

Test objektorientierter Systeme

Tobias Beichter

Hauptseminar Objektorientierter Entwurf

20. Februar 2002

1 Übersicht

Diese Ausarbeitung behandelt den Test objektorientierter Systeme. Es werden die besonderen Schwierigkeiten dargestellt, die im Gegensatz zu prozedural entwickelten Systemen beim Testen auftreten. Um den Test objektorientierter Systeme zu ermöglichen, werden zu den jeweiligen Schwierigkeiten entsprechend geeignete Testmethoden vorgestellt.

2 Einführung

Softwaresysteme bestehen aus einer Vielzahl an Softwarekomponenten, welche untereinander in Beziehung stehen. Die Qualität eines Softwaresystems hängt von der Qualität der einzelnen Softwarekomponenten und der Qualität der Beziehungen zwischen ihnen ab. Ein Teil zur Sicherstellung der Produktqualität ist der Test. Die Aufgabe beim Testen besteht darin, mit möglichst geringem Aufwand einen möglichst hohen Schaden zu vermeiden. Die Kosten des Schadens setzen sich zusammen aus der Fehlersuche, der Fehlerbehebung und den Kosten, die der Fehler verursacht hat. Ein Fehler ist dabei jede Abweichung der Implementierung von ihrer Spezifikation.

3 Traditionelle Testmethoden

Bei prozedural implementierten Programmen wird zwischen zwei Betrachtungsweisen unterschieden: Das ist zum einen der Black-Box-Test und zum anderen der Glas-Box-Test. Beim Black-Box-Test wird die zu testende Komponente nur als Black-Box betrachtet, das bedeutet, dass die Testfälle nur anhand der Spezifikation der Komponente erstellt werden. Dabei ist es irrelevant, wie die tatsächliche Realisierung der Komponente aussieht. Aus diesem Grund ist der Black-Box-Test nach wie vor nützlich und kann für den Test objektorientierter Systeme eingesetzt werden. Der Glas-Box-Test dagegen geht speziell auf die interne Realisierung, der zu testenden Methode ein. Als mögliche Testmethode gibt es die Zweigüberdeckung, bei welcher sämtliche Kanten des Kontroll-

flussgraphen abgedeckt werden. Diese Testmethode kann für OO-Systeme auch angewendet werden. Sie hat aber im Endeffekt nur eine geringe Aussagekraft, da bei einem guten objektorientierten Programmierstil die Methoden im allgemeinen klein und wenig komplex sind. Das gleiche gilt für die Bedingungsüberdeckung, Pfadüberdeckung und die Anweisungsüberdeckung. Diese Testmethoden können aber nützlich zum Aufspüren von totem Code sein.

4 Warum ist der OO-Test anders?

Die Objektorientierung stellt neue Features wie Vererbung, Polymorphismus und dynamisches Binden zur Verfügung. Diese neuen Freiheitsgrade stellen aber auch eine Vielzahl an neuen Fehlerquellen dar. OO fördert die Wiederverwendung von Klassen. Das bedeutet aber auch, dass die Objekte der Klassen in unterschiedlichen Kontexten wiederverwendet werden und nicht speziell für einen Kontext zugeschnitten sind. Jeder Kontext verwendet also eine Teilmenge der Funktionalität einer Klasse. Dabei ist zu beachten, dass es Abhängigkeiten zwischen Methoden und Attributen geben kann, was beim OO-Test berücksichtigt werden muss. Neu ist auch, dass Objekte sich wie Zustandsautomaten verhalten. Objekte befinden sich immer in einem bestimmten Zustand, welcher das Verhalten der Objekte beeinflusst. Ob eine Methode ausgeführt werden darf, welche Rückgabewerte sie hat oder wie der Endzustand des Objekts aussieht, ist vom Ausgangszustand des Objekts bzw. von anderen Objekten, die mit dieser Methode in Beziehung stehen, abhängig.

Der wesentliche Unterschied für den Test von objektorientierten Systemen im Vergleich zu prozeduralen Systemen, ist die Verlagerung des Kontrollflusses aus den Prozeduren in die Nachrichten, die zwischen Objekten versendet werden. Daher steht bei OOP der Integrationstest im Vordergrund, während es bei prozeduralen Systemen der Modultest ist.

Weil der Kontrollfluss bei OO-Systemen verteilt ist, sind die traditionellen statischen Testverfahren, wie Walkthrough- oder Inspektionstest nicht so leicht

durchzuführen, wie bei zusammenhängenden Prozeduren oder Funktionen.

Durch die Möglichkeit des dynamischen Bindens, bei der erst zur Laufzeit des Programms entschieden wird, welche Implementierung zu dem entsprechenden Methodenaufruf verwendet wird, ist ein Walkthroughtest gar unmöglich.

Um effektiv testen zu können, müssen die möglichen Fehlerquellen identifiziert und verstanden werden.

5 Kapselung

Die Kapselung ist zwar kein neues Feature bei der OO-Programmietechnik, weil es sie auch schon bei der modularen Programmietechnik gibt, aber sie hat einen wesentlichen Einfluss auf den Test. Wo traditionelle Systeme z.B. aus einem Dutzend Modulen bestehen, bestehen OO-Systeme dagegen aus einem Vielfachen an gekapselten Klassen. Damit ergibt sich eine viel feinere Granularität als bei modularen Systemen.

Die Kapselung bedeutet für den Test, dass nur die Schnittstellen nach außen sichtbar sind. Die Sicht auf die interne Realisierung und der damit verbundene Entwurf der Testfälle für den Glas-Box-Test ist nicht möglich, falls nur die Schnittstellen und nicht der Quellcode der Klassen von Fremdanbietern vorliegen.

