Testgetriebene Entwicklung - ein Leitfaden

Verfasser	S. Frick (empros gmbh)
Verteiler	an alle interessierten Kreise

Dokumentname	Test getriebene Entwicklung (041009).doc
Datum	09. Oktober 2004
Überarbeitung von	Version vom 19. September 2004

Quellennachweis

- Siehe Ressourcen im Anhang

Inhaltsverzeichnis

EINFÜHRUNG **4**

EIN KURZER BLICK AUF DIE MERKMALE GUTER OO-ENTWÜRFE **8**

BEHANDELTE THEMEN 8
ARCHITEKTUR 8
INTELLEKTUELLE BEHERRSCHBARKEIT 8
LOSE KOPPELUNG: AUF DER SUCHE NACH ORTHOGONALITÄT 9
GUTER OO-ENTWURF UND TESTBARKEIT 10

DESIGNGRUNDSÄTZE **11**

BEHANDELTE THEMEN 11
DAS OPEN/CLOSED-PRINZIP 11
THE LAW OF DEMETER 12
SCHNITTSTELLEN-BASIERTER ENTWURF 13
ENTWURFSMUSTER 14
PROGRAMMING BY INTENTION 15
DESIGN BY CONTRACT 15

UML **17**

BEHANDELTE THEMEN 17
DIE UML – EIN SCHRITT IN DIE RICHTIGE RICHTUNG 17
DIE UML IST NICHT AUSREICHEND 17
DIE UML IST ZU KOMPLEX 17
DER EINSATZ VON UML TOOLS ALLGEMEIN 18
DER EINSATZ VON UML TOOLS ZUR CODEERZEUGUNG 18
SOFTWARE ENTWURF: VON DER UML ZUM CODE 19
UML ERFOLGREICH NUTZEN 19

REFACTORING **20**

BEHANDELTE THEMEN 20

DOKUMENTATION **21**

BEHANDELTE THEMEN 21
INTERNE DOKUMENTATION 22
Detailspezifikation und werkzeug-gestützte Codeerzeugung *22*
Detailspezifikation und werkzeug-gestützte Erzeugung von Dokumentation *23*
EXTERNE DOKUMENTATION 26

WAS WIRD WIE GETESTET **27**

BEHANDELTE THEMEN 27
UNIT TESTING (MODULTEST) 27
INTEGRATIONSTESTS 27
VALIDIERUNG 28
RESSOURCENVERBRAUCH, FEHLERSITUATIONEN UND WIEDERHERSTELLUNG 28
PERFOMANCE-TESTS 28
USABILITY-TESTS (TESTS AUF BEDIENERFREUNDLICHKEIT) 29

WIE WIRD GETESTET? **30**

BEHANDELTE THEMEN 30
REGRESSIONSTESTS 30
TESTDATEN 30
Echte Daten *30*
Synthetische Daten *31*
GUI-TESTS 31
TESTS TESTEN: TESTSABOTEURE 31
SORGFÄLTIG TESTEN: TESTABDECKUNG 32
TEST DRIVEN DEVELOPMENT (TDD) – DER „TEST- FIRST“ ANSATZ 33
TESTEN, WAS SCHIEF GEHEN KANN 33
VERTRAGSORIENTIERT TESTEN 34

WANN WIRD GETESTET? **35**

BEHANDELTE THEMEN 35
TESTGELEGENHEITEN 35
Gelegenheit 1 ..... *35*
Gelegenheit 2 ..... *36*
Gelegenheit 3 ..... *36*
Gelegenheit 4 ..... *36*
Gelegenheit 5 ..... *36*
Gelegenheit 6 ..... *36*

TESTSTRATEGIEN **38**

BEHANDELTE THEMEN 38
DUMMY, ATTRAPPEN UND MOCKS 38
Mock Object *38*
Self Shunt *39*
Log String *39*
Crash Test Dummy *39*

TESTFAUSTREGELN **40**

C++ SPEZIFIKA **41**

Der idiomatische Gebrauch von C++ *41*
Compilerwarnungen nicht ignorieren *41*
Vererbung *41*
Design By Contract (DBC) in C++ *42*

INDEX **43**

RESSOURCEN **44**

Einführung

Testen ist eine Notwendigkeit. Aber: Testen ist mehr als ein notwendiges Übel, das wir möglichst schnell hinter uns bringen sollten. Testen ist ein wichtiger Bestandteil erfolgreicher Softwareentwicklung. Softwareentwicklung ist eine anspruchsvolle und komplexe Tätigkeit. Die Zahl der existierenden Methoden, Techniken und Werkzeugen ist erschreckend gross. Nicht geringer oder weniger erschreckend aber auch die Zahl der Softwarevorhaben, die scheitern oder mit massiven Zeit- und Budgetüberschreitungen beendet werden. Es scheint, wir haben nicht einen Mangel an Methode und Techniken zu beklagen. Vielmehr fehlt es in Projekten häufig am zielgerichteten und besonnenen Einsatz der geeigneten Verfahren, die sich in der Praxis bewährt haben. Das vorliegende Dokument fasst eine Reihe von Ideen und Konzepten rund um das Thema test-getriebene objektorientierte Softwareentwicklung zusammen. Dabei konzentrieren wir uns auf die vielfältigen Aspekte des Entwurfs von der Architektur, über den Grob- bis zum Detailentwurf. Wir werfen einen Blick auf die Qualitätsmerkmale guter Entwürfe und beleuchten eine Reihe von bewährten Techniken, die uns helfen, diese Qualitätsmerkmale zu erreichen.

Die sogenannte testgetriebene Entwicklung (Test Driven Development, kurz TDD) ist das zentrale Thema der vorliegenden Arbeit. Testgetriebene Entwicklung stellt automatisierte Unit-Tests an den Anfang der Programmiertätigkeit. Das Ziel dieser Art der Entwicklung ist „sauberer Code, der funktioniert“, um es in den Worten von Ron Jeffries zu formulieren [Jeffries]. Diese Art der Softwareentwicklung bildet ein tragendes Element innerhalb von Extreme Programming (XP) und stellt den Code als Ziel der Entwicklertätigkeit in den Vordergrund. Diese Fokussierung auf den ausführbaren Code erhöht die Sensibilität für die Wichtigkeit des Programmcodes. XP bewahrt uns damit vor der fatalen Tendenz, Programmierung als mechanische und intellektuell wenig anspruchsvolle Tätigkeit zu sehen. Wer den Softwarearchitekten höher einschätzt als den Programmierer, hat das Wesen erfolgreicher Softwareentwicklung nicht verstanden. Gefragt sind Talent und Erfahrung, wenn es um den Entwurf von soliden und einfach verständlichen Architekturen geht – zudem die Fähigkeit, angemessene Abstraktionen für ein Problem zu finden. Dabei handelt es sich um eine kreative und intellektuell anspruchsvolle Tätigkeit. Interessanterweise unterscheidet sich die Tätigkeit des Programmierers in dieser Hinsicht nicht im geringsten von derjenigen des Architekten. Auch die Aufgabe des Programmierers besteht darin, für ein bestimmtes Problem eine angemessene Abstraktion zu finden und diese mit Hilfe einer Programmiersprache solide und einfach verständlich umzusetzen. Architekt und Programmierer: beide entwerfen Modelle. Der wesentliche Unterschied ergibt sich aus dem Detaillierungsgrad der ersonnenen Abstraktionen und dem Grad an Unsicherheit bezüglich ihrer Korrektheit. Ein Architekturmodell entzieht sich aufgrund seines hohen Abstraktionsgrads der direkten Überprüfung. Demgegenüber handelt es sich beim Quellcode um ein ausführbares Modell, dessen korrekte Funktion zumindest getestet werden kann. Idealerweise bildet der Quellcode als Entwurf im Kleinen die Fortsetzung der Architektur als Entwurf im Grossen, und ermöglicht, die indirekte Überprüfung der Architektur. Architektur- und Programmentwurf liegen somit wesentlich näher beieinander als vielen bewusst ist.

Eine Schwierigkeit beim Entwerfen von Softwaresystemen besteht aufgrund der Unsicherheiten bezüglich der Abgrenzung der unterschiedlichen Stufen des Entwurfs. von der Architektur über den Grobentwurf hin zum Detailentwurf. Insbesondere stellen sich regelmässig die folgenden Fragen:

- Wie detailliert sollen die mit UML erstellten Modelle sinnvollerweise sein?
- Wie wird die Konsistenz der unterschiedlichen Modelle der jeweiligen Designstufen gewahrt?

- In welcher Form und mit welchem Detaillierungsgrad werden die jeweiligen Entwürfe dokumentiert?
- Was zeichnet guten Programmcode aus?
- Welche Aspekte des Programmcodes müssen dokumentiert werden?
- Wie wird die Konsistenz von Programmcode und Dokumentation gewährleistet?
- Welche Teile der erstellten Detailentwürfe sind wann wie zu testen?

Auf diese und weitere Fragen versucht der vorliegende Leitfaden mögliche Antworten zu geben. Die hier vorgestellten Konzepte und Ideen basieren auf einem pragmatischen Verständnis von Softwareentwicklung, die zum Ziel hat, funktionstüchtige und änderungsfreundliche Software zu liefern. Wir haben es bereits erwähnt, und der Titel dieser Arbeit unterstreicht dies: Das Thema Testen spielt dabei eine wichtige Rolle, insbesondere der frühe und konsequente Einsatz von automatisierten Unit-Tests, da diese die Basis für sämtliche weiteren Testaktivitäten sind. Zudem fällt ihre Erstellung und Pflege klar in den Verantwortungsbereich der Entwickler. Daneben existieren natürlich noch viele weitere Arten von Tests, Testmethoden und Testwerkzeugen, die aber für den testorientiert arbeitenden Entwickler erst in zweiter Linie von Interesse sind. So werden z.B. Akzeptanz- und Systemtests je nach Projektgrösse und Organisationsstruktur von eigens dafür vorgesehen Testteams erstellt und durchgeführt.

Die meisten Entwickler hassen Tests. Sie testen überhaupt nicht oder nur sehr nachlässig und umschiffen dabei unbewusst die kritischen Stellen ihres Codes, von denen sie wissen, dass es Probleme gibt. Pragmatische Programmierer verhalten sich anders: Wir suchen aktiv und ruchlos nach unseren Fehlern, auf dass sie nicht von anderen gefunden werden müssen. Hunt & Thomas [Hunt&Thomas] vergleichen die Fehlersuche mit dem Fischen mit einem Netz: Wir verwenden feine kleine Netze (die Unit-Tests) für den Fang der kleinen Fische und grosse grobe Netze (Integrationstests), um die Haie zu fangen. Manchmal schaffen es ein paar Fische zu entkommen. Daher stopfen wir alle Löcher, die wir entdecken, damit uns mehr und mehr der glitschigen „Fehlerfische“ aus unserem Projektsee ins Netz gehen.

Je früher uns ein Test eine schlechte Nachricht überbringt, desto nützlicher ist er. Daher beginnen wir zu testen, sobald wir Programmcode haben, so dass wir unsere Fehler erwischen solange sie noch Jungfische sind. Die kleinen Fische haben nämlich die unangenehme Eigenschaft, sehr schnell zu menschenfressenden Haien heran zu wachsen. Und einen Hai zu fangen, ist wesentlich schwieriger. Glücklicherweise können wir all die Testerei sehr gut automatisieren, so dass uns unnötige Handarbeit erspart bleibt. Frühes und häufiges Testen ist wichtig. Die besondere Herausforderung besteht darin, unsere Software so zu entwerfen, dass sie einfach getestet werden kann. Es besteht ein direkter Zusammenhang zwischen gut entworfenen objektorientierten Systemen und ihrer Testbarkeit. Von Bedeutung sind dabei Aspekte wie Einfachheit, Verständlichkeit und Änderungsfreundlichkeit. Auch davon handelt die vorliegende Arbeit.

Als Einstimmung auf die vielfältigen Themen, hier eine Sammlung von Gedanken, die für die vorliegende Arbeit Pate gestanden sind:

- Jedes Softwaresystem verfügt über eine Architektur. Entscheidend ist die Frage, ob die Architektur ein Zufallsprodukt oder das Ergebnis gezielter intellektueller Bemühungen ist.

- Die Architektur gut entworfener Softwaresysteme ist explizit, wohl strukturiert und zielgerichtet.

