wird ersichtlich, die Vererbung ist zwar ein mächtiges Prinzip, kann aber das Testen intensivieren. Wenn ein Programmsystem sehr stark auf Vererbungsmechanismen basiert, dann sind spezielle Testfälle für jede Ebene der Hierarchien unumgänglich.

c) Der *Zustandstest* prüft das Softwaresystem entlang des Zustandsverlaufes eines Objektes. Ein solcher Zustand wird repräsentiert durch die Werte der Attribute zu einem bestimmten Ausführungspunkt des Systems. Der Zustand eines Objektes ist eine signifikante, einprägsame Orientierung für die Entwickler, die sich auch vorzüglich bei sogenannten „Debuggern" bei objektorientierten Entwicklungssystemen einbauen läßt. Der Objektzustand motiviert eine natürliche Gruppe von Testfällen: jeder relevante Zustand eines Objekts sollte beobachtet und geprüft werden. Praktisch stellt man dafür eine *Zustandsmatrix* auf, die auf einer Achse alle relevanten Zustände eines Objektes und auf der zweiten Achse die Nachrichten an das Objekt abbildet. Diese einfache Test-Matrix lenkt u.a. auch die Aufmerksamkeit des Entwicklers auf solche Spezialfälle, die in der konstruktiv/synthetisierenden Denkweise des Designers leicht übersehen werden. Die Zustandsmatrix liefert eine sehr nützliche Quersicht auf Wertekombinationen, bei denen das Objekt eine bestimmt Art des Verhaltens zeigen soll. Hier findet sich ein wirklicher Vorteil der OO: Testfälle für diesen Test auf strukturell recht tiefer Ebene mit White-Box-Elementen, können schon aus Analyse- und Design-Dokumenten gewonnen werden. Natürlich geht es auch hierbei um gute Signifikanz und repräsentative Äquivalenzgruppen: nicht alle theoretisch möglichen Kombinationen aus Attributen und Nachrichten können schematisch berücksichtigt werden.

3.2 Einige praxisbezogene Aspekte der Realisierung von Tests in der OOSE

Beim Testen von konkreten objektorientierten Programmen mit den angeführten Testmethoden sind erfahrungsgemäß folgende einfache Hinweise mit zu beachten:

- *Fremde Klassenbibliotheken:* White-Box Testen erfordert den Zugang zum Source Code. Zugekaufte Komponenten können daher manchmal nicht mit dieser Methode getestet werden. Beim Import fremder Bibliotheken sollte man daher die Auswirkungen solcher Entscheidungen auf spätere Entwicklungsphasen eingehend diskutieren.
- *Testreihenfolge*: White-Box Tests können üblicherweise erst nach Fertigstellung der Implementierung definiert werden, da erst zu diesem Zeitpunkt die Code-Struktur endgültig ist. Jede Änderung im Code erfordert eine Änderung der Testfälle. Die Definition und Durchführung von White-Box-Tests ist in der Regel aufwendig und sollten erst nach Black-Box- und Zustands-Test erfolgen.
- *Außen hui, innen ...:* Black-Box Testen kann blenden. Die Definition der Testfälle kann bereits während der Analyse (System Black-Box) bzw. dem Design (Klassen Black-Box) begonnen werden, da die Schnittstellen zu den Benutzern bzw. zwischen den Klassen zu diesen Zeitpunkten größtenteils schon fixiert werden. Sowohl Definition als auch Ausführung sind im Vergleich zu White-Box Tests unaufwendiger. Ein erfolgreicher Black-Box-Test sagt aber möglicherweise wenig über die innere Stabilität und Komplexität einer Klasse aus. Die White-Box-Qualität einer Klasse bestimmt aber wiederum ihren wirklichen langfristigen (Projekt-)Wert.
- *Ablaufkomplexität zwischen den Klassen*: Sowohl bei White-Box als auch Black-Box Verfahren auf Klassenebene werden klassenübergreifenden Zusammenhängen kaum erfasst. Die Durchführung von Tests zur Überprüfung dieser klassenübergreifenden Abläufen kann manchmal aufwendig werden, darf aber keinesfalls im Vertrauen auf die korrekte Funktion der Einzelklassen vernachlässigt werden. Sollte vorhandene Komplexität hier nicht schon im Entwurf durch eigene Ablauf-Objekte modelliert worden sein, dann ist eine zusätzliche Ebene des Testens zwischen Klassen- und Integrationstests anzuraten.
- *Gesamtsystemzustände*: Ein globaler Systemzustand in traditionellen prozeduralen Systemen wird in der OOSE durch eine Vielzahl von lokalen Objektzuständen substituiert. Die Analyse des Systems zu bestimmten Zeitpunkten (z.B. nach Auftreten eines Fehlers) kann dadurch aufwendiger und die Fehlereingrenzung schwieriger werden, wenn z.B. bei einem Fehler mehrere Objekte falsche Zustände aufweisen.