Zur Testausführung müssen die Objekte in einen bestimmten Ausgangszustand gebracht werden, von welchem aus der Test beginnen soll. Die Kapselung verbietet aber den direkten Zugriff auf Attribute und erschwert damit, den gewünschten Ausgangszustand zu erreichen. Das gleiche Problem tritt beim Protokollieren bzw. Abfragen der Zwischenzustände eines Objektes auf, da hier wiederum nur die Schnittstellenoperationen verwendet werden können.

Als möglicher Ausweg können als korrekt befundene Methoden verwendet werden, um über Umwege die Zustände setzen bzw. abfragen zu können. Es muss dabei jedoch beachtet werden, dass nicht jede Methode in jedem Objektzustand ausführbar ist, und dass Methoden oft stark voneinander direkt oder über Attribute abhängig sind.

Als Lösung bieten sich Möglichkeiten an, die den direkten Zugriff auf die geschützten Methoden und Attribute der Objekte erlauben, z.B. können in C++ die friend-Funktion oder in Ada Child-Packages verwendet werden. Bei anderen Sprachen wie Java, die keine solche Möglichkeit unterstützen, gibt es nur die Möglichkeit von Built-In-Tests. Dies bedeutet, dass die Testtreiber direkt in die entsprechenden Klassen integriert werden, was aber zu einer Vermischung des zu testenden Codes und dem Code der Testtreiber führt. Dies muss aber nicht unbedingt als negativ be-

wertet werden, da dadurch die Testtreiber immer zur Verfügung stehen und leichter zur Implementierung konsistent gehalten werden können.

Weiter haben sich Assertions als hilfreich für den Test und hauptsächlich zur Prävention von Fehlern erwiesen. Diese ermöglichen es, Invarianten zu überprüfen, was z.B. beim Design-by-Contract zur möglichen Überprüfung der Umsetzung gehört. Dort wird im Prinzip ein Vertrag zwischen einer Methode und ihrer Umgebung gemacht. Dabei wird von der Umgebung die Erfüllung der Vorbedingungen gefordert, und als Gegenleistung garantiert die Methode die Erfüllung der Nachbedingungen. Dieses kann mit der Hilfe von Assertions am Anfang und Ende einer Methode überprüft und garantiert werden. Somit können Integrationsfehler vermieden werden. Um Einbußen bei der Laufzeit durch Assertions zu vermeiden, gibt es noch die Möglichkeit, diese nur bei Bedarf zu aktivieren.

Eine nicht zu empfehlende Alternative für den direkten Zugriff auf Objektzustände ist der Einsatz eines Debuggers. Er ermöglicht nur das Abfragen in einer speziellen Ausführung des Programmes und ist nicht zur Wiederverwendung geeignet. Debugger sind hilfreich, um die Gründe von Fehlverhalten zu ermitteln.

Abschließend lässt sich zur Kapselung sagen, dass sie Bugs durch die Erhöhung des internen Zusammenhalts und der Verminderung der Kopplung reduziert, wodurch es weniger Abhängigkeiten und somit auch weniger mögliche Fehlerquellen gibt. Den Vorteilen der Kapselung steht ein erschwertes Testen gegenüber, falls der Testtreiber außerhalb der zu testenden Komponente liegt.

6 Dynamisches Binden

Als neues Feature bei der OO-Programmietechnik gibt es das dynamische Binden. Dabei wird erst zur Laufzeit des Programms entschieden, welche Implementierung für den entsprechenden Methodenaufruf ausgeführt wird. Aus diesem Grund ist es schwer nachzuvollziehen, warum das ausgeführte Programm ein Fehlverhalten verursacht hat, da es mehrere Möglichkeiten einer Implementierung geben kann. Durch diese Delokalisierung der möglichen Implementierungen wird die Wahrscheinlichkeit von Fehlern erhöht.

Der sogenannte JoJo-Effekt verdeutlicht das Problem. Sobald die Klassenhierarchien breiter und tiefer werden, steigt die Gefahr, dass Fehler beim dynamischen Binden gemacht werden. Bei dem Versuch die Nachrichtensequenzen nachzuvollziehen, entsteht ein Gefühl, wie wenn man der Bewegung eines JoJos folgt. Jedesmal, wenn ein Objekt sich selbst eine

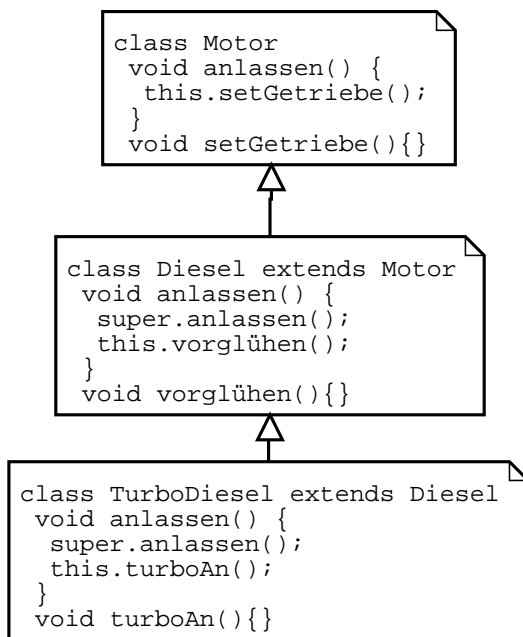


Abb. 1: Code für den JoJo-Effekt

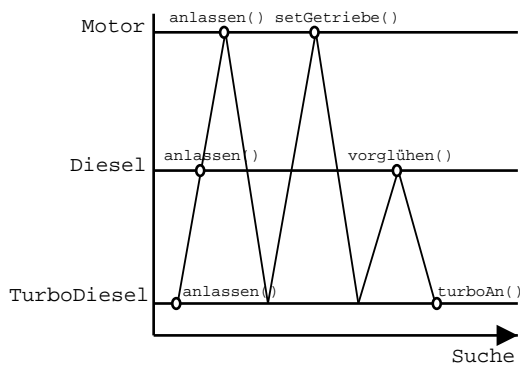


Abb. 2: Der JoJo-Effekt

Nachricht schickt, wird die entsprechende Methode in der Klasse gesucht, von der dieses Objekt instanziiert wurde. Falls diese dort nicht implementiert ist, wird in ihrer Oberklasse geschaut usw. bis die Implementierung gefunden wurde.