- Eine solide und einfache Architektur bildet die Basis für objektorientierte Entwürfe, die diesen Namen auch verdienen.
- Guter objektorientierter Entwurf erhöht die intellektuelle Beherrschbarkeit und damit die Änderungsfreundlichkeit unserer Software.
- Gute objektorientierte Entwürfe weisen eine ausgezeichnete Testbarkeit auf.
- Gute objektorientierte Entwürfe sind schlank und weisen weder doppelte noch unnötige Elemente auf.
- Gute objektorientierte Entwürfe sind so einfach wie möglich, aber nicht einfacher.
- Gute objektorientierte Entwürfe können einfach angepasst und erweitert werden. Sie ermöglichen organische Software, die form- und veränderbar ist.
- Gute objektorientierte Entwürfe bestehen aus untereinander lose gekoppelten und isoliert getesteten Objekten.
- Die UML ist unausgewogen, da sie einerseits zu komplex und andererseits zu wenig ausdrucksstark ist, um alle relevanten Aspekte komplexer Softwaresysteme zu erfassen. Sie eignet sich als Kernelement der konzeptionellen Modellierung nach dem Grundsatz: Weniger ist mehr.
- UML schlägt die Brücke zwischen der sehr informellen und meist zu unpräzisen natürlichen Sprache auf der einen Seite und den i.d.R. zu detaillierten Anweisungen einer Programmiersprache auf der anderen.
- Der Einsatz der UML ist dann am effizientesten und effektivsten, wenn sie in dieser Brückenfunktion verwendet wird. Die UML eignet sich sehr gut, um die grundlegenden Aspekte eines Entwurfs zu illustrieren. Der Versuch, zu viele Details mit Hilfe der UML einzufangen, führt zu ungewollter Komplexität. Zudem erhöht sich dadurch der Aufwand, um die detaillierten UML Diagramme mit dem verfeinerten - und letztlich einzig gültigen - Detailentwurf in Form von Programmcode in Einklang zu bringen. Der Begriff „Round Trip Engineering“ ist ein gefährlicher Euphemismus für ein Problem, das wertvolle Ressourcen bei geringem Gegenwert verschlingt.
- Gute objektorientierte Entwürfe sind ausreichend und korrekt dokumentiert. Dabei steht Korrektheit der Dokumentation vor Vollständigkeit.
- Je schlanker die Dokumentation ausfällt, um so grösser ist die Chance, dass sie aktuell gehalten wird.
- Zu wissen, welche Aspekte zu dokumentieren sind, ist genau so wichtig, wie zu wissen, welche Dinge nicht dokumentiert werden sollen. Das gilt auf allen Ebenen der Systemmodellierung, aber insbesondere für den Programmcode.
- Das Erstellen von Programmcode ist Entwerfen im Detail. Die Programmiertätigkeit ist das Erstellen eines detaillierten und ausführbaren Modells - Code als Design.
- Guter Programmcode erklärt, was gemacht wird und verbirgt das *Wie* solange wie möglich. Guter

Programmcode ist damit weitgehend selbsterklärend; Kommentare dienen lediglich der Ergänzung.

- Guter Programmcode ist die Prosa der Software-Entwickler, schwierig zu schreiben und hoffentlich erhellend und ein Genuss zu lesen.

- Die Dokumentation der einzelnen mit Hilfe von Programmcode modellierten Komponenten und ihr Zusammenspiel untereinander besteht aus den Schnittstellen der jeweiligen Komponenten und den dazugehörigen Testfällen. Diese Testfälle erfüllen dabei zwei Aufgaben: Sie dokumentieren einerseits die korrekte (vertragskonforme) Verwendung der jeweils getesteten Komponenten und gewährleisten andererseits, dass die getesteten Komponenten gemäss ihrer Spezifikation funktionieren.

- Test-getriebene Entwicklung zwingt zu einfachen und durchgängig objektorientierten Entwürfen.

- Testbarkeit bedingt Entwürfe aus untereinander lose gekoppelten Objekten mit enger Kohäsion und klaren Schnittstellen.

- Der Grad der Testbarkeit in einem objektorientierten Entwurf kann als Mass für die Güte des Entwurfs gesehen werden.

- Änderungsfreundlichkeit bedingt Einfachheit.

- Einfachheit erhöht die intellektuelle Beherrschbarkeit und fördert die Testbarkeit.

- Testbarkeit erhöht die intellektuelle Beherrschbarkeit.

- Intellektuelle Beherrschbarkeit bewahrt die Einfachheit, da nachträgliche Änderungen nicht ungewollte Komplexität aufgrund von Unverständnis einführen.

- Intellektuelle Beherrschbarkeit fördert die Änderungsfreundlichkeit.

Ein kurzer Blick auf die Merkmale guter OO-Entwürfe

Behandelte Themen

- Architektur
- Intellektuelle Beherrschbarkeit
- Welches sind die wichtigsten Aspekte guter Entwürfe?
- Welcher Zusammenhang besteht zwischen einem guten OO-Entwurf und seiner Testbarkeit?

Architektur

Jedes System verfügt über eine Architektur. Gute Systeme verfügen über eine explizite, bewusst gestaltete Architektur mit dem Ziel, die wesentlichen Komponenten mit ihren Schnittstellen und Beziehungen untereinander griffig darzustellen. Dabei beschreibt die Architektur sowohl dynamische als auch statische Aspekte. Sie ist damit Bauplan und Ablaufplan für die Software in einem.

Architektur schafft Ordnung: Sie dokumentiert angemessen und problembezogen die Struktur und das Zusammenwirken der einzelnen Komponenten. Die Architektur stellt das Verständnis des Systems sicher. Sie bildet die Grundlage für änderungsfreundliche Systeme – sie ist ein Rahmen für geordnete Veränderung.

Architektur ist Reduktion auf das Wesentliche. Eine gute Architektur lenkt die Wahrnehmung des Betrachters auf die wesentlichen Aspekte des Systems, indem sie unnötige Details weglässt. Die Grenze zwischen Architektur und dem darauf aufbauenden Entwurf ist fliessend.

Intellektuelle Beherrschbarkeit

Gute Entwickler sind bescheiden! Sie anerkennen die Begrenztheit des menschlichen Verstandes und tun alles, damit ihre Entwürfe überschaubar und nachvollziehbar bleiben. Kurz, sie sind bemüht, die intellektuelle Beherrschbarkeit einer Lösung zu gewährleisten. Erfahrene Entwickler verwechseln dabei geniale Entwürfe nie mit komplizierten Lösungen.

Nur wenn wir ein System verstehen, können wir auch gewährleisten, dass es den gestellten Anforderungen genügt. Ein zu komplexes oder gar kompliziertes System schmälert die Chancen auf hohe Qualität. Das hat damit zu tun, dass Anforderungen ändern und somit auch das System angepasst werden muss. Ist das System zu komplex, so ist es oft nur mit sehr grossem Aufwand möglich, Anpassungen geordnet vorzunehmen. Geordnet meint dabei in einer Art und Weise, die die Stabilität des Systems nicht gefährdet. Ausserdem erhöht sich in einem komplexen System die Gefahr, dass wir Anforderungen fehlerhaft oder ungenügend abbilden, ohne es zu bemerken. Das ist übrigens einer der Gründe, warum frühes und häufiges Testen so wichtig ist.

Lose Koppelung: Auf der Suche nach Orthogonalität

Der Begriff Orthogonalität bedeutet in der Informatik: Unabhängigkeit und Entkoppelung. Zwei oder mehr Dinge sind orthogonal, wenn Veränderungen an einem keines der anderen beeinflusst. In einer Architektur verwenden wir z.B. Schichtenmodelle, um auf der Gesamtsystemebene Orthogonalität zu erreichen. Unser Ziel ist es, Komponenten zu entwerfen, die in sich abgeschlossen sind – man spricht hier auch von Kohäsion. Wenn Komponenten voneinander isoliert sind, dann kann man sie unabhängig voneinander ändern. Und solange man die öffentliche Schnittstelle einer Komponente nicht ändert, vermeiden wir auch Probleme aufgrund von Änderungen, die sich über das gesamte System fortpflanzen. Lose Koppelung (Orthogonalität) und Kohäsion sind also eine gute Sache – wir können damit unsere Produktivität steigern und wir minimieren das Risiko. Dies auch deshalb, weil wir orthogonale Systeme wesentlich leichter testen können.

Leider besteht immer, wenn Sie Programmcode schreiben, die Gefahr, dass Sie die Orthogonalität in Ihrem System verringern. Darum lohnt es sich, die Aspekte Koppelung und Kohäsion stets im Auge zu behalten. Das Beachten der folgende Punkte hilft Ihnen, die Orthogonalität zu bewahren [Hunt&Thomas]:

- **Halten Sie Ihren Quelltext entkoppelt.** Schreiben Sie schüchternen Code. Die Einhaltung des Open/Closed-Prinzips und des Gesetzes von Demeter kann Ihnen hier helfen.

- **Vermeiden Sie globale Daten.** Sie wissen warum! Die objektorientierte Variante der globalen Daten ist übrigens die unbedachte Verwendung des Singleton-Musters. Seien Sie vorsichtig, und setzen Sie Singeltons sehr sparsam ein. Bedenken Sie auch: Das Testen von Komponenten, die auf ein Singelton-Objekt zugreifen müssen, ist schwierig bis unmöglich. Da es von jeder Singleton Klasse nur eine Instanz gibt, auf deren Erzeugung der Verwender i.d.R. keinen Einfluss hat, stellt sich das Problem der Austauschbarkeit. Wenn es sich beim Singleton um eine kostspielige Ressource handelt – und das ist gerade bei einem Singelton häufig der Fall – dann entsteht rasch der Wunsch, diese kritische Ressource für das isolierte Unit-Testing durch eine leichtgewichtige Attrappe zu ersetzen. Doch gerade das verhindert das Singleton-Muster, da wir von Aussen keinen Einfluss auf die Erzeugung des Singleton nehmen können. Die einfachste Lösung besteht darin, einerseits den Einsatz des Singleton-Musters in einem System zu begrenzen und andererseits die Singelton-Erzeugung konsequent über strategiebasierte Fabrikmethoden zu regeln. Damit wird es einfach möglich, verschiedene Erzeugungsstrategien je nach Bedarf zu verwenden.

- **Vermeiden Sie ähnliche Funktionen.** Wiederholter Programmcode ist ein Symptom für strukturelle Probleme in Ihrem Entwurf. Wenn Sie z.B. immer wieder die gleichen Anweisungen zu Beginn und am Ende einer Sequenz von Arbeitsschritten haben und nur dazwischen je unterschiedliche Befehle, dann schauen Sie sich doch mal das Template-Method oder Strategy-Muster an.

Behalten Sie Ihren Programmcode ständig im Auge. Seien Sie wachsam und reagieren Sie früh aber beherzt, wenn der Code sich widersetzen beginnt [Code-Druck]. Nutzen Sie jede Gelegenheit, den Code besser zu strukturieren, Fehler zu beheben und Unklarheiten zu beseitigen. So können Sie verhindern, dass Ihr Programm schleichend zu einem Softwareslum wird.

Guter OO-Entwurf und Testbarkeit

Ein orthogonales, d.h. auf lose Koppelung und enge Kohäsion ausgerichtetes – System ist einfacher zu testen. Sie können nämlich einen Grossteil der Systemtests auf der Ebene der einzelnen Module durchführen, da die Interaktionen zwischen den Systemteilen begrenzt sind. Das ist ein grosser Vorteil, denn Unit-Tests sind wesentlich einfacher zu schreiben und auszuführen als Integrationstests.

Interessanterweise bildet das Schreiben der Unit-Tests selbst einen guten Test für die Orthogonalität in Ihrem System. Was benötigen Sie alles, um einen Unit-Test durchführen zu können – besteht Ihr System den 60-Sekundentest [60Sekunden]? Wenn Sie für einen Unit-Test das halbe System compilieren und linken müssen, dann haben Sie definitiv ein Problem, das es zu Lösen gilt: Sie haben ein Modul gefunden, das vom übrigen Teil des Systems schlecht entkoppelt ist. Genau dieser Aspekt des Unit-Testing macht den Test-First Ansatz so attraktiv: Je früher wir uns über die Testbarkeit Gedanken machen, desto eher werden wir feststellen, ob unser System orthogonal entworfen ist. Daher lohnt es sich die testgetriebene Entwicklung gerade – aber nicht nur – für Entwickler mit nur geringer OO-Erfahrung, um rasch das nötige Entwicklergespür für guten OO-Entwurf zu entwickeln.

Achten Sie von Beginn an auf die Testbarkeit Ihrer Entwürfe: Wenn Sie ein Modul oder auch nur eine einzelne Routine entwerfen, sollten Sie nicht nur dessen Schnittstelle entwerfen sondern den Code, der die Schnittstelle testet, gleich mit dazu. Sie werden feststellen, dass sich die Zuverlässigkeit Ihres Codes markant verbessert und Sie viele Fehler beheben, indem Sie diese erst gar nicht begehen.

Designgrundsätze

Behandelte Themen

- Worin besteht der Unterschied zwischen Typen und Klassen?
- Worin bestehen die Vorteile einer Schnittstellen-orientierten Sichtweise?
- Welcher Zusammenhang besteht zwischen dem Schnittestellen-basierten Entwurf und dem Einsatz von Entwursmustern?
- Welche Punkte sind beim Einsatz von Entwurfsmustern grundsätzlich zu beachten?
- Warum ist bei der Verwendung des Singleton-Musters Vorsicht geboten?
- Was ist Programming by Intention und welche Vorteile hat diese Art der Programmierung?
- Was ist Design By Contract und Vorteile bietet diese Art der Programmierung?
- Welche Auswirkungen hat dies auf die zu testenden Software?
- Was ist das Gesetz von Demeter und welche Vorteile ergeben sich aus seiner Verwendung?

Das Open/Closed-Prinzip

Dieses Prinzip besagt: *Module sollten für Änderungen geschlossen und für Erweiterungen offen sein.* Es berührt eine Reihe von fundamentalen Aspekten der Objektorientierung, namentlich Vererbung, Polymorphismus und Kapselung. Wenn eine Änderung in einer Software zu einer wahren Kaskade an weiteren Änderungen führt, dann zeigt solches System alle Zeichen, die wir allgemein mit schlechtem Entwurf in Verbindung bringen: Die Software ist starr und alles andere als formbar, sie ist anfällig für ungewollte Seiteneffekte und wirkt zerbrechlich - die kleinste Anpassung erzeugt in diesem starren Gebilde soviel Spannung, dass es zu zerbrechen droht.

Das "Open-Closed"-Prinzip hilft uns beim Entwerfen von änderungsfreundlicher Software. Es fordert, dass wir Module entwerfen, die sich *niemals* ändern. Änderungen in den Anforderungen werden abgedeckt, indem bestehendes Verhalten erweitert wird.