Wie bereits erläutert wird das automatisierte Testen dann unumgänglich, wenn häufig Regressionstests erforderlich sind. Automatische Tests werden häufig auf zwei Ebenen durchgeführt:

- auf Systemebene im Rahmen des funktionalen Systemtests mit Werkzeugen wie Capture/Replay Tools sowie im Rahmen von Streß- und Lasttest mit eigens dafür entwickelten Werkzeugen
- auf der Klassenebene mit Hilfe von eigenen Klassen, welche die Testfälle für eine bestimmte zu testende Klasse implementieren.

Die Automatisierung auf Systemebene unterscheidet dabei nicht zwischen Systemen objektorientierter oder anderer Art, da die Testfälle nur über die vorhanden System-

schnittstellen wie der Anwenderschnittstelle ausgeführt werden und die innere Struktur definitionsgemäß nicht relevant ist. Bei der Automatisierung von Tests auf Klassenebene sind - wie schon in 3.1 bemerkt - mögliche polymorphe Belegungen von Variablen bei der Implementierung von Testfällen speziell zu berücksichtigen:

- Betrachtet man beispielsweise die Eingangsparameter von Methoden, so muss bei der Überprüfung von Methodenaufrufen sichergestellt werden, dass die Methodenaufrufe mit allen (Sub)-Typen des deklarierten Typs überprüft werden, um das Verhalten der möglicherweise modifizierten Methoden der Subklassen des deklarierten Parametertyps innerhalb der aufgerufenen Methode in die Tests einzuschließen.
- Bei der Verwendung von Rückgabewerten einer Methode kann nicht davon ausgegangen werden, dass alle zukünftigen Implementierungen der Methoden in Subklassen die Konventionen in Bezug auf Fehlerbehandlung einhalten. Daher ist sicherzustellen, dass in Codeteilen mit Methodenaufrufen von Variablen mit möglicherweise polymorphen Inhalt bei der Rückkehr von der Methode genügend Überprüfungen vorgenommen werden (z.B. Überprüfung auf Null-Werte und Überprüfung von Exceptions), damit Methoden von neuen Subtypen zu keinem unerwünschten Verhalten führen können.

Für die Durchführung der automatischen Tests stellt auch das Information Hiding der Klassen ein unübliches Hindernis dar. Während Werkzeuge wie Debugger (z.B. mit Hilfe von Sprachfeatures wie Reflexion oder Introspektion) auch private Teile eines Objektes darstellen können, ist dies auf Klassenebene bei der Auswertung der Ergebnisse in einer Testklasse oft unmöglich bzw. äußerst aufwendig.
Eine praktische Lösungsmöglichkeit besteht darin, den Source-Code um Methoden zu erweitern, welche Zugriff auf eben diese privaten Datenstrukturen erlauben bzw. Code zum Zwecke des Testens (z.B. für Tracing-Nachrichten) bereitstellen würde. Dieser Ansatz der Code-Instrumentierung bringt folgende Risiken mit sich:

- Die Einbau und Entfernung dieser Methoden erfordert zusätzlichen Aufwand.
- Die getestete Klasse ist nicht identisch mit der Klasse im finalen Produkt. Die zugehörigen Änderungen sind eine mögliche Fehlerquelle.
- Die während der Testphase zur Verfügung stehenden Methoden beinhalten das Risiko der unbefugten Benutzung, sofern es nicht strikte Mechanismen für die Nichtverwendung gibt. Widrigenfalls zerbricht ein klares Design rasch am schlampigen Programmieren.

Ein weiterer Ansatz ergänzt den Code mit Informationen, die für die Tests benutzt werden können, lassen den Code aber unverändert (z.B. in Form von speziell gestalteten Kommentaren). Damit ist auch ein Zugang zum nicht öffentlich zugänglichen Zustandes des Objektes denkbar. Die in Eiffel realisierten Assertions implementieren dieses Konzept. Sie ermöglichen in Ergänzung zum Source Code die Definition von Bedingungen, welche vor und nach der Ausführung von Methoden auf deren Gültigkeit überprüft werden und damit den Zustand des Objektes überprüfen können.