In dem Beispiel von Abbildung 1 und 2 wird ein Objekt von der Klasse TurboDiesel instanziiert und die Methode anlassen() aufgerufen. Diese ruft dann die Methode anlassen() aus der Klasse Diesel und diese wiederum die Methoden anlassen() in der Klasse Motor auf. Jetzt wird die Methode setGetriebe() aufgerufen. Der Beobachter muss zuerst in der Klasse TurboDiesel nach ihrer Implementierung schauen, bis er aufgestiegen ist, und sie in der Klasse Motor findet. Der Beobachter muss sich im klaren sein, von welcher Klasse das Objekt ist, um dann die entsprechenden Methodenaufrufe nachvollziehen zu können.

Anhand dieses kleinen Beispiels ist es klar, dass bei großen und breiten Klassenhierarchien schnell der Überblick verloren geht. Gerade wenn in so einer Klassenhierarchie Fehler gefunden werden müssen, Reengineering betrieben wird oder die Klassenhierarchie erweitert werden soll, entstehen schnell Fehler. Dieser Effekt verstärkt sich noch, wenn es sich bei der Vererbung nicht um eine echte Spezialisierung handelt, sondern diese dazu verwendet wird, Code zu kopieren.

Ein weiterer Nachteil der Delokalisierung macht sich beim Testen der Bindungen der Client-Klasse bemerkbar: Es müssen erst sämtliche Server-Klassen, die von der Client-Klasse benutzt werden, integriert werden. Wenn Änderungen an Server-Klassen gemacht werden, z.B. durch Korrigieren gefundener Fehler, müssen unter Umständen sämtliche Client-Klassen angepasst werden.

Um die beteiligten Client- und Server-Klassen beim dynamischen Binden testen zu können, muss jede mögliche Bindung getestet werden. Dies wird zum einen mit Testtreibern ermöglicht, die für jede Server-Klasse testen, ob jede mögliche Bindung korrekt behandelt wird, damit Client-Klassen die Server-Klassen nicht falsch benutzen können. Mit den Methoden der Client-Klassen kann im Gegenzug überprüft werden, ob sie die Server-Klassen richtig benutzen. Da jede mögliche Bindung getestet werden soll, ist der Überdeckungsgrad nur schwer erreichbar. Mit dem Liskov'schen Substitutionsprinzip oder dem Design-by-Contract von Meyer (siehe oben) kann man sich vor unerwünschten Nebeneffekten und Problemen bei der Integration von Klassen schützen, und erhöhen somit die Korrektheit und Robustheit der Klassen. Das Liskov'sche Substitutionsprinzip besagt, dass alle Unterklassen durch ihre Oberklasse substituiert werden können.

7 Vererbung

Die Einführung des Programmierkonzepts Vererbung ermöglicht es, Unterklassen von Basis-Klassen abzuleiten. Dabei haben die Unterklassen die Eigenschaften der Basis- bzw. Oberklassen, solange diese nicht redefiniert werden. Die Vererbung stellt eine Generalisierungs- oder Spezialisierungsbeziehung dar.

Der Test muss berücksichtigen, dass die Vererbung neue Fehlerquellen entstehen lässt. Das Kapselungsprinzip verlangt, dass der Objektzustand nur über die öffentlichen Schnittstellen geändert werden darf. Die Vererbung lockert das Kapselungsprinzip dadurch, dass die Methoden jetzt nicht mehr nur in der Klasse des Objektes zu suchen sind, sondern auch in deren Oberklassen.

Mehrfaches und wiederholtes Erben erschwert die Verständlichkeit und erhöht damit die Fehleranfälligkeit der Architektur des Systems. Das kann soweit führen, dass unbewusst eine Methode überschrieben wird und damit ein Fehlverhalten des Objekts provoziert wird. Dieser Fall ist in einem Studienprojekt tatsächlich passiert. Es wurde ein Open-Source Grapheneditor verwendet. Um diesen an die eigenen Bedürfnisse anpassen zu können, wurde von dessen Klassen abgeleitet und das gewünschte Verhalten implementiert. Diese spezialisierten Unterklassen hatten noch weitere Methoden von denen eine `getProperty()` hieß. Durch eine neue Release des Grapheneditors gab es jetzt in der entsprechenden Oberklasse eine neue Methode, die ebenfalls `getProperty()` hieß. Damit wurde diese neue Methode durch die alte Implementierung der spezialisierten Unterklasse in unbeabsichtigter Weise überschrieben. Dies führte am Anfang zu einem unerklärlichen Fehlverhalten. Dies hätte vermieden werden können, wenn die geänderten Klassen immer im Kontext der Vererbungshierarchie betrachtet worden wären.

Bei der Mehrfachvererbung müssen die Klassen im Kontext sämtlicher Oberklassen betrachtet und getestet werden. Geerbte Methoden und Attribute aus getesteten Oberklassen müssen im Kontext der abgeleiteten Klassen nochmals getestet werden, da nicht davon ausgegangen werden kann, dass diese in Verbindung mit redefinierten Methoden noch das gewünschte Verhalten haben. Selbstverständlich sind alle redefinierten Methoden im Kontext ihrer Klasse zu testen. Es besteht jedoch die Möglichkeit, dass ein Teil der Testfälle von den Methoden der Oberklasse wiederverwendet werden können, indem sie auf die geänderten Anforderung angepasst bzw. erweitert werden.