Klassen, die dem "Open-Closed"-Prinzip gehorchen, verfügen über zwei wesentliche Eigenschaften:

- Sie sind *offen* gegenüber Erweiterungen. Das heisst, das Verhalten solcher Klasse kann erweitert werden, um neue Anforderungen einer bestehenden oder gar neuen Anwendung zu erfüllen
- Sie sind *geschlossen* gegenüber nachträglicher Veränderung. Das heisst, dass sie nicht mehr modifiziert werden müssen, um als Basis für neue Anforderungen dienen zu können.

Auf den ersten Blick scheinen die Forderungen dieses Prinzips nach Offenheit *und* Geschlossenheit ein Widerspruch zu sein. Wie können wir ein Modul so entwerfen, dass es erweitert werden kann,

ohne es zu verändern? Der Zauberspruch, der diesen vermeintlichen Widerspruch auflöst lautet: „Abstraktion und Polymorphie“. Hierzu existiert eine Reihe von Faustregeln, deren Beachtung die Einhaltung des Open/Closed-Prinzips erleichtert:

> ***Faustregeln***
>
> *Explizite Fallunterscheidung aufgrund des Typs eines Objektes ist in der Regel ein Fehler. Der Designer sollte statt dessen Polymorphismus verwenden.*
>
> *Explizite Fallunterscheidung der Werte eines Attributs ist meist ein Fehler. Die Klasse sollte in eine Vererbungshierarchie zerlegt werden, wobei jeder mögliche Wertinhalt des Attributs in eine abgeleitete Klasse umgewandelt wird.*
>
> *Siedle gemeinsame Daten und Verhaltensweisen und/oder die Schnittstelle einer Klasse in der Vererbungshierarchie so weit oben wie nur möglich an.*
>
> *Wenn zwei Klassen nur über gemeinsame Daten verfügen, jedoch keine gemeinsamen Verhaltensweisen aufweisen, dann sollten diese gemeinsamen Daten in einer Klasse untergebracht werden, die von den beiden Klassen aggregiert wird.*
>
> *Wenn zwei oder mehr Klassen über gemeinsame Daten und Verhaltensweisen (d.h. Methoden) verfügen, sollte jede dieser Klassen von einer gemeinsamen Basisklasse erben, welche die gemeinsamen Daten und Methoden umfasst.*

Die Testfälle für ein Modul, welches unter Beachtung des Open/Closed-Prinzips entworfen worden sind, weisen eine hohe Änderungstabilität auf. Das hat den positiven Effekt, dass wir uns bei Erweiterungen auch bei den Testfällen voll und ganz auf die Erweiterungen konzentrieren können.

The Law of Demeter

Das Gesetz von Demeter für Funktionen hat zum Ziel, die Koppelung zwischen Objekten zu verkleinern, indem es versucht, Sie daran zu hindern, in ein Objekt hineinzugreifen, um Zugriff auf ein drittes Objekt zu erhalten.

Das *Gesetz von Demeter* hilft uns, schüchternen Code zu schreiben. Schüchterner Code verrät möglichst wenig von sich und interagiert mit möglichst wenig anderen Codeelementen.

Gemäss dem Gesetzt von Demeter sollte eine Methode eines Objekts nur Methoden aufrufen von:

- sich selbst
- Parametern der Methode
- in der Methode erzeugten Objekten
- Komponenten-Objekten.

```
class Demeter {

  public:
    void exampleMethod(const B& b) {
      C c;
      bool d = isOk(); // eigene Methode
      b.open();        // Methode auf Parameter
      c.read();        // Methode auf erzeugtem Objekt
```

```
      _a.print();       // Methode auf Komponenten Objekt
    };
  private:
    bool isOk() { return true; };
    A _a;

};
```

Schnittstellen-basierter Entwurf

Eine wichtige Entwurfsregel der Objektorientierung lautet: *Entwerfe auf eine Schnittstelle hin und nicht auf eine Klasse.* In einem objektorientierten System erfolgt der Zugriff auf ein Objekt über seine Operationen. Diese Operationen sind in der sogenannten öffentlichen *Schnittstelle* des Objektes beschrieben. Die Schnittstelle formt dabei einen vollständigen Satz an Anfragen (Botschaften), die an dieses Objekt gerichtet werden können.

Schnittstellen sind grundlegend für objektorientierte Systeme. Ein Objekt kann nur über seine öffentliche Schnittstelle angesprochen werden. Ohne Schnittstelle gibt es von Aussen keine Möglichkeit, etwas über ein Objekt heraus zu finden. Die Schnittstelle verbirgt dabei das Innere eines Objektes; sie verrät nichts über die konkrete Implementierung. Verschiedene Objekte können die gleichen Operationen einer Schnittstelle völlig unterschiedlich umsetzen.

Typen benennen und beschreiben Schnittstellen. Objekte unterstützen Schnittstellen. Die Definition eines Objekts ist in seiner Klasse beschrieben. Die Klasse ist als Bauplan der Ort, wo die Deklaration der Schnittstelle und die Implementation zu einem Ganzen zusammengefügt werden. Objekte werden von Klassen erzeugt, wobei eine strenge Beziehung zwischen einem Objekt und seiner Klasse besteht - die sog. Instantiierungsbeziehung. Typen und Klassen können spezialisiert werden, wobei das Ableiten von Typen und das Ableiten von Klassen nicht das Gleiche ist.

Die Klasse beschreibt, *wie* das Objekt implementiert ist. Im Gegensatz dazu bezieht sich der Typ lediglich auf die Schnittstelle, die beschreibt, *was* mit dem Objekt getan werden kann. Ein Objekt kann viele Typen haben, d.h. mehrere Schnittstellen unterstützen, und die Objekte verschiedener Klassen können den gleichen Typ besitzen.

Die Klasse legt ebenfalls einen Typ fest, denn sie beschreibt die Anfragen, die ein Objekt dieser Klasse verarbeiten kann. Aufgrund der Instantiierungsbeziehung zwischen einem Objekt und seiner Klasse ist stets auch impliziert, dass es die in seiner Klasse festgelegte Schnittstelle unterstützt.

Sprachen wie C++ und Eiffel unterscheiden nicht explizit zwischen Klassen und Objekttypen; beide werden mit Hilfe von Klassen spezifiziert. Sprachen wie Java oder C# kennen Schnittstellen als eigenes Sprachkonzept und erlauben damit eine Unterscheidung zwischen Typen und Klassen. In diesen Sprachen können wir Typen unabhängig von Klassen beschreiben und Klassen können in ihrer Deklaration die implementierten – und damit unterstützten - Typen explizit benennen. In C++ können wir Schnittstellen mit sog. Schnittstellen-Klassen nachbilden und die Implementierung mit Hilfe von (Mehrfach-)Vererbung ausdrücken (vgl. hierzu den Abschnitt *Vererbung* im Kapitel *C++* Spezifika).

Klassenvererbung legt die in einer Klasse definierte Implementierung mittels der Implementierung einer anderen Klasse fest. Es handelt sich damit um einen Mechanismus, um Code und Repräsentation wieder zu verwenden; daher kann man diese Form der Vererbung auch als

Implementierungsvererbung bezeichnen. Schnittstellenvererbung (oder Subtyping) beschreibt demgegenüber, wann ein Objekt anstelle eines anderen Objekts verwendet werden kann. Diese zwei Konzepte können sehr leicht verwechselt werden, denn viele Sprachen treffen diese Unterscheidung nicht explizit. So bedeutet Vererbung in C++ sowohl Schnittstellen- als auch Implementierungsvererbung.

Obwohl die meisten Programmiersprachen nicht zwischen Schnittstellen- und Implementierungsvererbung unterscheiden, tun dies Entwickler in der Praxis dennoch. Das bedeutet konkret, dass in robusten und verständlichen Entwürfen Klassenvererbung stets im Sinne von Schnittstellenvererbung eingesetzt wird. Die korrekte Verwendung von Vererbung basiert dabei auf dem Liskovschen Substitutionsprinzip, das einfach formuliert wie folgt lautet: *Objekte der abgeleiteten Klasse können stets an die Stelle von Objekten der Oberklasse treten.* Hierzu existiert auch eine entsprechende Faustregel: *Vererbung sollte ausschliesslich zur Modellierung von Spezialisierungs-Hierarchien verwendet werden.* Klienten können sich so auf die Schnittstelle einer Basisklasse beziehen, ohne die davon abgeleiteten Spezialisierungen kennen zu müssen. Daraus ergeben sich zwei Vorteile:

1. Klienten müssen nichts über die Typen der verwendeten Objekte wissen, solange diese Objekte die erwartete Schnittstelle erfüllen.

2. Klienten müssen nichts über die Klassen wissen, welche die Implementierung für die verwendeten Objekte liefern. Klienten wissen nur über die abstrakten Klassen Bescheid, welche die benötigte Schnittstelle deklarieren.

Damit reduzieren wir die Abhängigkeiten zwischen Teilsystemen und vereinfachen das Testen unserer Module. Je komplexer die jeweils miteinander interagierenden Objekt-Implementierungen sind, desto mehr sind wir nämlich darauf angewiesen, diese komplexen Implementierungen transparent auswechseln zu können. Nur so gelingt es uns mit vertretbarem Aufwand zu testen. Ohne Schnittstellen ist dies praktisch nicht zu erreichen. Vgl. hierzu auch die Abschnitte *Lose Koppelung: Auf der Suche nach Orthogonalität* im Kapitel *Ein kurzer Blick auf die Merkmale guter OO-Entwürfe* und *Dummy, Attrappen und Mocks* im Kapitel *Teststrategien*.

Entwurfsmuster

Entwurfsmuster erhöhen die Flexibilität eines Entwurfs und fördern die Kommunikation zwischen Entwicklern. Ihre Verwendung ist daher grundsätzlich zu begrüssen. Es ist jedoch zu beachten, dass der unüberlegte und übertriebene Einsatz von Mustern die Komplexität eines Entwurfs erhöht und damit die intellektuelle Beherrschbarkeit verringert.

Eines der wesentlichen Ziele von Entwurfmustern ist die Erhöhung der Änderungsfreundlichkeit auf der Basis von untereinander möglichst lose gekoppelten Komponenten und der Verwendung von Schnittstellen. Damit verbessern die meisten Entwurfsmuster auch die Testbarkeit von OO-Entwürfen - eine unrühmliche Ausnahme bildet das Singleton-Muster (vgl. hierzu den Abschnitt *Lose Koppelung: Auf der Suche nach Orthogonalität*).

Programming By Intention

Je mehr unser Programm sich wie die Lösung eines Problems aus der Fachdomäne liest, und je weniger es an eine Vorführung von Konstrukten der verwendeten Programmiersprache erinnert, desto besser ist es. Wir können die intellektuelle Beherrschbarkeit unserer Programme stark verbessern, wenn wir uns so lange wie möglich auf das *Was* konzentrieren und uns um das Wie so früh wie nötig aber so spät wie möglich kümmern. Diese Art der Programmierung wird auch als „Programming by Intention“ bezeichnet. Dieser Ansatz stellt die Absicht (*was getan wird*) in den Vordergrund. Dabei fordert und fördert diese Art der Programmierung den Entwurf vom Groben zum Detail.

Indem wir eine Routine einführen, welche die „Computer-Details“ verbirgt, behandeln wir das Problem auf einer abstrakteren Stufe. Dadurch drücken wir unsere Absichten klarer aus, und unser Programmcode wird verständlicher. Wir legen also eine Schicht von Routinen an, welche die Verarbeitung der Daten auf der Ebene der Programmiersprache verarbeiten Das restliche Programm ist damit in der Lage, das Problem auf einer abstrakteren und zugleich wirklichkeitsnäheren Stufe zu behandeln.

Wird diese Technik konsequent angewendet, so reduziert sich die Menge der für das Verständnis nötigen Kommentare in einem Programm beträchtlich.

Design By Contract

Dieses Konzept geht auf Betrand Meyer zurück und wird in der von ihm entwickelten objekt-orientierten Programmiersprache Eiffel vorbildlich unterstützt. Design By Contract – kurz DBC - fordert und fördert den verantwortungs-getriebenen Entwurf. Dabei werden die Interaktionen zwischen Objekten auf der Basis von Verträgen beschrieben. Es handelt sich im Grundsatz um eine Erweiterung des Schnittstellen-Gedankens.

Das Herzstück dieser Technik bildet die Zusicherung. Dabei handelt es sich um einen logischen Ausdruck, der niemals falsch werden darf. Das Scheitern einer Zusicherung ist ein Fehler und in Sprachen, die strukturiertes Exception-Handling ermöglichen, wird in einem solchen Fall eine Exception ausgelöst. Zusicherungen existieren in drei Formen: Vorbedingung, Nachbedingung und Invarianz.

Vor- und Nachbedingungen beziehen sich auf die Operationen eines Objekts. Mit Hilfe einer Vorbedingung lässt sich beschreiben, wie die Welt vor der Ausführung einer Methode aussehen soll. So könnte z.B. die Vorbedingung für den Übergabeparameter an eine Methode zur Berechnung einer Quadratwurzel der folgende Ausdruck sein: "eingabe >= 0". Damit wird ausgedrückt, dass es ein Fehler ist, die Funktion mit einem Parameter aufzurufen, der kleiner als Null ist. Die Nachbedingung beschreibt die Welt, wie sie nach der Ausführung einer Operation aussehen soll. So könnte die Nachbedingung für unsere Quadratwurzel-Methode wie folgt lauten: "eingabe = ergebnis * ergebnis". Nachbedingungen sind eine nützliche Art, auszudrücken, was geschehen soll, ohne zu sagen, wie dies passieren soll. Dies entspricht der gängigen Trennung von Schnittstelle und Implementation.