Ein andere aktueller Ansatz ist die Aspekt Orientierte Programmierung (AOP). Statt existierenden Source-Code zu verändern oder diesen durch zusätzliche Information zu ergänzen (was die Lesbarkeit und damit auch Wartbarkeit verringert), werden jene Aspekte, welche in einer bestimmten Situation während der Software-Entwicklung von Interesse sind (z.B. Zustand eines Objektes während des Testens) losgelöst von der Klassendefinition eigens implementiert. Aspekt Orientierte Programmierung ermöglicht die Betrachtung von Aspekten über mehrere Objekte hinweg und damit auch die Analyse von komplexen Systemzuständen. Mit Hilfe spezieller Werkzeuge werden Test- und Tracking-Aspekte während der Tests der Klasse hinzugefügt und mit ihr ausgeführt. Der Source Code der Klasse bleibt immer unverändert und wird auch durch diverse Zusatzinformation nicht unnötig erweitert.

4. Resümee

Trotz ihrer für die Gesamtqualität eines Softwaresystems so wichtigen Rolle wird die Phase des Testens im industriellen Alltag der Software-Entwicklung nach wie unterschätzt. Mangelnde Testvorbereitung verursacht überraschende Aufwände zu unerwartenden Zeitpunkten. Im vorliegenden Paper haben wir darauf hingewiesen, wie dem entgegengetreten werden kann.
Es liegt in der Natur des Funktionstests, daß er gewissermaßen eine "Tendenz der optimierten Destruktion" hat: mit möglichst wenigen Testfällen sollen möglichst viele Fehler gefunden werden. Aus diesem Grund sind - das zeigt die Praxiserfahrung – Designer und Programmierer oft schlechte Tester. Und gute Tester sind noch lange keine guten Entwickler. Letztlich ist das Testen wie vieles in der Softwareentwicklung eine Frage des geeigneten und verfügbaren Personals. Das Testen eines größeren OO Software-Systems erfordert ähnlich wie die Phase des OO-Designs einen einschlägig qualifizierten Spezialisten.

5. Literatur

[Ab98] Abrahams, M.; Barkley, J.: RTL verification strategies. Wescon/98, 1998.

[Be90] Beizer, Boris: Software Testing Techniques. International Thomson Computer Press, 1990.

[Bi99] Binder, Robert: Testing object-oriented systems: models, patterns, and tools. Addison-Wesley, 2nd print, 1999.

[Ci02] Ciolkowski, Marcus et al.; Software Inspections, Reviews & Walkthroughs. Proceedings of the 24rd Int. Conference on Software Engineering, 2002.

[Da94] Davis, A.M.: Fifteen principles of software engineering. IEEE Software , vol. 11, no. 6, Nov. 1994.

[Du01] Dunsmore, A.; Roper, M.; Wood, M. : Systematic object-oriented inspection - an empirical study. Proceedings of the 23rd International Conference on Software Engineering, 2001.

[El01] Elrad, Tzilla; Filman, Robert E.; Bader, Atef: Aspect-Oriented Programming: Introduction. Communications of the ACM, vol. 44, no. 10, 2001.

[Fi94] Fiadeiro, Jos. Luiz; Maibaum, Tom: Verifying for Reuse: foundations of object-oriented system verification. Theory and Formal Methods, 1994.

[Ki01] Kikuchi, N.; Kikuno, T. : Improving the testing process by program static analysis. Eighth Asia-Pacific Software Engineering Conference, 2001.

[Le91] Leavens, G.T. : Modular specification and verification of object-oriented programs. IEEE Software , vol. 8, no. 4 , July 1991

[Me92] Meyer, Betrand: Eiffel: The Language. Prentice Hall, 1992

[Mc97] McDonald, J.; Murray, L.; Strooper, P.: Translating Object-Z specifications to object-oriented test oracles. Asia Pacific Software Engineering Conference and International Computer Science Conference 1997, 1997.

[Ra99] Rangaraajan, Kr.: How can I test Java Classes? Dr.Dobb´s Journal, July 99.

[Ro99] de Rooij, R.C.M.; van Katwijk, J.: An approach towards formal verification of object oriented real-time systems. Sixth International Conference on Real-Time Computing Systems and Applications, 1999.

[Zh94] Zhangn, L.; van Katwijk, J.; Brink, K.: Applying Software Engineering Principles in Train Control Systems. Proceedings of the IEEE Workshop on Real-Time Applications, 1994