8 Der dynamische Testprozess

Im Vergleich zu statischen Testverfahren, bei denen die zu testende Komponente nicht ausgeführt sondern nur analysiert wird, verlangen die dynamischen Testverfahren die Ausführung des Quellcodes, um Fehler zu finden. Der dynamische Testprozess findet nicht am Ende einer Implementierungsphase statt, sondern ist dieser parallel überlagert. Das ganze sieht dann so aus, dass sofort nach dem Fertigstellen oder der Integration einer oder mehrerer Komponenten diese getestet werden. Dabei ist der Testprozess folgendermaßen aufgebaut:

Der Prozess beginnt mit dem Test aller Klassen, bei denen die verschiedenen Arten von Klassen mit unterschiedlichen Testmethoden auf ihre Funktionsweise überprüft werden. Im folgenden werden logische und physikalische Teile des Systems, die sogenannten Subsysteme identifiziert und getestet. Sobald einige

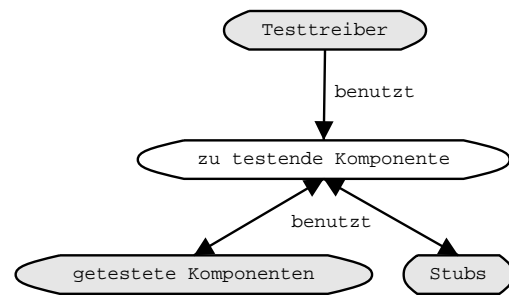


Abb. 3: Die Testumgebung

Komponenten integriert wurden, findet der Integrationstest statt. Nach der vollständigen Integration und deren Integrationstests findet der System- und Abnahmetest statt, der den Abschluss der Implementierung und der Tests unter realen Einsatzbedingungen bildet.

9 Das Testgeschirr

Um Komponenten effektiv und wiederholt automatisch testen zu können, wird ein System benötigt, das die Komponenten initialisiert, ihre Methoden aufruft und entscheidet, ob der Test negativ oder positiv ist. Ein solches System ist das Testgeschirr (siehe Abbildung 3). Es besteht aus dem Testtreiber, der die zu testende Komponente und die von ihr benötigten Komponenten. Diese bestehen zum einen aus getesteten Komponenten und zum anderen aus Stubs.

Stubs oder auch Stümpfe haben nur scheinbar die benötigte Funktionalität implementiert und dienen als Ersatz für noch nicht implementierte oder getestete Komponenten.

Der Testtreiber hat die Aufgabe, die zu testenden Komponenten zu initialisieren. Bei einer Klasse bedeutet dies, dass von ihr ein Objekt instanziiert für den folgenden Test in den entsprechenden Ausgangszustand gebracht wird. Danach ruft der Testtreiber die entsprechenden Methoden mit den für die Testfälle ermittelten Eingabeparametern auf. Zwischen den einzelnen Aufrufen muss der Zustand der Komponente überprüft und protokolliert werden.

Die Überprüfung, ob ein Fehler entdeckt wurde oder nicht, findet anhand der Ist- und Soll-Resultate statt. Die Soll-Resultate werden mit Hilfe der Spezifikation ermittelt. Jede Abweichung von der Spezifikation stellt einen Fehler dar.

10 Der Klassentest

Bei OO-Systemen ist die Klasse die kleinste sinnvoll testbare Einheit, weil die Methoden im allgemeinen klein und stark abhängig von Attributen und anderen Methoden innerhalb der Klasse sind.

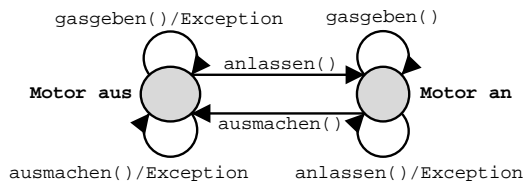


Abb. 4: Der Zustandsgraph zum Objekt der Klasse Motor

Objekte befinden sich immer in einem bestimmten Zustand. Das führt dazu, dass sich Objekte wie Zustandsautomaten verhalten und Zwischenzustände, Endzustand und Rückgabewerte vom Ausgangszustand abhängen.

Für den Test von Klassen kann man deren Verhalten in vier Klassen einteilen:

- Non-modale Klassen sind unabhängig von ihrem Zustand und unabhängig von der Reihenfolge ihrer Methodenaufrufe. Beispielsweise ist eine einfache Klasse zur Datenhaltung mit unabhängigen Attributen, welche mit einfachen `set-` und `get-`Methoden gesetzt und abgefragt werden eine Non-modale-Klasse.
- Uni-modale Klassen sind unabhängig von ihrem Zustand, aber abhängig von der Reihenfolge ihrer Methodenaufrufe. Beispiel: Bei einer Ampel kann die Methode `schalteAufRot` nur nach der Methode `schalteAufGelb` aufgerufen werden.
- Quasi-modale Klassen sind abhängig von ihrem Zustand und unabhängig von der Reihenfolge ihrer Methodenaufrufe. Eine verkettete Liste ist ein Beispiel für eine quasi-modale Klasse.
- Modale Klassen sind abhängig von ihrem Zustand und abhängig von der Reihenfolge ihrer Methodenaufrufe. Beispiel: Bei einer Gangschaltung muss die Schaltung im Zustand neutral sein, bevor die Methode `legeRückwärtsgangEin()` aufgerufen wird. Aus dem Zustand neutral sind auch nur die Methoden `legeRückwärtsgangEin()` oder `schalteInErstenGang()` erlaubt.