Die dritte Form der Zusicherung, die Invariante bezieht sich auf ein Objekt als Ganzes. Sie wird auf Klassenebene festgelegt und gilt immer für alle Instanzen dieser Klasse. Immer bedeutet hier: zu jedem Zeitpunkt, zu dem ein Objekt zur Ausführung von Methoden verfügbar ist. Das bedeutet, dass die Invariante zu allen Vor- und Nachbedingungen aller öffentlichen Methode einer Klasse hinzugefügt

wird. Die Invariante kann während der Ausführung der Methode falsch werden, sie muss jedoch am Ende der Methode wieder wahr sein.

Zusicherungen sind ein effektives Hilfsmittel zur Erstellung klarer Schnittstellen. Sie sollten daher direkt als Teil der Schnittstellendeklaration aufgenommen werden können. Leider unterstützt nur Eiffel Zusicherungen als Teil der Sprache. Da Zusicherungen normalerweise nur während der Entwicklung geprüft werden, empfiehlt es sich, die Prüfung von Zusicherungen für das Endprodukt zu deaktivieren.

UML

Behandelte Themen

- Der zielgerichtete Einsatz der UML im Entwurf
- Stärken und Schwächen der UML
- Die UML als Kommunikationshilfsmittel
- Detailgrad der erstellten UML Artefakte

Die UML – ein Schritt in die richtige Richtung

Die „Unified Modeling Language“ (UML) hat sich in den letzten Jahren als De-Facto Standard im Bereich der objektorientierten Softwareentwicklung durchgesetzt. Die UML definiert eine Standardnotation und die Semantik für die korrekte Anwendung dieser Notation für Softwaremodelle von Systemen, die auf der Basis objektorientierter bzw. komponentenorientierten Technologien gebaut werden. Die UML bietet hierzu eine allgemeine und konsistente Notation zur Beschreibung objektorientierter Systeme. Damit reduziert sie – zumindest theoretisch – die Lernkurve für Entwickler, da diese nur eine Notation erlernen müssen. Damit ist die UML klar ein Schritt in die richtige Richtung und erspart uns immerhin die früher tobenden „Notationskriege“. Aber die UML ist nicht perfekt. Daher ist es wichtig, die UML so einzusetzen, dass wir ihre Stärken nutzen und ihre Schwächen meiden.

Die UML ist nicht ausreichend

Die UML stellt eine Vielzahl von unterschiedlichen Modellen zur Verfügung. Jedes dieser Modelle hat seine spezifischen Vor- und Nachteile. Kein Modell ist für sich alleine aussagekräftig genug, um alle relevanten Aspekte einzufangen. Die UML ist recht robust, ihr fehlen aber bestimmte Modelltypen und die Anwendung bestimmter Modellelemente ist im besten Fall unklar. Versuchen Sie nicht, alles mit der UML abbilden zu wollen. Wenn Sie ein Datenmodell erstellen müssen, tun Sie dies am besten mit Hilfe eines ERM. Und zur Illustration bestimmter Abläufe eignen sich die guten alten Flussdiagramme oft besser als ein UML Aktivitätsdiagramm.

Die UML ist zu komplex

Wenn Sie sich die definierten Artefakte der UML ansehen, stellen Sie rasch fest, dass die Fülle der Notationen und Feinheiten viel grösser ist als Ihnen lieb sein kann. Wer die UML Referenz der drei Amigos genauer studiert, muss erkennen, dass selbst die Väter der UML Mühe haben, die vielfältigen Modellelemente und ihre jeweilige Semantik widerspruchsfrei darzulegen. Das sollte Ihnen eine deutliche Warnung sein. Beschränken Sie sich auf das Wesentliche und beherzigen Sie die „UML-Version“ des Pareto-Prinzips: „80% Ihrer Modellierungsbedürfnisse lassen sich mit 20% der verfügbaren Notation abdecken.“ Die UML ist wirklich sehr umfangreich und kann ziemlich einschüchtern. Die UML ist seltsam: Auf der einen Seite bietet sie zu wenig und andererseits enthält

sie schlicht zu viel.

Der Einsatz von UML Tools allgemein

Gegen den Einsatz von UML Tools ist grundsätzlich nichts einzuwenden. Bedenken Sie aber, dass alleine der Einsatz eines solchen Werkzeuges nicht automatisch zu einer Verbesserung der von Ihnen entwickelten Software führt. Sie müssen schon ziemlich gut über objektorientierte Entwurfstechniken, über die UML sowie ihre Stärken und Schwächen Bescheid wissen, um einen Nutzen aus solchen Werkzeugen zu ziehen. Der UML Werkzeugeinsatz eignet sich meist besser für ein systematisches Requirements Engineering als für das Erstellen von Entwürfen. Insbesondere die Verwendung von UML Werkzeugen zur Codegenerierung ist problematisch – zu viele Feinheiten des Detailentwurfs können mit der UML nicht mit vertretbarem Aufwand in der nötigen Klarheit modelliert werden.

Das Mantra des erfolgreichen Software-Entwicklers lautet: *Vereinfache, vereinfache, vereinfache*. Wir streben nach Einfachheit in unseren Entwürfen und beim Einsatz von Hilfsmitteln.

Die Vorteile einfacher Werkzeuge:

- Einfache Werkzeuge sind einfach zu verwenden.
- Einfache Werkzeuge vermitteln taktiles Feedback. Man kann sie anfassen und einfach manipulieren; Karteikarten können beispielsweise herumgereicht und auf dem Tische bewegt werden.
- Einfache Werkzeuge schaffen intellektuelle Freiräume. So bringt der Einsatz von Papier und Bleistift eine gewollte Verlangsamung in den kreativen Gestaltungsprozess der Modellierung. Diese Verlangsamung gibt dem Entstehen der Lösung, dem Aufwerfen von kritischen Fragen und damit dem Lösungsfindungsprozess insgesamt mehr Raum, um sich zu entfalten.
- Einfache Werkzeuge sind kostengünstig.

Der Einsatz von UML Tools zur Codeerzeugung

Der Einsatz von UML Tools zur Codeerzeugung ist in der Regel nicht ratsam. Um die erstellten UML-Modelle als direkte Basis für Code verwenden zu können, müssen diese Modelle einen hohen Detaillierungsgrad aufweisen. Nur so ist eine sinnvolle Transformation in Code überhaupt möglich. Diese Details betreffen spezifische Aspekte der tatsächlichen Umsetzung im Code und sind in der UML teilweise gar nicht oder nur mit hohem Aufwand zu beschreiben. Dies führt zu ungewollter Komplexität [Brooks], da die Modelle mit - für das grundsätzliche Verständnis unnötigen - Details überfrachtet werden und damit auch ihren Nutzen als Kommunikationshilfsmittel weitgehend einbüssen. Die beste Form von Implementierungsmodell ist Quellcode; er lässt sich ausführen und damit auch testen. Der Versuch, ausführbare Modelle mit Hilfe von UML zu beschreiben ist nicht ratsam.

Software Entwurf: Von der UML zum Code

Wie können wir also die UML in der Entwurfsphase sinnvoll einsetzen? Das Erstellen von Modellen mit der UML ist nicht Selbstzweck. Vielmehr modellieren wir,

- um zu verstehen und
- um zu kommunizieren.

Die UML eignet sich ideal, um grundlegende Entwurfsaspekte und -Ideen anschaulich zu illustrieren. Sie erleichtert die Kommunikation zwischen Entwicklern und erlaubt den raschen Informationsaustausch. Der Schwerpunkt liegt dabei eindeutig auf einer eher informellen Verwendung. D.h. es geht hier z.B. weniger darum, die Feinheiten zwischen einer Assoziation und einer allgemeinen Abhängigkeitsbeziehung UML-konform zu unterscheiden als vielmehr um die Tatsache, dass grundsätzlich eine Beziehung zwischen zwei Elementen besteht. Damit wird die UML zu einem Werkzeug unter vielen auf dem Weg zum eigentlichen Ziel: einer funktionstüchtigen Software. Kombinieren Sie die UML bedarfsgerecht mit anderen Hilfsmitteln wie CRC-Karten, Mindmaps, Datenmodellen, Prototypen etc. Ihr Ziel ist es, die Problemstellung und die damit verbundenen Risiken zu verstehen und auf der Grundlage dieses Verständnisses eine geeignete lauffähige Lösung zu implementieren.

Auf dieser Grundlage können wir auch die folgende Frage beantworten: Wie weit sollen Konzepte in UML modelliert werden? Modellieren Sie Ihre Entwürfe mit der UML:

- *solange es sich in Ordnung anfühlt.* Wenn Sie merken, dass sich die UML-Elemente zu sperren beginnen, wenn es mühselig wird, bestimmte Dinge auszudrücken, deren Beschreibung mit Programmcode oder anderen Techniken besser gelingen würde, dann ist es Zeit, die UML für das betroffene Element hinter sich zu lassen.

- *bis Sie eine klare Vorstellung davon haben, was Sie tun müssen, um eine geforderte Anforderung zu erfüllen.* Wenn Sie beim Betrachten der jeweiligen UML-Modelle in der Lage sind, Schätzungen auf der Basis von Tagen abzugeben, dann stehen die Chancen gut, dass Sie bereit sind, den Detailentwurf im Code in Angriff zu nehmen .

UML erfolgreich nutzen

Machen Sie die UML zu Ihrem Modellierungskern und setzen Sie auf eine eher informelle Anwendung der Notation. Die grösste Stärke der UML liegt in ihrer *Illustrationsfähigkeit*. Nutzen Sie diese zu Ihrem Vorteil und verzichten Sie darauf, jedes noch so kleine Detail mit Hilfe der UML notieren zu wollen. Denken Sie daran, dass Sie UML-Modelle erstellen, um zu verstehen und um verständlich zu machen. Kurz: Der Hauptgrund für die Nutzung der UML ist *Kommunikation*. Martin Fowler [Fowler2] hat dies sehr treffend ausgedrückt:

„Ich benutze UML, weil sie mir ermöglicht, bestimmte Konzepte klarer darzustellen, als es mit den Alternativen möglich wäre. Die natürliche Sprache ist zu unpräzise und wird bei komplexeren Konzepten schnell verwirrend. Code ist präzise, aber zu detailliert. So benutze ich die UML für einen bestimmten Grad an Präzision, ohne mich in Details zu verlieren. Das heisst nicht, ich vermeide Details. Statt dessen hebe ich mit der UML *wichtige* Details hervor.“

Refactoring

Behandelte Themen

Was ist Refactoring?

Welcher Zusammenhang besteht zwischen Refactoring und Unit-Testing?

Refactoring beschreibt die Tätigkeit, bestehende Detailentwürfe (Programmcode) geordnet zu verändern, zu vereinfachen oder architektonisch umzuarbeiten. Dabei ist ein wichtiger Aspekt des Refactorings, dass die bestehende Semantik der Programme nicht verändert wird. Ein häufiges Vorurteil und Missverständnis bezüglich Refactoring betrifft das Themen Architektur und Vorabentwurf. Häufig wird behauptet, Refactoring verleite zu sorglosem Drauflosprogrammieren, und führe dazu, dass der Entwickler sich nicht mehr um eine saubere Architektur und Entwuf bemühe. Aber Refactoring macht weder den Architektur- noch den Grobentwurf als Vorabtätigkeit zur Programmierung überflüssig. Es erlaubt vielmehr ein entspannteres Umgehen mit Entwürfen. Refactoring gibt dem Entwickler die Freiheit, seine Meinung zu ändern, ohne dabei die bestehende Funktionalität eines Systems zu zerbrechen. Konsequent angewendet, erhält und verbessert Refactoring die Qualität des Codes. Der Entwickler ist in der Lage, einfache Entwürfe zu wählen, die bei Bedarf erweitert oder angepasst werden können, um neue bzw. geänderte Anforderungen zu erfüllen. Damit trägt Refactoring dem iterativen und inkrementellen Vorgehen ideal Rechnung.

Refactoring sorgt dafür, dass unser Code - gleich einem gut angelegten und blühenden Garten - gedeihen und wachsen kann und unseren sich wandelnden Anforderungen gerecht wird. Frühes und häufiges Refactoring verhindert, dass unser Code verwildert. Dabei ist das konsequente automatische Unit-Testing eine unverzichtbare Vorbedingung.

Dokumentation

Behandelte Themen

- Agil Dokumentieren
- Automatisiert Dokumentieren
- Interne und externe Dokumentation
- Programmcode als Basis für die Systemdokumentation
- Selbsterklärender Code
- Kommentare im Programmcode
- Ausführbare Dokumente

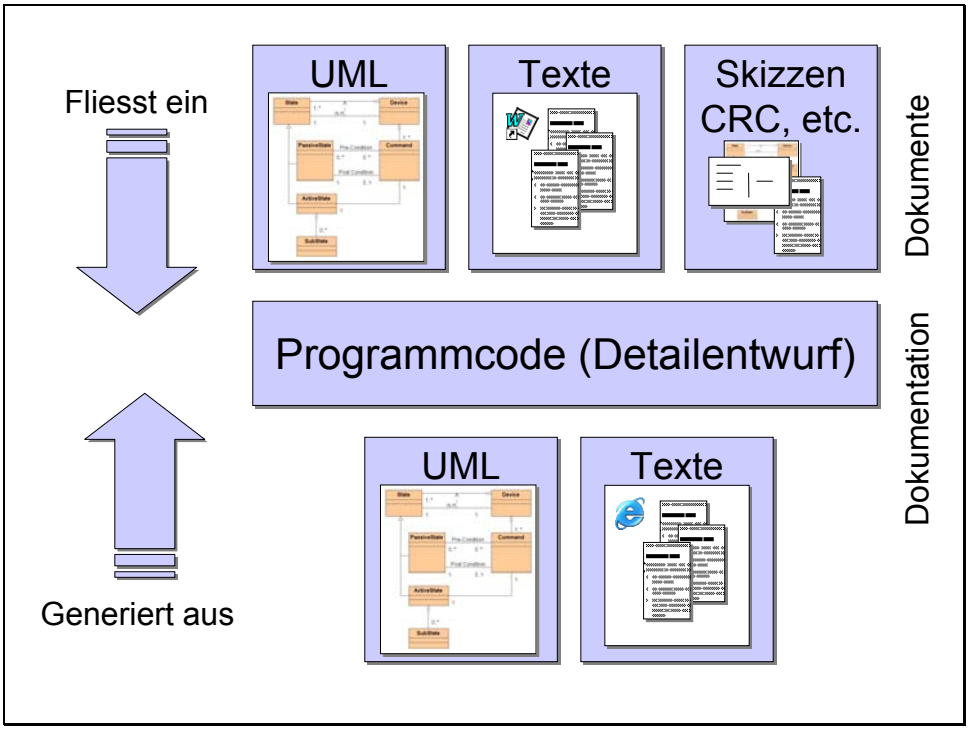
Das schreiben von Dokumenten gehört in der Regel nicht zu den beliebtesten Tätigkeiten eines Entwicklers. Meist wird diese Arbeit als Zeitverschwendung empfunden. Doch Dokumentation ist wichtig und nützlich. Wir können uns das Leben erleichtern, indem wir darauf achten, Dinge nicht mehrfach zu tun. Das würde nur Zeit verschwenden und die Gefahr erhöhen, dass die Dokumentation nicht aktuell ist.

Pflegen Sie eine agilen Dokumentationsstil. Manche Dokumente, die Sie erstellen, entwickeln sich im Verlauf eines Projekts zu Teilen der offiziellen Systemdokumentation. Die grosse Mehrzahl wird dagegen nicht überleben. Ihre Aufgabe ist es, die richtigen Dokumente mit dem richtigen Detailgrad zu erstellen. Die beste Dokumentation ist grundsätzlich die, die nicht geschrieben werden muss. Für die restlichen Dokumente gilt:

- Formulieren Sie präzise und beschränken Sie sich auf das Wesentliche. Meiden Sie den Hang zum Detail und schwammigen Formulierungen; reden Sie nicht um den heissen Brei. Sagen Sie, was Sie meinen, und verstehen Sie, was Sie sagen
- Automatisieren Sie das Erstellen der Dokumentation soweit als möglich. Das hilft Ihnen, die Arbeit nur einmal zu machen; Sie sparen Zeit und die Chancen steigen, dass Sie Ihre Dokumentation einfach aktuell halten können.
- Unterscheiden Sie zwischen interner und externer Dokumentation. Interne Dokumente umfassen die Kommentare im Programmcode, Architektur-, Design- und Testdokumente, etc. Die externe Dokumentation umfasst alle Dokumente, die nach aussen gehen, wie z.B. Benutzerhandbücher. Doch unabhängig davon, von wem und für wen die Dokumentation geschrieben ist, gilt es folgendes zu bedenken: Jede Dokumentation ist ein Spiegel des Programmcodes. Bei Abweichungen zwischen Code und Dokumentation zählt alleine der Code, und zwar unabhängig davon, ob der nun gut oder schlecht ist.

Interne Dokumentation

Im Verlaufe eines Projekts entstehen eine Vielzahl von Dokumenten in der Form von Texten, UML-Modellen, Skizzen, CRC-Karten, etc. Sie sind Hilfsmittel auf dem Weg zur Lösung. Sie dokumentieren dabei weniger das tatsächliche System als vielmehr das jeweils aktuelle Verständnis der Problemstellung und ihrer Lösungsvarianten. Diese Dokumente aufzubewahren hat daher mehr nostalgischen Wert als einen praktischen Nutzen. Sie wiegen sich in falscher Sicherheit, wenn Sie diese Dokumente für die *Dokumentation* Ihres Systems halten. Wenn Sie wissen möchten, wie ein bestimmtes Systemelement entworfen und umgesetzt ist, dann ist der entsprechende Programmcode Ihre einzig verlässliche Quelle. Das heisst nicht, dass Sie keine zusätzliche Dokumentation benötigen. Es bedeutet einfach, dass die Basis für Ihre Dokumentation in der Regel der Programmcode ist. Versuchen Sie – wenn immer möglich - die relevanten Dokumente auf der Basis des Programmcodes zu *generieren*.



Detailspezifikation und werkzeug-gestützte Codeerzeugung

Bevor wir näher auf die Möglichkeiten eingehen, wie wir die nötige Dokumentation weitgehend automatisch erstellen können, ein paar Gedanken zum umgekehrten Weg und der Frage: Können wir den Programmcode aus vorgängig erstellten Dokumenten erzeugen lassen? Dieses Vorgehen ist grundsätzlich denkbar. Voraussetzung dafür ist, dass es uns gelingt, das geforderte Systemverhalten mit den dafür vorgesehenen Hilfsmitteln - z.B. mit Hilfe der UML – griffig, detailgetreu und widerspruchsfrei zu spezifizieren. Nur dann können diese Entwürfe eindeutig und damit automatisch in Programmcode umgewandelt werden. Die Erfahrung der letzten Jahre hat gezeigt, dass die heute verfügbaren Methoden und Werkzeuge die dazu nötige Reife noch nicht erlangt haben; Die Methodenpäpste und Tool-Hersteller sehen das natürlich ganz anders. Doch zwischen den schönen Marketingversprechen und der wirklichen Welt der Softwareentwicklung klafft ein grosser Spalt, in den Sie – wenn Sie nicht vorsichtig sind – nur allzu leicht hinein fallen können, um für immer darin zu verschwinden.

Die UML spielt beim Entwerfen von Softwarelösungen eine wichtige Rolle. Sie ist aber ein Mittel zum Zweck und nicht der Zweck selbst. Es ist sehr aufwendig und mühselig mit Hilfe der UML, alle Entwurfsdetails so festzuhalten, dass daraus mit Hilfe eines Codegenerators verständlicher und effizienter Programmcode erzeugt werden kann; die UML ist für diese Art der Modellierung schlicht ungeeignet, da sie einerseits zu komplex und andererseits zu wenig ausdrucksstark ist. Der übertriebene Einsatz der UML führt zwangsläufig zu ungewollter Komplexität. Die mit Details überladenen UML-Modelle verlieren dabei einen ihrer entscheidenden Vorzüge: klare Kommunikation durch Illustration des Wesentlichen. Mehr zum sinnvollen Einsatz im Softwareentwurf erfahren sie im Abschnitt *UML*.

Detailspezifikation und werkzeug-gestützte Erzeugung von Dokumentation

Der Programmcode ist das detaillierteste Systemdokument überhaupt: Es beschreibt, illustriert und legt Zeugnis davon ab, ob und wie das System funktioniert. Daraus sinnvolle und brauchbare Dokumentation zu erstellen ist grundsätzlich nicht schwierig – vorausgesetzt, der Programmcode ist entsprechend entworfen und strukturiert.

Guter Programmcode ist nach dem Prinzip „Programming By Intention“ entworfen – er basiert auf klaren und leicht einzuhaltenden Codierrichtlinien, setzt Entwurfsmuster wohl dosiert ein und verwendet konsequent die Idiome der jeweiligen Programmiersprache; Guter Programmcode ist in weiten Teilen selbstdokumentierend. Testorientierte Entwicklung und regelmässiges Refactoring helfen dabei, Struktur und Qualität des Codes zu erhalten bzw. zu verbessern. Die einzelnen Module des Systems verfügen über klare vertragsbasierte Schnittstellen, ihre korrekte Funktion und ihre korrekte Verwendung sowie ihr Zusammenspiel untereinander, wird durch die vorhandenen Testfälle überprüft und beschrieben. Kritische Entwurfsentscheidungen und nicht offensichtliche Aspekte des Entwurfs werden mit Hilfe von entsprechenden Kommentaren herausgestellt. Kurz: guter Programmcode ist darauf ausgelegt, Struktur, Inhalt und Funktionsweise klar zu kommunizieren.

Guter Code erklärt sich selbst

Gute und sprechende Namen für Klassen, Methoden und Variablen tragen wesentlich zur Verständlichkeit bei. Das ist eigentlich eine Binsenweisheit – und dennoch: Noch immer erwecken viele Programme den Eindruck, bei ihrer Erstellung habe gerade Buchstabenknappheit geherrscht. Namen wie `foo`, `xtor`, `param1`, `doIt`, `stuff` etc. sind keine guten Bezeichner, denn sie kommunizieren ihren Zweck nicht. Vielleicht wählen Sie kurze Namen, weil Sie das Schreiben langer Bezeichner als mühsam empfinden. Doch bedenken Sie: Sie und andere werden den Code weit häufiger lesen als schreiben müssen. Darum: Schreiben sich doch einfach `connectionPool` anstatt `cp`. Übrigens ist die Verwendung der ungarischen Notation, um den Typ einer Variablen in ihren Namen einzubetten, in objektorientierten Systemen eine ziemlich schlechte Idee. Verzichten Sie also darauf, z.B. eine Zähler Variable vom Typ Integer `iIndex` zu nennen. Erstens sollte ein Name den abstrakten Zweck einer Variablen benennen und nicht erklären, wie sie implementiert ist. Und zweitens, wird im häufigen Fall, dass der Typ der Variable ändert, es genauso oft geschehen, dass der Name nicht entsprechend geändert wird. Damit hat die eingebettete Information nicht nur ihren Wert eingebüsst, sie ist vielmehr zu einer ärgerlichen Quelle der Verwirrung bezüglich des Typs geworden.

Noch schlimmer als bedeutungslose Namen sind irreführende. Eine Funktion wie `GetData`, die in Wirklichkeit Daten speichert, oder `printReport`, die nicht nur eine Auswertung erstellt sondern auch

noch die Datenbank aktualisiert, sind nur zwei Beispiele dieser scheusslichen Gattung. Namen sind für unser Gehirn von grosser Bedeutung, und irreführende Namen bringen das Chaos über Sie und Ihren Code.

Weitere Informationen zu diesem Thema finden Sie im Kapitel

Programming By Intention.

Kommentare im Code

Programmcode sollte Kommentare enthalten, aber zu viele Kommentare sind dabei genauso schädlich wie zu wenige. Gute Kommentare erläutern, *warum* etwas getan wird, sie erklären also Sinn und Zweck des Codes. Der Code selber zeigt, *wie* etwas gemacht wird. Das Erklären des Wie in Kommentaren ist redundant und verstösst gegen den Grundsatz: „Keine Wiederholungen“. Indem wir darauf achten, uns in den Kommentaren nicht zu wiederholen, verringern wir auch die Gefahr, dass Programmcode und die zugehörigen Kommentare sich widersprechen, ein weit verbreitetes Übel und Quelle grosser Verunsicherung.

Das Dokumentieren des Programmcodes ist die ideale Gelegenheit, all die kleinen schwer fassbaren Dinge fest zu halten, die sonst nirgends richtig hinein passen - dazu gehören: Gründe für bestimmte Entwurfsentscheidungen, Entwurfskompromisse, verworfene Alternativen, etc. Je nach Umfang ist es sinnvoll, diese Informationen in ein separates Dokument auszulagern und an der entsprechenden Stelle im Programm lediglich darauf zu verweisen. Bedenken Sie aber, dass sich damit die Gefahr erhöht, dass der Programmcode und die ausgelagerten Kommentare auseinander driften und damit mehr schaden als nützen.

Folgende Kommentare sind angebracht:

- Kurzer Modulkopf, Kommentare für wichtige Daten und Typendeklarationen, ein griffiger Kommentarkopf pro Klasse und je Methode, der festhält, was die Klasse bzw. die Methode tut, was bei ihrer Verwendung zu beachten ist und alle nicht-offensichtlichen aber für das Verständnis relevanten Informationen.

- Detailkommentare im Implementierungsteil von Methoden, um nicht offensichtliche Aspekte des Codes zu illustrieren. Halten Sie sich auch hier an die Regel, das Warum und nicht das Wie zu erklären. Konsequentes „Programmin By Intention“ macht Ihren Code verständlicher und erlaubt Ihnen, die Zahl von Detailkommentaren markant zu reduzieren.

Und hier eine Liste von Dingen, die *nicht* in Kommentare gehören:

- Eine Liste der exportierten Funktionen der Datei. Lassen Sie diese Arbeit dafür vorgesehenen Werkzeuge tun. Die können das besser und zuverlässiger als Sie.

- Revision History. Diese Information stellt Ihnen Ihre Versionsverwaltung zur Verfügung. Hingegen kann es nützlich sein, das Datum der letzten Änderung und den dafür verantwortlichen Entwickler fest zu halten.

- Eine Liste der Dateien, welche die Datei verwendet. Auch hierbei können Sie sich auf entsprechende Werkzeuge verlassen.

- Der Dateiname. Wenn der Name der Date zwingend in der Datei erscheinen muss, dann automatisieren Sie diese Aufgabe. Sie haben nämlich keine Lust, daran zu denken, den Dateikopf anzupassen, wenn Sie die Datei verschieben oder umbenennen. Und Sie wissen ja: Das kommt häufiger vor, als Ihnen lieb sein kann.

Ausführbare Dokumente

Es ist verhältnismässig einfach, eine aktuelle und korrekte Dokumentation auf der Basis des Programmcodes zu generieren. Werkzeuge wie JavaDoc, Doxygen oder Doc++ und wie sie alle heissen, helfen Ihnen dabei. Aber nicht alle Informationen können direkt aus dem Quellcode extrahiert werden. Dazu gehören z.B. Konfigurationen, Datenbankschemata, etc. Auch hier gilt der Grundsatz: „Keine Wiederholung“. Legen Sie sich jeweils auf eine steuernde Quelle fest und generieren Sie die davon abhängigen Dokumente. Ihr Quellcode ist das Modell: eine Sicht dieses Modell kann compiliert werden; andere sind dazu gedacht, ausgedruckt oder in einem Webbrowser angezeigt zu werden. Es ist unser Ziel, immer nur am Modell zu arbeiten – dabei handelt es sich entweder um den Code oder um eine anderes Dokument – und alle auf dem Modell basierenden Sichten sollen automatisiert aktualisiert werden können. HTML, XML und XSLT leisten hier wertvolle Dienste.