Da sich Klassen wie Zustandsautomaten verhalten, können Testfälle dadurch ermittelt werden, dass die Knoten, Kanten oder die Pfade des zugehörigen Graphen überdeckt werden. Ähnlich wie beim traditionellen Strukturtest kann dadurch der Überdeckungsgrad gemessen werden. Abbildung 4 zeigt den Zustandsgraph für die Klasse `MOTOR` aus der Abbildung 5.

Die Anzahl der möglichen Zustände ist umso höher, je mehr Attribute und Methoden eine Klasse

hat. Ein vollständiges Austesten einer Klasse ist normalerweise unwirtschaftlich bzw. unmöglich, da die Anzahl der Testfälle schnell astronomisch hohe Werte annimmt.

Die Grenzen des Versuchs alles testen zu wollen, sind schnell erreicht. Angenommen, es sollen alle möglichen Dreiecke aus drei Linien auf einem Bildschirm mit einer Auflösung von 1024x768 Pixel getestet werden. Es gibt folglich $(1024 * 768)^2$ Möglichkeiten eine Linie und $(1024 * 768)^6$ Möglichkeiten ein Dreieck zu zeichnen. Falls 1000 Dreiecke pro Sekunde getestet werden können, wird ungefähr die Zeit seit Beginn des Universums benötigt, um alle Kombinationen zu testen. Aus diesem Grund ist es die Kunst beim Testen, mit möglichst wenigen Testfällen möglichst viele Fehler aufzuspüren.

Robert V. Binder [Bin00] hat eine Checkliste für den Klassentest mit folgenden Punkten vorgeschlagen:

- Jede Methode wird ausgeführt.
- Jede ausgehende Exception wird ausgelöst, jede hereinkommende Exception wird behandelt.
- Jeder Zustand wird erreicht.
- Jede Methode wird in jedem Zustand des Objektes ausgeführt (richtig, wo es passend ist, verboten, wo es unpassend ist).
- Jeder Zustandsübergang wird ausgeführt.
- Passende Belastungs- und Misstrauenstests werden ausgeführt.
- Alle Parameter und Rückgabewerte werden mit Äquivalenzklassen- und Grenzwerttests geprüft.

Der Klassentest bezieht sich nicht nur auf die isoliert zu testende Klasse. Von ihr benutzte Klassen und ihre Oberklassen müssen dabei auch berücksichtigt werden.

Ist die zu testende Klasse eine Unterklasse, kann ein Teil der Testfälle in angepasster Form von der Oberklasse verwendet werden.

Folgende Punkte sollten bei der Erstellung von Testfällen berücksichtigt werden:

- Für alle neuen Methoden sind Testfälle aufzustellen und auszuführen.
- Für alle redefinierten Methoden sind neue Testfälle aufzustellen und auszuführen.
- Für alle geerbten Methoden müssen alle Testfälle der Oberklasse erneut durchgeführt werden, da der Kontext der Unterklasse anders ist.

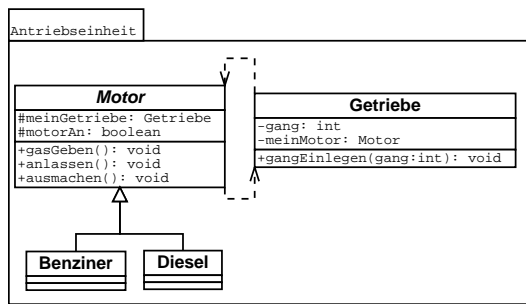


Abb. 5: Das Klassendiagramm des Subsystems Antriebseinheit

11 Test abstrakter Klassen

Es kann kein Objekt direkt von einer abstrakten Klasse erzeugt werden, da diese nur ein Interface darstellt. Sie definiert somit die syntaktische Schnittstelle für die in den konkreten Klassen zu implementierenden Methoden. Aus diesem Grund kann eine abstrakte Klasse nicht ohne eine konkrete Implementierung getestet werden.

Für die konkrete Implementierung empfiehlt es sich, bereits getestete Klassen zu verwenden oder möglichst einfache Stubs zu erzeugen, die mit wenig Aufwand die Spezifikation erfüllen. Diese können dann wie jede andere Klasse mit den verschiedenen Testmethoden getestet werden.

12 Test generischer Klassen

Generische bzw. parametrisierte Klassen können mehrere formale Klassenparameter besitzen. Diese Klassenparameter müssen zur Instanziierung eines Objekts angegeben werden.

Um eine generische Klasse testen zu können, muss folglich das zu instanzierende Objekt erst parametrisiert werden. Dies führt zu der Frage, mit welchen Klassenparametern dies geschehen soll? Bei generischen Klassen empfiehlt es sich, zuerst mit einer einfachen repräsentativen konkreten Klasse zu testen. Damit sollte die Klasse wie jede andere Klasse sinnvoll testbar sein.

Bei generischen Klassen mit mehreren Klassenparametern kann es vorkommen, dass das Verhalten des einen Parameters das Verhalten des anderen beim Kontroll- oder Datenfluss beeinflusst. Daher ist es sinnvoll, möglichst viele Typ-Kombinationen zu testen. Dazu gehört auch deren Permutation oder die Instanziierung mit mehreren Parametern vom gleichen Typ.

Als Parameter können auch Klassen einer Klassenhierarchie dienen. Um die generische Klasse damit testen zu können, wird die generische Klasse mit der

Basis-Klasse instanziiert, und alle Bindungen werden mit den Objekten der Unterklassen getestet.

Oft ist es sinnvoll, den Testtreiber für generische Klassen selbst generisch zu implementieren, damit er für viele Typen wiederverwendbar ist. Dabei muss darauf geachtet werden, dass typabhängige Testfälle, wie z.B. die Überprüfung der Grenzwerte bei Integer- und Float-Zahlen, auch typabhängig behandelt werden.

Eine parametrisierbare Klasse kann niemals vollständig getestet werden, da es unendlich viele Typen mit unterschiedlichem Verhalten gibt.