Und plötzlich ist das Erstellen einer brauchbaren und aktuellen Dokumentation gar nicht mehr so schlimm.

Externe Dokumentation

Auch für diese Form der Dokumentation gilt: Vermeiden Sie Wiederholung und automatisieren Sie! Auch für das Erstellen der externen Dokumentation, z.B. Benutzerhandbücher, eignen sich HTML und XML; Alternativ können Sie auch ein spezielles Autorenwerkzeug einsetzen.

Was wird wie getestet

Behandelte Themen

- Testen im kleinen: Unit-Tests
- Teilsysteme testen
- Ressourcenverbrauch, Fehlersituationen und Wiederherstellung
- Perfomance-Tests
- Usability-Tests (Tests auf Bedienerfreundlichkeit)

Dies sind die wichtigsten Tests, die Sie durchführen müssen:

- Unit Tests
- Integrationstests
- Validierung
- Ressourcenverbrauch, Fehlersituationen und Wiederherstellung
- Perfomance-Tests
- Usability-Tests (Tests auf Bedienerfreundlichkeit)

Diese Liste ist ein guter Anfang. Sie ist aber nicht vollständig und je nach Projekt ist es nötig, weitere Testvarianten zu berücksichtigen.

Unit Testing (Modultest)

Bei einem Unit-Test handelt es sich um Code, der die in einem Modul verfügbare Funktionalität verwendet. Typischerweise erzeugt der Unit-Test das für den Test benötigte Umfeld und ruft dann die Routinen auf dem zu testenden Modul auf. Der Test vergleicht dann die zurückgegebenen Werte entweder mit erwarteten Werten oder den Ergebnissen früherer Testläufe (regressives Testen). Der Unit-Test ist die Grundlage für alle weiteren Arten von Tests. Wenn bereits die Einzelteile Ihres Systems für sich nicht funktionieren, werden sie auch im Zusammenspiel nicht gut arbeiten. Daher müssen alle verwendeten Module ihre Tests bestehen, bevor Sie überhaupt daran denken, weiter zu machen.

Sobald alle benötigten Module für sich funktionieren, können Sie sich an die nächste Stufe wagen: Es gilt das Zusammenspiel der Komponenten untereinander zu testen.

Integrationstests

Wenn wir eine getestete Komponente im Gesamtsystem integrieren, können wir darauf vertrauen, dass sich die einzelne Komponente wie erwartet verhält. Integrationstests zeigen, dass die wichtigen

Teilsysteme für sich und im Zusammenspiel funktionieren. Auch auf dieser Stufe sind saubere Schnittstellen auf der Basis von Verträgen eine wichtige Voraussetzung für die reibungslose Integration. Ihr Fehlen bildet den idealen Nährboden für Fehler und gehört zu den häufigsten Gründen für Integrationsprobleme.

Integrationstests sind lediglich eine Erweiterung des Unit-Testing. Jetzt testen Sie einfach, ob die einzelnen Teilsysteme ihre Verträge einhalten.

Validierung

Hier geht es um die zentrale Frage: Erfüllt das System die Anforderungen? Prüfen Sie früh und regelmässig, ob Ihre Anwendung die Anforderungen erfüllt, Stichwort: Requirement Engineering. Prototypen und Benutzerschnittstellen-Entwürfe können Ihnen hier weiter helfen. Denken Sie daran: Ein tadellos funktionierendes System ist nutzlos, wenn es die falschen Probleme löst.

Ressourcenverbrauch, Fehlersituationen und Wiederherstellung

Nachdem Sie eine gute Vorstellung davon haben, ob Ihr System sich unter idealen Bedingungen korrekt verhalten wird, stellt sich die Frage: Wie verhält sich das System in der richtigen Welt? In der richtigen Welt sind die Ressourcen begrenzt und können knapp werden. Dazu gehören:

- Arbeitsspeicher
- Plattenspeicher
- CPU Bandbreite
- Zeit
- Datendurchsatz
- Netzwerkbandbreite
- Farbtiefe und Bildschirmauflösung

Fehler aufgrund von Ressourcenbeschränkungen können nur teilweise geprüft werden. Oft ist es auch schwierig, eine geeignete Recovery-Strategie für den Fehlerfall zu definieren. Wenn Ihr Programm feststellt, dass der verfügbare Speicher erschöpft ist, dann sind Ihre Optionen begrenzt: Es ist durchaus denkbar, dass Sie nichts mehr tun können ausser Ihr Porgramm schief gehen zu lassen. Wenn ihr System schliesslich versagt (und es wird versagen), wird es das mit Würde und Anstand tun? Wird es so gut wie möglich versuchen, die Arbeit seiner Anwender zu bewahren? Oder wird es dem Anwender eine garstige Fehlermeldung entgegen schleudern und alles mit sich in den Abgrund reissen?

Perfomance-Tests

Performance- und Stresstests spielen je nach Art Ihres Projekts eine entscheidende Rolle. Fragen Sie sich, ob die Software Performanceanforderungen unter realen Bedingungen erfüllt. Je nach Projekt benötigen Sie spezielle Test-Hardware und Software, um die Last realistisch simulieren zu können.

Usability-Tests (Tests auf Bedienerfreundlichkeit)

Diese Testform unterscheidet sich markant von den bisher aufgeführten Testarten. Tests auf Bedienerfreundlichkeit und Benutzbarkeit wird unter realen Bedingungen mit tatsächlichen Anwendern durchgeführt. Hier stehen klar die Anwender und ihre Bedürfnisse im Vordergrund. Eine gute Anwendung ist wie ein massgeschneiderter Handschuh: sie passt wie angegossen. Gerade uns Softwareentwicklern fällt es mitunter schwer, die Benutzerbedürfnisse wertfrei zu akzeptieren. Sie sollten Usability-Tests genau wie Validierung und Verifikation so früh wie möglich durchführen. Sie ersparen sich und Ihren Anwendern damit viel Ärger und Mühe. Um es in den Worten von Hunt&Thomas zu sagen: Das Verfehlen der Benutzbarkeitsanforderungen ist genauso ein Fehler wie durch Null zu dividieren.

Wie wird getestet?

Behandelte Themen

- Regressives Testen
- Der Umgang mit Testdaten
- GUI-Tests und lose Koppelung
- Testsaboteure
- Testabdeckung

Nachdem wir uns angesehen haben, was wir testen müssen, widmen wir uns jetzt der Frage, wie wir testen. Dies betrifft die folgenden Punkte:

- Regressionstests
- Testdaten
- GUI-Tests
- Tests testen
- Sorgfältig testen

Regressionstests

Ein Regressionstest vergleicht das Ergebnis des aktuellen Tests mit früheren oder bekannten Werten. Da Änderungen am Code immer die Gefahr in sich tragen, dass wir bestehende Funktionalität brechen, benötigen wir ein Sicherheitsnetz, dass uns rasch und sicher auffängt. Damit können wir dafür sorgen, dass wir unser System nicht kaputt flicken.

Alle bisher erwähnten Testarten können als Regressionstests ausgeführt werden. Gerade dies ist eine enorme Stärke des xUnit-Testframeworks.

Testdaten

Woher nehmen wir die Daten, die wir für die Durchführung unserer Tests benötigen? Es stehen zwei Arten von Daten zur Auswahl: Echte Daten oder synthetische. In der Regel brauchen wir beide Arten, da sie uns helfen, unterschiedliche Arten von Fehlern zu entdecken.

Echte Daten

Echte Daten stammen aus unterschiedlichen Quellen: Auszug aus einem bestehenden System, Daten aus einem Konkurrenzprodukt oder von einem Prototyp. Echte Daten sind typische Benutzerdaten. Dabei sind wir oft erstaunt, wenn wir herausfinden, was diese *typischen* Daten ausmacht. Damit helfen uns echte Daten, konzeptionelle Missverständnisse und Fehler aufzudecken.

Synthetische Daten

Synthetische Daten sind künstliche Daten. Wir verwenden sie, um spezifische Aspekte der Anwendung zu simulieren. Verwenden Sie synthetische Daten in folgenden Situationen:

- Die benötigte Datenmenge ist sehr gross, zumindest grösser als die momentan verfügbaren echten Daten.
- Sie benötigen Daten, um die Grenzbereiche zu testen. Dabei kann es sich auch um fehlerhafte Daten handeln.
- Sie benötigen Daten, die einer bestimmten statistischen Verteilung folgen: Was passiert, wenn jede dritte Transaktion fehlschlägt? Wie verhält sich unser Sortieralgorithmus bei vorsortierten Daten? Wie verhält sich unser Logg-Server, bei einer temporären Häufung von gleichartigen Fehlern?

GUI-Tests

Das Testen der tatsächlichen grafischen Benutzerschnittstelle erfordert meist spezielle Testwerkzeuge. In der einfachsten Form basieren diese Werkzeuge auf einem „Capture / Playback"-Mechanismus, der das Aufzeichnen und Abspielen von Benutzereingaben erlaubt. Je nach Art des GUI, können GUI-Tests manchmal nicht vollständig automatisiert werden. Da bleibt Ihnen dann nichts weiter übrig, als sich vor den Schirm zu setzen und manuell zu testen.

Doch verzweifeln Sie nicht! Einer der vielen Vorteile von Entwürfen, die aus lose gekoppelten Komponenten bestehen, ist ihre modulare Testbarkeit. Sinnvollerweise basieren Ihre Entwürfe auf einem sauberen Schichtenmodell, um die GUI und Geschäftslogik voneinander zu trennen. Im Detail bedeutet dies in der Regel die Umsetzung des MVC-Musters, so dass sie sowohl die Geschäftslogik (das Modell im MVC) als auch die GUI-Controller unabhängig voneinander und ohne GUI testen können. Das macht zwar die GUI-Tests nicht überflüssig, reduziert aber ihren Umfang und ihre Komplexität erheblich. Zudem können Fehler dadurch wesentlich einfacher lokalisiert werden. Haben Sie Ihre Modell- und Controller-Klassen getestet und im GUI-Test treten Probleme auf, so können Sie davon ausgehen, dass sich der Fehler im GUI-Code eingeschlichen hat.

Tests testen: Testsaboteure

Aus der Tatsache, dass wir nicht in der Lage sind, perfekte und fehlerfreie Programme zu schreiben, folgt, dass es nicht möglich ist, perfekte und fehlerfreie Tests zu schreiben. Zudem neigt der Entwickler beim Schreiben von automatisierten Unit-Tests dazu, dem eigenen Code gegenüber zu nachsichtig zu sein. So besteht die Gefahr, dass der Programmierer bei der Implementierung des Testes mehr die Rolle des Entwicklers der zu testenden Klasse (CUT = Class under Test) und weniger die Rolle des Testers einnimmt und dadurch zu wenig sorgfältig testest. Solche Unit-Tests können das Risiko fehlerhafter Implementierungen erhöhen, da sich Entwickler in der falschen Sicherheit wiegen, ihre Software sei gut getestet, obwohl in Tat und Wahrheit die Testqualität zu wünschen übrig lässt. Durch den gezielten Einsatz von Testsaboteuren lässt sich dieses Risiko minimieren.

Die Gesamtheit der Unit-Tests für eine Software bildet eine Art Sicherheitssystem, das uns alarmieren

soll, wenn sich ein Fehler in unseren Code eingeschlichen hat. Indem Sie nun versuchen, gezielt Fehler in den getesteten Code einzuschmuggeln, können Sie testen, ob Ihre „Alarmanlage“ auch tatsächlich funktioniert. Zu diesem Zweck verändern Sie als Testsaboteur gezielt den Programmcode und führen Fehler ein. Anschliessend überprüfen Sie, ob die entsprechenden Unit-Tests den Fehler zuverlässig aufdecken.

Sorgfältig testen: Testabdeckung

Beim erstellen der Testfälle stellt sich immer wieder die Frage, ob wir sicher sein können, dass wir unsere Codebasis gründlich genug getestet haben. Die kurze Antwort: Wir können und werden nie sicher sein. Es ist nicht möglich, Software hundertprozentig zu testen. Wir können uns aber z.B. mit Hilfe von Code Coverage Werkzeugen einen Überblick über unsere Testabdeckung verschaffen, um so Code zu identifizieren, für welchen keine Tests existieren. Der Einsatz solcher Werkzeuge ist vorallem dann besonders sinnvoll, wenn das zu testende System nicht testgetrieben entwickelt worden ist, da für ein solches Systeme die Wahrscheinlichkeit nicht-getesteter Funktionalität naturgemäss grösser ist.

Der Einsatz von Code Coverage Werkzeugen ist sicher ein guter Start. Dabei dürfen wir jedoch eines nicht vergessen: Alleine die Tatsache, dass für einen bestimmten Codeteil ein entsprechender Test existiert, heisst noch lange nicht, dass dieser Teil vollständig getestet und fehlerfrei ist. Hierzu ein sehr einfaches Beispiel einer Funktion mit zwei Argumenten, die jeweils Zahlen zwischen 0 und 999 als gültige Werte akzeptieren (Quelle [Hunt&Thomas, Seite 243]):

```
int magic(int a, int b) {
  return a /(a+b);
}
```

Theoretisch existieren für diese kleine Funktion 10^6 Zustände, dabei verhält sich die Funktion für 10^6-1 Zustände korrekt, während es in einem Fall – wenn a + b Null ergibt – zu einem Fehler aufgrund einer Division durch Null kommt. Die Tatsache, dass die `magic` Funktion verwendet wird, garantiert noch nicht, dass die kritischen Zustände der Funktion getestet wurden. Um die Funktion vollständig zu testen, müssten *alle* möglichen Zustände identifiziert werden, um entscheiden zu können, was getestet werden muss. Hierbei handelt es sich um ein extrem schwieriges Problem, für das keine einfache Lösung existiert.

Der Einsatz von Code Coverage Werkzeugen entbindet uns nicht von der Pflicht, uns über die Daten für unsere Tests – hier in der Form von Werten für Funktionsargumente – Gedanken zu machen. Die Daten, die wir für unsere Tests verwenden und – meist noch zentraler – die Abarbeitungsreihenfolge des zu testenden Codes sind von entscheidender Bedeutung. Die folgende Faustregel soll uns an diesen Umstand erinnern:

> ***Faustregel***
>
> *Testen Sie die Zustandsabdeckung und nicht die Codeabeckung.*

Test Driven Development (TDD) – der „Test- First“-Ansatz

Der „Test-First“-Ansatz ist eine Vorgehensweise, welche das Schreiben von Testfällen als zentrale Entwicklertätigkeit ansieht. Test-First ist einerseits eine qualitätssichernde Tätigkeit und steuert andererseits auch den Softwareentwurf in Richtung Testbarkeit und Einfachheit. Das ideale „Test-First“-Vorgehen sieht wie folgt aus [Link02]:

- Bevor wir eine Zeile Produktionscode schreiben, entsteht ein entsprechender Test, der diesen Code auf die Probe stellt.
- Wir schreiben nur soviel Produktionscode, wie es der Test verlangt. D.h. wir wählen die einfachste Implementierung, die gerade funktioniert.
- Die Entwicklung findet in Mikroiterationen statt, d.h. in kleinen Schritten, in denen sich Testen und Kodieren abwechseln. Eine solche Mikroiteration dauert oft nicht länger als 10 Minuten.
- Zum Zeitpunkt der Integration des Produktionscodes ins Gesamtsystem, müssen alle Unit-Tests laufen.

Daraus ergeben sich die folgenden Vorteile:

- Hohe Testabdeckung der einzelnen Module. Dadurch werden Änderungen, die vorhandene Funktionalität zerstört, sofort entdeckt.
- Die Tests dokumentieren den Code, da sie sowohl die normale Verwendung als auch die erwarteten Reaktionen bei Fehlern zeigen.
- Rasches Feedback. Aufgrund der kurzen Mikroiterationen können wir nur wenig programmieren und daher auch nur wenig falsch machen.
- Der Entwurf eines Programmes wird wesentlich von den Tests bestimmt. Dadurch erhalten wir praktisch immer einfachere Entwürfe als wenn wir den gesamten Detailentwurf vorab erstellt hätten.

Testen, was schief gehen kann

Eine wichtige Testregel im Extreme Progamming lautet: „Test everything that could possibly break“ (dt. Teste alles, was schief gehen kann).

Der Kern dieser Aussage betrifft den Umstand, dass es keine allgemein gültige Regel dafür gibt, welche Elemente in einer Software besonders fehleranfällig sind. Jedes Team, jeder Entwickler macht seine spezifischen Fehler. Betrachtet man diese Fehler über die Zeit hinweg, so stellt man fest, dass diese sich oft ähneln. Daraus können systematisch Lehren gezogen werden, die das eigene Handeln verändern und damit auch die Art der begangenen Fehler. Dies gilt es auch beim Schreiben der Testfälle zu beachten.

Um kritische Teile zu identifizieren, die besonders sorgfältig getestet werden sollten, mag zudem folgender Hinweis helfen: Teste überall dort, wo bereits viele Fehler aufgetreten sind. Gemäss dem Pareto-Prinzip sind im Schnitt 90% der Fehler in weniger als 10% der Klassen eines System zu

finden.
Es stellt sich weiter die Frage nach der richtigen Anzahl von Tests. Auch hier muss jedes Team und jeder Entwickler sein persönliches Testniveau finden. Hier gilt es, sich iterativ an den jeweils optimalen Wert heran zu tasten. Werden zum Beispiel viele Fehler erst nach der Freigabe einer Software entdeckt, dann liegt die Vermutung nahe, dass zu wenig Testfälle geschrieben wurden. In diesem Fall lohnt es sich, die aufgetretenen Fehler sorgfältig zu analysieren, um Aufschluss über die Art der gemachten Fehler zu erhalten und das eigene Testverhalten zu verbessern.

Dagegen kann es vorkommen, dass das Schreiben der Testfälle zunehmend Zeit zu verschlingen scheint und die Tests bei Refactorings oder Erweiterungen nicht merklich beim Aufdecken von Fehlern helfen. In einer solchen Situation lohnt es sich darüber nachzudenken, welche Testfälle man weglassen kann, weil keine Fehler der getesteten Art mehr auftreten.

Vertragsorientiert testen

Wenn wir für ein Modul einen Unit-Test schreiben, dann testen wir am besten, ob das Modul seinen Vertrag erfüllt. Damit schlagen wir zwei Fliegen mit einer Klappe: Zum einen, prüfen wir, ob das Modul seinen Vertrag einhält und zum anderen, ob der Vertrag wirklich das ausdrückt, was wir meinen. Wir wollen sicher stellen, dass das Modul seine Dienste über einen weiten Bereich von Testfällen und Grenzbedingungen korrekt anbietet.

Wann wird getestet?

Behandelte Themen

- Wie häufig werden Testfälle ausgeführt?
- Wie wird vorgegangen, wenn ein Fehler entdeckt wird?
- Welche Gelegenheiten bieten sich, um Tests zu schreiben, zu ändern oder zu überarbeiten?
- Tests als Hilfsmittel zum Erlernen neuer APIs.

In vielen Projekten sind systematische Tests etwas, das man sich für den Schluss aufhebt. Wenn alle anderen Arbeiten abgeschlossen sind und noch Zeit übrig ist, dann wird getestet. Testen wird als notwendiges Übel empfunden, das man am liebsten einfach vergessen würde. Tests haben den Ruf mühselig, zeitaufwendig und vorallem langweilig zu sein; Testen macht einfach keinen Spass! Und noch schlimmer: Tests zeigen oft, dass die Qualität eines Produktes den festgelegten Anforderungen nicht entspricht und daher eine Freigabe erst nach umfangreichen Anpassungen möglich ist. Damit wird der Test zum Überbringer der schlechten Nachrichten.

Die beste Strategie, um Tests nutzbringend einzusetzen ist, für das Testen von Beginn weg genügend Zeit einzuplanen und sich auf schlechte Nachrichten gefasst zu machen: Setzen Sie Tests so ein, dass wenn es schlechte Nachrichten zu verkünden gibt, dies durch Tests so früh wie möglich geschieht!

Wir möchten so häufig testen, wie es geht und sicher immer *bevor* wir Code in unser Programmcode Repository einchecken. In manchen Versionsmanagement-Tools können wir diese Aufgabe automatisieren. Ansonsten hilft eine simple Batchdatei, die Testausführung zu automatisieren. Normalerweise ist es kein Problem sämtliche Unit- und Integrationstests regelmässig regressiv auszuführen. Manche Tests können Sie dagegen nicht mit vertretbarem Aufwand so häufig ausführen. Dazu gehören beispielsweise Stresstests, die bestimmte Hard- und Software benötigen, deren Bereitstellung zeit- und kostenintensiv sein kann. Solche Tests können seltener ausgeführt werden. Achten Sie aber darauf, dass Sie diese Tests als wiederkehrende Tätigkeit in Ihre Planung aufnehmen und sicher stellen, dass sie auch tatsächlich ausgeführt werden. Versuchen Sie unter allen Umständen die Testausführung soweit als möglich zu automatisieren.

Testgelegenheiten

Schauen wir uns als nächstes an, *wann* wir Testcode schreiben. Nachfolgend eine Liste von Gelegenheiten für das Schreiben von neuen Testfällen bzw. das Anpassen von bestehendem Testcode:

Gelegenheit 1

Wir schreiben Tests jedesmal, wenn es darum geht, eine neue Anforderung umzusetzen. Dies entspricht dem Vorgehen in der test-getriebenen Entwicklung und sollte so oft als möglich angewandt

werden.

Gelegenheit 2

Nach der Umsetzung einer Anforderung werden sowohl der Applikationscode als auch der Testcode bezüglich Qualität und Refactoring-Möglichkeiten untersucht. Der Testcode wird zudem bezüglich Testabdeckung geprüft (vgl. hierzu den Abschnitt Testabdeckung).

Gelegenheit 3

Immer wenn wir Defizite in unserem Testcode feststellen, z.B. bezüglich mangelnder Testabdeckung oder beim Versuch, bestehenden Code zu verstehen.

Gelegenheit 4

Bei einem Refactoring von bestehendem Code. Dabei warnen uns die bestehenden Testfälle, wenn wir mit dem Refactoring einen Fehler in unser Programm eingeschleppt haben.

Gelegenheit 5

Wenn Fehler auftreten. Das ist eine der wichtigsten Testregeln überhaupt. Leider wird diese Möglichkeit oft verschenkt. Wenn ein Fehler durch die Testmaschen schlüpft, dann benötigen Sie einen zusätzlichen Test, um ihn das nächste Mal zu erwischen.

Wenn ein Tester aus Fleisch und Blut einen Fehler entdeckt, dann sollte dies das erste und letzte Mal gewesen sein. Wir modifizieren und erweitern unsere automatisierten Tests, damit sie den Fehler von nun an zuverlässig aufdecken, für immer und ohne Ausnahme, egal, wie trivial und einfach die Behebung des Fehlers auch scheint und egal, ob der Entwickler lautstark verkündet: „Nein, dieser Fehler wird nicht wieder auftauchen!“ Denn der Fehler *wird* wieder auftauchen, ganz bestimmt! Und wir haben einfach nicht die Zeit, immer und immer wieder die gleichen Fehler zu jagen, die automatische Tests für uns einfangen können.

Gelegenheit 6

Idealerweise sollte die Dokumentation für eine API eine Reihe von Unit-Test enthalten, welche die korrekte Verwendung der API aufzeigen. Leider ist dies für die wenigsten der heute verfügbaren Programmierschnittstellen, Frameworks und Bibliotheken der Fall.

Wenn der Entwickler die Anwendung einer API erlernen will, dann helfen Unit-Tests. Damit können zwei komplementäre Ziele erreicht werden: Die durch den Lernenden getroffenen Annahmen bezüglich der korrekten Verwendung der API, werden dokumentiert. Sie liegen in ausführbarer und damit testbarer Form vor. So entsteht Schritt um Schritt eine „Dokumentation“, welche die korrekte Anwendung der API beschreibt. Das zweite Ziel ist weniger offensichtlich aber nicht weniger wichtig und nützlich: Da die korrekte Verwendung der API durch eine Reihe von automatisiert ausführbaren Testfällen dokumentiert ist, können neue Versionen der API rasch bezüglich Rückwärtskompatibilität bestehender Funktionalität getestet werden.

Gerade bei einem umfangreichen und komplexen API besteht die Gefahr, dass bestimmte Dinge nicht

so funktionieren wie sie sollten. Dies insbesondere dann, wenn für die API keine Testsammlung existiert. Somit brechen diejenigen Testfälle, welche Funktionalität der API verwenden, die sich nicht spezifikationsgemäss verhalten. Ein solcher Test dokumentiert einen Fehler in der Implementierung der API. Im Idealfall wird der Testfall an den für die API-Implementierung verantwortlichen Entwickler übergeben und der Fehler wird behoben. Der Fehler auslösende Testfall wird bis zur Fehlerbehebung aus der Testsammlung entfernt, entweder mit Hilfe der Versionsverwaltung oder einfacher indem der entsprechende Code auskommentiert wird.

Teststrategien

Behandelte Themen

- Was tun, wenn das zu testende Modul eine kostspielige oder komplexe Ressource benötigt?
- Wie testen wir, dass zwei Objekte korrekt miteinander arbeiten?
- Wie testen wir einfach, ob eine bestimmte Aufrufreihenfolge befolgt wurde?
- Wie testen wir, ob Fehlercode im Fehlerfall korrekt ausgeführt wird?

Dummy, Attrappen und Mocks

Was tun, wenn das zu testende Modul eine kostspielige oder komplexe Ressource benötigt? Führen Sie eine Attrappe ein, die diese Ressource simuliert; das typische Beispiel ist eine Datenbank oder eine Netzwerkressource. Der Einsatz von Attrappen setzt voraus, dass die zu simulierenden Ressourcen in Form von Schnittstellen beschrieben sind. Nur so ist es möglich, die tatsächliche Implementierung transparent durch die Attrappe zu ersetzen, vgl. hierzu den Abschnitt *Schnittstellen-basierter Entwurf*.

Mock Object

Eine ausgeklügelte Variante der Attrappen sind die sog. Mock-Objekte [MockObjects]. Diese simulieren nicht nur die jeweilige Ressource, sondern sie sind auch in der Lage, ihre korrekte Verwendung selber zu überprüfen. Zum Thema der Mock Objekte liessen sich ganze Bücher füllen. Sie bilden eine der mächtigsten Testmuster überhaupt. Die Verwendung eines Mock-Objekts anstelle der tatsächlichen Implementierung bietet sich in den folgenden Fällen an:

- Das echte Objekt ist schwierig aufzusetzen.
- Das echte Objekt zeigt Verhalten, das schwierig auszulösen ist, z.B. das Auslösen eines Netzwerkfehlers.
- Das echte Objekt hat oder ist eine Benutzerschnittstelle.
- Das echte Objekt ist langsam.
- Der Test muss das echte Objekt fragen, wie dieses verwendet worden ist, z.B. um sicher zu stellen, dass eine bestimmte Callback-Methode aufgerufen worden ist.
- Das echte Objekt existiert noch nicht (vgl. Test-First).