13 Test von Subsystemen

Subsysteme bestehen aus mehreren Klassen, die einen logischen oder physikalischen Teil der Anwendung darstellen. Sie sind nur als Ganzes ausführbar, wobei auch kleinere Teile einzeln testbar sind. Es ist sinnvoll ein großes komplexes System in Subsysteme zu unterteilen. Dadurch sind die Teilsysteme handhabbarer und können getrennt von den anderen Teilsystemen entwickelt und getestet werden. In Abbildung 6 sind die zwei Subsysteme Antriebseinheit und Lenkeinheit abgebildet.

Der Subsystemtest soll Fehler aufdecken und die Funktionsweise des Subsystems sicherstellen, um die spätere Integration mit anderen Subsystemen zu erleichtern. Falls in einem Teil des Systems keine Ressourcen für den Klassentest vorhanden sind, können diese oberflächlich mit dem Subsystemtest getestet werden. Aber er kann den Test der Klassen natürlich nicht wirklich ersetzen. Durch die Abstraktion der Klassen zu einem Subsystem können nicht alle Zustände der Klassen erreicht werden. Im allgemeinen werden nicht alle Methoden der Klassen benötigt und verwendet. Damit sind sie dann auch nicht über das Subsystem testbar.

Zum Subsystemtest gehört, dass Assoziationen zwischen Klassen getestet werden. Zu den möglichen Fehlern bei der Implementierung von Assoziationen gehören:

- Falsche Multiplizität
Die Multiplizität einer Assoziation gibt an, mit wievielen Instanzen der gegenüberliegenden Klasse das aktuelle Objekt über Referenzen verbunden werden kann. Die Implementierung weist eine gültige Anzahl zurück oder nimmt eine ungültige an. Um die Multiplizität testen zu können, überprüft man die Grenzwerte und eine typische Anzahl von Referenzen.
- Aktualisierungsanomalien
Falls unbeabsichtigt redundante Informationen

durch Instanzen und Verbindungen erstellt wurden, müssen diese alle bei einer Änderung aktualisiert werden. Um auf Aktualisierungsanomalien testen zu können, muss man eine Instanz aktualisieren und überprüfen, ob alle verbundenen Instanzen auch aktualisiert werden.

- Löschanomalien
Sobald eine Instanz gelöscht wird, muss dafür gesorgt werden, dass sich die Daten in einem zur Spezifikation konsistenten Zustand befinden. Es muss geklärt werden, was mit Instanzen passiert, die nur von der gelöschten Instanz benutzt wurden. Löschanomalien werden aufgedeckt, indem die Instanz gelöscht wird. Die zuvor bestehenden Referenzen sollten dann auch gelöscht werden, oder die Instanz darf nicht gelöscht werden, solange noch Referenzen darauf bestehen.
- Fehlende Verbindung
Es besteht die Möglichkeit, dass die Verbindung zwischen Objekten fälschlicherweise fehlt.
- Falsche Verbindung
Es kann vorkommen, dass Verbindungen zu falschen Instanzen erstellt werden.

Ähnliche Fehler treten auch bei Datenbanksystemen auf.

Im Beispiel aus Abbildung 5 muss die Assoziation zwischen der Klasse Motor und der Klasse Getriebe überprüft werden. Es muss z.B. überprüft werden, dass ein Objekt der Klasse Motor eine Referenz auf ein Objekt der Klasse Getriebe hat, und dieses genau das Objekt der Klasse Motor wieder referenziert. Sonst gibt es eine falsche Verbindung, die z.B. zu Aktualisierungsanomalien führen kann, wenn das Objekt der Klasse Getriebe aktualisiert wird.

Mit der Hilfe von UML-Sequenzdiagrammen kann der Kontrollflussgraph erstellt werden. Dieser kann mit traditionellen Testmethoden wie Zweig- oder Pfadüberdeckung getestet werden. Der Vorteil dabei ist, dass alleine durch die Erstellung des Kontrollflussgraphen Fehler erkannt werden. Desweiteren beschreiben UML-Sequenzdiagramm oft USE-Cases, die dabei überdeckt werden.

Subsysteme müssen auch mit eingehenden Exceptions von Server-Klassen umgehen können. Es gibt drei Möglichkeiten, dies zu testen:

- Der Server wird direkt oder die Server-Klassen werden derart durch USE-Cases manipuliert, dass die Exceptions ausgelöst werden. Das ist jedoch sehr schwierig, für die Hardware gefährlich oder gar unmöglich.
- Die Implementierung wird so manipuliert, dass die gewünschten Exceptions ausgelöst werden.

Dies hat den Nachteil, dass der Test nicht wiederverwendbar ist, wenn die Implementierung weiterentwickelt wird.

- Die sinnvollste Möglichkeit ist wohl, eine Wrapper-Klasse um die Server-Klassen zu bauen. Diese Wrapper-Klasse kann an der Schnittstelle zwischen dem Subsystem und den Server-Klassen die gewünschten Exceptions simuliert auslösen. Der Vorteil ist dabei, dass die Wrapper-Klasse bei Bedarf eingesetzt, erweitert und wiederverwendet werden kann und der Code der Client- und Server-Klassen nicht angefasst werden muss.

Die Testaxiome von Weyuker veranschaulichen die Grenzen der Übertragbarkeit von Testfälle für die Codeüberdeckung:

- Antikomposition
Unterschiedliche Kontexte benötigen normalerweise unterschiedliche Testfälle. Das bedeutet beispielsweise, dass wenn die Durchführung sämtlicher Klassentests eines Subsystems nicht automatisch auch den Subsystemtest erledigt.
- Antidekomposition
Ein ausreichender Test der Client-Klassen muss nicht zu einem ausreichenden Test ihrer Server-Klassen führen. Die Client-Klassen verwenden unter Umständen die Server-Klassen nicht vollständig, oder die Server-Klassen werden in einem anderen Kontext wiederverwendet.
- Antiextensionalität
Semantisch gleiche oder ähnliche Methoden können unterschiedliche Testfälle benötigen. Methoden können auf unterschiedliche Art und Weise implementiert sein, wodurch die Testfälle für die Testmethoden des Glas-Box-Tests anders gewählt werden müssen. Das Gleiche gilt beim Überschreiben von Methoden in abgeleiteten Klassen. Wenn die Basis-Klasse z.B. eine verkettete Liste und die abgeleitete Klasse eine geordnete und verkettete Liste darstellt, dann müssen die Testfälle in der abgeleiteten Klasse auch die Ordnung der Element überprüfen. Dies ist in der Oberklasse nicht nötig.

14 Der Integrationstest

Das Ziel beim Integrationstest ist es, dass möglichst alle Fehler in der Interoperabilität der Komponenten gefunden werden. Komponenten können dabei Methoden, Klassen, Cluster oder Subsysteme sein. Durch den Integrationstest vor dem Systemtest können dort auftretende Fehler schneller lokalisiert

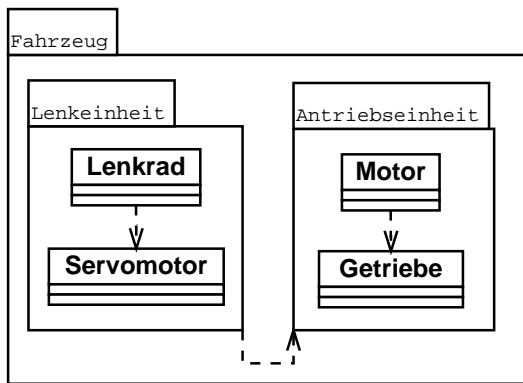


Abb. 6: Subsysteme Antriebseinheit und Lenkeinheit

werden, da meistens die zuletzt integrierten Komponenten Fehler verursachen. Weitere Gründe für den Integrationstest sind die bei Subsystemen besprochenen Axiome Antikomposition und Antidekomposition.

In dem Beispiel aus Abbildung 6 muss die Interoperabilität zwischen Lenk- und Antriebseinheit getestet werden. Dazu gehört, dass Abhängigkeiten, wie zwischen Lenkausschlag und Geschwindigkeit, überprüft werden.

Bei der Integration von Komponenten sind die verschiedenen Cluster bzw. starke Zusammenhangskomponenten im System zu identifizieren und ein Abhängigkeitsgraph zu erstellen. Je nach Abhängigkeit kann die entsprechende Integrationsstrategie verfolgt und die integrierten Komponenten getestet werden.

Ansätze für den Integrationstest sind:

- Ereignisgesteuerter Test [Jor94]
Für den Ablauftest werden sogenannte Methode-Message-Pfade (MM-Paths) verwendet. Bei den MM-Pfaden wird eine Methode ausgeführt, die weitere Methoden über Nachrichten aufruft. Ein MM-Pfad endet, sobald die letzte Methode keine neue Nachricht mehr erzeugt bzw. zur Ruhe kommt. Aus einem MM-Pfad können mehrere MM-Pfade entstehen. Durch ein Ereignis von außen wird die Ausführung des Systems veranlasst und führt zu einer bestimmten Reaktion des Systems. Solch ein Ablauf wird als Atomic System Function (ASF) bezeichnet.
- Kommunikationsbasierter Test [Jue95]
Die Strategie des kommunikationsbasierten Tests, die von einer Abteilung der Siemens AG entwickelt wurde, besteht aus zwei Hauptschritten:
Im ersten Schritt wird die Integration von Methoden innerhalb von Klassen getestet. Dazu werden Server-Klassen als Stubs implementiert.

Die Reihenfolge der Integration folgt der Vererbungshierarchie angefangen mit der Basis-Klasse und gefolgt von den abgeleiteten und den redefinierten Methoden. Im zweiten Schritt werden dann die Assoziationen integriert und getestet.

- Dienstleistungsbasierter Test [Ove94]
 1. Teste die neue Klasse isoliert.
 2. Teste alle Interaktionen der Klasse mit den Server-Klassen.
 3. Teste alle Interaktionen, bei denen die Klasse eine mehrerer Client-Klassen einer Server-Klasse ist.
 4. Teste alle Interaktionen, bei denen die Klasse eine Server-Klasse für eine Client-Klasse ist.
 5. Teste alle Interaktionen, bei denen die Klasse eine Server-Klasse für mehrere Client-Klassen ist.

15 Der Systemtest

Warum der Systemtest bei OO-Systemen notwendig ist, wenn die Klassen und Subsysteme schon getestet wurden, beschreibt das Antikompositionsaxiom.

Beim Systemtest wird das System als Ganzes getestet. Die Testfälle ergeben sich aus den spezifizierten USE-Cases und durch die Auftraggeber gewünschten Testfälle.

Das Ziel beim Testen ist es, den entstehenden Schaden durch Fehler möglichst gering zu halten und diese möglichst frühzeitig zu erkennen. Aus diesem Grund ist es ratsam, die Anzahl der Testfälle im Verhältnis zur Häufigkeit der verwendeten USE-Cases und zur Größe des Schadenspotentials zu wählen.

Falls es noch einfache Operationen auf Objekte im System gibt, die nicht durch die Testfälle der USE-Cases abgedeckt wurden, sind für diese noch Testfälle zu erstellen.

In dem Beispiel mit dem Fahrzeug sind die USE-Cases Lenkausschlag, Gasgeben, Gang einlegen, usw. zu testen. Zusätzlich sollten noch nicht erlaubte Funktionen überprüft werden. Beispiel: Das Objekt der Klasse Motor ist im Zustand `motorAn = false`, und es wird versucht die Methode `gasgeben()` auszuführen.