Mock-Objekte holen den Grossteil des Test-Codes zu sich. Dadurch lassen sich Testfälle wesentlich vereinfachen. Mock-Objekte werden in das zu testende Objekt eingeschleust. So wird es möglich, das Objekt von innen zu testen.

Self Shunt

Wie testen wir, dass zwei Objekte korrekt miteinander arbeiten? Indem wir das zu testende Objekt mit unserem Testfall kommunizieren lassen anstelle des tatsächlichen Objekts. Der Testfall wird dabei zu einer Miniattrappe.

Log String

Wie testen wir einfach, ob eine bestimmte Aufrufreihenfolge befolgt wurde? Indem wir ein Mock-Objekt oder den Testfall als Self Shunt verwenden, der die Methodenaufrufe in einem Log String aufzeichnet und diesen Log String mit der erwarteten Aufrufsequenz vergleicht.

Crash Test Dummy

Wie testen wir, ob Fehlercode im Fehlerfall korrekt ausgeführt wird? Indem wir eine Attrappe einführen, die den Fehler produziert. Das zu testende Objekt kann dann einfach darauf geprüft werden, ob es korrekt mit diesem Fehler umgeht.

Testfaustregeln

Dieser Abschnitt enthält eine kleine List von wichtigen Testfaustregeln, die Sie bei Ihrer täglichen Arbeit beherzigen sollten:

- Eine Sammlung von automatisiert ausführbaren Testfällen ist ein mächtiges Instrument, um Fehler zu entdecken.
- Die Programmierarbeit ist nicht erledigt bis alle Tests geschrieben sind.
- Legen Sie für Ihren Testcode die gleichen Qualitätsmassstäbe an wie für Ihren Produktionscode. Auch für Ihren Testcode gilt: Reparieren Sie zerbrochene Fenster - vermeiden Sie insbesondere jegliche Form der Wiederholung im Code.
- Betrachten Sie Ihre Testfälle als wichtigen Teil der Systemdokumentation.
- Testen Sie häufig und früh.
- Testen Sie die Verträge Ihrer Module.
- Testen Sie alles, was schief gehen kann.
- Testen Sie Ihre Tests mit Hilfe von Saboteuren.
- Nehmen Sie nichts an, beweisen Sie es – mit einem Test.
- Testen Sie Zustandsabdeckung und nicht Codeabdeckung.
- Führen Sie Testfälle regelmässig aus. Führen Sie Tests genauso häufig aus wie Sie ihre Anwendung compilieren – jeden Test mindesten einmal am Tag.
- Wenn Sie eine Fehlermeldung erhalten, erstellen Sie einen Test, der den Fehler auslöst.
- Es ist besser eine unvollständige Testsammlung auszuführen als vollständige nicht auszuführen.
- Denken Sie an die Grenzbedingungen unter den die Dinge schief gehen könnten und konzentrieren Sie Ihre Aufmerksamkeit darauf.
- Denken Sie daran zu testen, das Exceptions ausgelöst werden, wenn erwartete Dinge schief gehen.
- Lassen Sie sich aus Angst davor, dass Tests nicht alle Fehler finden können, nicht davon abhalten, Tests zu schreiben, welche die meisten Fehler finden.

C++ Spezifika

Der idiomatische Gebrauch von C++

Idiom
[das; griechisch, "Eigentümlichkeit"]
die Eigentümlichkeiten der Sprache eines Menschen, einer Region oder Mundart.

C++ ist eine komplexe und mächtige Programmiersprache. Für den erfolgreichen Einsatz von C++ ist die Kenntnis der gängigen C++ Idiome ein Muss. Jeder C++ Programmierer muss sich bewusst sein, dass die von ihm erstellten Programme im Kontext von Idiomen existieren. Um ein Bild zu gebrauchen: Die Gesamtheit der gängigen C++ Idiome beschreibt einen Pfad, auf dessen Spuren der C++ Entwickler zur jeweiligen Lösung gelangen sollte. Sie tun gut daran, diesen bewährten Pfad nicht leichtfertig zu verlassen. Die Kenntnis und Verwendung der C++ Idiome stellt eine der besten Möglichkeiten dar, um zu klaren, effektiven und wartbaren Programmen zu gelangen. In diesem Sinne ähneln Idiome den Entwurfsmustern. Im Gegensatz zu Mustern sind Idiome aber wesentlich konkreter, da sie i.d.R. spezifische Sprachaspekte und Implementierungen betreffen.

Machen Sie sich also mit den gängigen Idiomen von C++ vertraut; Bücher wie Effective C++ [Meyers], Exceptional C++ [Sutter] oder C++ Gotchas [Dewhurst] helfen Ihnen dabei.

Compilerwarnungen nicht ignorieren

Es versteht sich eigentlich von selbst, dass Compilerwarnungen nicht ignoriert werden sollen. Es ist sinnlos, Zeit in Testfälle zu investieren, um Fehler im Code zu entdecken, wenn bereits die Compiler erzeugten Hinweise unbeachtet bleiben. Versuchen Sie Compilerwarnungen zu verstehen, anstatt sie zu ignorieren oder gar auszuschalten. Wenn Sie Compilerwarnungen ignorieren, dann verschwenden Sie Ihre Zeit. Denn mit grosser Wahrscheinlichkeit versuchen Sie, ein Problem von Grund auf zu lösen, während der Compiler Sie bereits auf die Ursache des Problems hinweist.

Vererbung

Beim Entwurf einer Klasse gilt es, die wichtige Frage zu klären, ob die Klasse spezialisiert werden kann, d.h. ob es möglich sein soll, abgeleitete Klassen zu deklarieren und umzusetzen.

In C++ sollte nur Public Inheritance (öffentliche Vererbung) verwendet werden. Daher reden wir im Folgenden nur noch von Vererbung im Sinne der öffentlichen Vererbung von C++. Für diese Form der Vererbung gilt die Faustregel:

> ***Faustregel***
>
> *Vererbung sollte ausschliesslich zur Modellierung von Spezialisierungsbeziehungen verwendet werden. Der hauptsächliche Einsatz von Vererbung als Mechanismus zur Wiederverwendung führt zu schwer verständlichen und schlecht wartbaren Entwürfen und ist daher nicht ratsam.*

Weiter gelten allgemein und in C++ insbesondere die beiden folgenden Faustregeln:

> ***Faustregeln***
>
> *Abstrakte Klassen müssen Basisklassen sein.*
>
> *Basisklassen sollten in der Regel abstrakte Klassen sein.*

Wie an anderer Stelle bereits ausgeführt wurde, ist eine wichtige Voraussetzung für die Testbarkeit von Entwürfen, die konsequente Verwendung von Schnittstellen. Insbesondere in streng typisierten Sprachen wie C++ sind Schnittstellen die unabdingbare Voraussetzung für das effiziente automatisierte Unit-Testing, da nur so die nötige Substituierbarkeit von Implementierungen erreicht wird, die für das modul-orientierte Testen unabdingbar ist.

Schnittstellen werden in C++ mit Hilfe von sogenannten Schnittstellen Klassen (Interface Classes) umgesetzt. Eine Schnittstellen-Klasse ist eine abstrakte Basisklasse ohne Daten, einem virtuellen Destruktor und typischerweise mit ausschliesslich virtuellen Memberfunktionen und ohne deklarierten Konstruktor. Schnittstellen-Klassen werden manchmal auch als Protokoll-Klassen bezeichnet, da sie lediglich das Verwendungsprotokoll für Instanzen beschreiben, welche die definierte Schnittstelle unterstützen, ohne eine spezifische Implementierung umzusetzen. Den Schnittstellen-Klassen ähnlich sind die sogenannten „Mix-In"-Klassen, die aber ein Minimum an Daten und Implementierung enthalten können.

Nachfolgend zwei einfache Beispiele für eine `Action` und eine `Logging` Schnittstelle.

```
class Action {
  public:
    virtual ~Action() {};
    virtual void operator()() = 0;
};

class Logging {
  public:
    virtual ~Logging() {};
    virtual void log(const char* aMsg) = 0;
};
```

Eine Klasse kann mehrere Schnittstellen unterstützen, d.h. implementieren. Diese Implementierungsbeziehung zwischen der Klasse und der von ihr unterstützten Schnittstellen kann in C++ mit Hilfe von Mehrfachvererbung erreicht werden.

Design By Contract (DBC) in C++

Bevorzugen Sie die Verwendung des `assert` Makro gegenüber Kommentaren, um Zusicherungen (Vorbedingungen, Nachbedingungen und Invarianten) zu formulieren. Ein eingeschaltetes `assert` führt eine Prüfung zur Laufzeit durch und kann daher nicht so leicht ignoriert werden wie ein Kommentar. Passen Sie aber auf, dass Sie durch die Verwendung von `assert` nicht ungewollte Seiteneffekte einführen, indem Sie z.B. eine Methode aufrufen, die den Zustand eines Objekts ändert. Vgl. Sie hierzu auch *Gotcha #28 Side Effects in Assertions* in [Dewhurst].

Index

A

Agil Dokumentieren 21
API erlernen 35
Architektur 4, 5, 6, 8, 9, 20, 21

B

Bedienerfreundlichkeit 26, 28
Bugreport 35

C

C++ Spezifika 13, 14, 40, 41
C++, Vererbung 11, 13, 14, 40
C++, Design By Contract 11, 15, 41
C++, Idiome 40
Codeabdeckung 29, 31, 32, 35, 39
Compilerwarnungen 40
Crash Test Dummy 38

D

Das Gesetz von Demeter 12
Design By Contract 11, 15, 41
Detailspezifikation 22, 23
Dokumentation 5, 6, 7, 21, 22, 23, 25, 35
Dokumentation, externe 25
Dokumentation, interne 22
Dummy 14, 37, 38

E

Entwurfsmuster 14, 23

F

Fehlermeldung 35

G

GUI-Tests 29, 30

I

Idiom 40
Integrationstest 5, 10, 26, 27, 34
Intellektuelle Beherrschbarkeit 7, 8

K

Kommentare 24
Kommentare, nicht angebrachte 24

L

Log String 38
Lose Koppelung 9, 10, 14, 30

M

Mindmap 19
Mock Object 37

O

Open/Closed-Prinzip 9, 11, 12

P

Programming by Intention 11, 15

R

Refactoring 20, 23, 35
Regressionstests 29

S

Schnittstelle 7, 8, 9, 10, 12, 13, 14, 15, 16, 23, 27, 37, 41
Self Shunt 38
Singleton 9
Softwareslum 9

T

Tesen, regressives 29
Test Driven Development (TDD) 4, 10, 32, 37
Test First 4, 32
Testabdeckung 29, 31, 32, 35, 39
Testbarkeit 5, 6, 7, 8, 10, 14, 30, 32, 41
Testdaten 29
Testen von GUIs 29, 30
Testen, sorgfältig 29, 31, 32, 35
Testen, vertragsorientiert 33
Testen, wann 34
Testen, wann – wenn Fehler auftreten 35
Testen, wie 29
Testfaustregeln 39
Testgelegenheiten 34
Testgetriebene Entwicklung 4, 7, 32
Tests testen 29, 30
Tests von Ressourcenverbrauch 26, 27
Tests von Wiederherstellung 26, 27
Testsaboteure 29, 30
Teststrategien 14, 37
Teststrategien, Crash Test Dummy 38
Teststrategien, Dummy 14, 37, 38
Teststrategien, Log String 38
Teststrategien, Mock-Object 37
Teststrategien, Self Shunt 38

U

UML 4, 6, 17, 18, 19, 22, 23
UML Tools zur Codeerzeugung 18, 22
UML Tools zur Codeerzeugungs 18
UML und Code 22, 23
Unit Test 26
Usability-Tests 26, 28

V

Validierung 26, 27
Validierung und Verifikation 28
Vererbung 11, 13, 14, 40
Vertragsorientiert testen 33

Z

Zusicherung 15, 16, 20, 41
Zustandsabdeckung 31, 39

Ressourcen

[60Sekunden]	http://www.empros.ch/nemoslog/50214994c313ae801.php
[Ambler]	Scott W. Ambler, Agile Modeling, Wiley, 2002.
[C++ Idioms]	http://www.bell-labs.com/user/cope/Patterns/C++Idioms/EuroPLoP98.html
[Code-Druck]	http://www.empros.ch/nemoslog/50214994c51107501.php
[Dewhurst]	Stephen C. Dewhurst, C++ Gotchas. Avoiding Common Problems in Coding and Design, Addison-Wesley, 2002.
[Fowler00]	Martin Fowler, UML konzentriert, Addison-Wesley, 2000.
[Fowler99]	Martin Fowler, Refactoring. Improving the Design of Existing Code, Addison-Wesley, 1999.
[Hunt&Thomas]	Andrew Hunt & David Thomas, The Pragmatic Programmer: From Journeyman to Master, Addison-Wesley, 1999.
[Jeffries]	Ron Jeffries, Extreme Programming Installed, Addison-Wesley, 2002.
[Meyers]	Scott Meyers, Effective C++. 50 Specific Ways to Improve your Programs and Designs, Addison-Wesley, 1997.
[MockObjects]	http://www.mockobjects.com/wiki/
[Starke]	Gernot Starke, Effektive Software- Architekturen. Ein praktischer Leitfaden, Hanser, 2002.
[Sutter]	Herb Sutter, Exceptional C++, Addison-Wesley, 1999.