Es ist unerheblich, ob die interne Realisierung objektorientiert ist oder nicht, da der Systemtest ein Black-Box-Test ist.

16 Der Regressionstest

Der Regressionstest hat die Aufgabe, Fehler zu finden, die durch modifizierte Komponenten entste-

hen. Er ist daher gerade bei der Entwicklung von OO-Systemen und bei der Wartung wichtig, da hier verstärkt mit kurzen inkrementellen Zyklen die Software entwickelt oder angepasst wird. Die Testfälle werden wiederverwendet. Nach einer Modifizierung des Codes können die Testergebnisse vor und nach der Modifizierung verglichen werden. Dabei ist es, möglich Seiteneffekte durch die Modifizierung zu erkennen, falls nicht nur die von der Änderung betroffenen Testfälle anders reagieren. Für geänderte Komponenten müssen auch weiterhin entsprechende Testfälle erzeugt werden.

Angenommen es gibt im Beispiel die Anforderung für einen Grenzwert für die minimale Motorleistung. Dann wird dieser Grenzwert mit einem Testfall überprüft. Wenn die Klasse `Motor` aus irgendwelchen Gründen geändert wird, dass der Motor eine geringere Leistung als der Grenzwert hat, würde dieser Testfall diesen Fehler entdecken.

Bei Integrationsstrategien wie der Highfrequency Integration, wo immer alles sofort integriert wird, oder beim Extreme Programming ist das Regressionstesten zum Teil mehrmals am Tag nötig. Diese Art zu testen ist nur automatisiert möglich. Dabei kann der Umfang des Tests variieren. Man kann z.B. immer den vollständigen Regressionstest durchführen oder nur Testfälle, die die Änderungen, kritische USE-Cases, bestimmte Testprofile, usw. betreffen.

Für Fehler, die nicht durch den Regressionstest gefunden wurden, können zusätzliche Testfälle hinzugefügt werden, die diese Fehler aufdecken würden. Dies kann bei späteren Änderungen, oder wenn eine ältere Version mit dem entsprechenden Fehler wieder eingebaut wird, hilfreich sein.

17 Automatisierung

Testen von Softwaresystemen ist zum Großteil nur noch automatisiert möglich. Tausende von Testfällen können nicht über Nacht im Rahmen von Regressionstest von Hand durchgeführt werden. Zum einen würde dies zuviel Zeit und damit Kosten beanspruchen, und zum anderen kann ein Rechner mit einer viel höheren Genauigkeit Eingaben eingeben, Funktionen ausführen, Resultate vergleichen und Testprotokolle in Datenbanken ablegen.

Automatisierte Tests sind beim Korrigieren von Fehlern hilfreich und schon nach wenigen Testläufen kostengünstiger. Die Testprotokolle sind so einfacher aktuell und vollständig zu halten als bei manuellen Tests. Manuelle Tests können willkürlich eingegeben werden, ohne dass es spezifizierte Testfälle gibt. Sobald aber Testfälle automatisiert ausgeführt werden, sind zumindest diese Testfälle im Code der Testtreiber festgehalten und mit Kommentaren versehen.

Eine großes Problem bei der Automatisierung stellt noch das Testen von graphischen Oberflächen dar. Hier gibt es jedoch auch schon Ansätze, die das sogenannte Play-Back-Verfahren benutzen, oder die Durchführung wird per Skript gesteuert. Diese Tests sind jedoch relativ langsam.

18 Einordnung in den Prozess

OO-Systeme tendieren zur Entwicklung in kurzen inkrementellen Zyklen. Dies muss der Entwicklungsprozess berücksichtigen. Bei der Entwicklung wird nach folgendem Motto vorgegangen: Design a little, Code a little and Test little.

Statische Testmethoden wie Inspektionen und Walkthroughs sind wegen der Verteilung des Kontrollflusses weniger effektiv. Aus diesem Grund müssen weit mehr Ressourcen für den dynamischen Test eingeplant werden, als es bei prozeduralen Systemen nötig ist.

19 Zusammenfassung

Schon beim Entwurf sollte die Testbarkeit des Systems berücksichtigt werden. Dazu zählen klare Subsysteme, strikte Vererbung, keine komplexen Systemarchitekturen und das Umsetzen des Design-by-Contract Prinzips. Gerade in der Einfachheit und Verständlichkeit liegt die Stärke von erfolgreichen, erweiterbaren und wartbaren Systemen.

Für OO-Systeme gibt es viele Testmethodenansätze, deren Erprobung in der Praxis aber noch weitgehend unklar ist.

Abschließend läßt sich zum Testen objektorientierter Systeme sagen, dass Teile von den traditionellen Testverfahren, wenn auch nicht so effektiv, übernommen werden können. Für OO-Systeme sind neue Testmethoden entwickelt worden. Eine umfangreiche Sammlung ist in dem Buch von Robert V. Binder [Bin00] enthalten.

Literatur

- [Bin94] Robert V. Binder. Testing object-oriented systems: A status report. <http://www.rbsc.com/pages/ootstat.html>, 1994.
- [Bin00] Robert V. Binder. *Testing Object-Oriented Systems – Models, Patterns, and Tools*. Addison-Wesley, 2000.
- [Jor94] C. Jorgenson, P.C. ; Erickson. Object-oriented integration testing. *CACM*, 37(9):30–38, September 1994.

- [Jue95] P. Juettner. Integration testing of object-oriented software. *8th Int. Software Quality Week*, May 1995.
- [Lic01] H. Lichter. Folien zu testen objektorientierter software, 2001.
- [Ove94] Jan Overbeck. *Integration Testing for Object-Oriented Software*. Dissertation an der TU Wien, 1994.