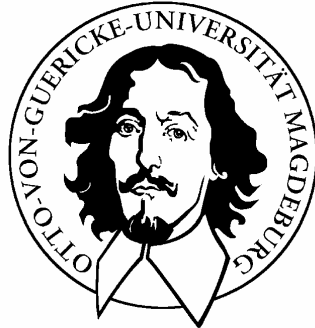


OTTO-VON-GUERICKE-UNIVERSITÄT MAGDEBURG



Fakultät für Informatik

Institut für Verteilte Systeme, Arbeitsgruppe Softwaretechnik

Diplomarbeit

Konzeptionelle Ansätze zum Test Java-basierter Software

Verfasser:

Yvonne Seitschek

16.02.2005

Betreuer:

Prof. Dr.-Ing. habil. Reiner R. Dumke

Universität Magdeburg

Fakultät für Informatik

Inhaltsverzeichnis

1	Einleitung.....	11
1.1	Motivation.....	11
1.2	Aufbau der Arbeit.....	11
2	Testmethoden	13
2.1	Statische Testmethoden.....	13
2.1.1	Checklistenverfahren.....	13
2.1.2	Symbolische Programmtestung.....	14
2.1.3	Programmverifikation	14
2.2	Dynamische Testmethoden	18
2.2.1	Black Box Test.....	18
2.2.2	White Box Test.....	20
3	Besonderheiten/Probleme beim Testen objektorientierter Software.....	23
3.1	Merkmale objektorientierter Software	23
3.2	Testgegenstände	24
3.3	Kapselung.....	24
3.4	Vererbung.....	25
3.4.1	Mehrfachvererbung	26
3.5	Abstrakte Klassen.....	27
3.6	Polymorphismus.....	27
3.6.1	Statisches Binden	28
3.6.2	Dynamisches Binden.....	28
3.6.3	JoJo-Effekt	28
4	Testablauf	30
4.1	Unittest	30
4.1.1	Invariant Boundaries	31
4.1.2	Nonmodaler Klassentest.....	34
4.1.3	Quasi-modaler Klassentest.....	35
4.1.4	Modaler Klassentest	38
4.2	Integrationstest	40
4.2.1	Big Bang Integration	41
4.2.2	Bottom-up Integration.....	43
4.2.3	Top-down Integration.....	46
4.2.4	Collaboration Integration	50
4.2.5	Layer Integration.....	53

4.2.6	Client/Server Integration	55
4.2.7	Distributed Services Integration	59
4.2.8	Backbone Integration	63
4.2.9	High-frequency Integration	67
4.3	Systemtest	70
4.3.1	Umgebungstest	71
4.3.2	Funktionstest	71
4.3.3	Test nichtfunktionaler Anforderungen	71
4.4	Abnahme-/Akzeptanztest	72
4.4.1	Erforderliche Einzeltests für die Abnahme	72
4.4.2	Fehlerklassifikation	73
4.4.3	Testdurchführung	73
5	Übersicht Test-Tools	75
5.1	JUnit	75
5.1.1	Aufbau JUnit Framework	76
5.1.2	Beispiel	77
5.2	JUnitEE	81
5.3	CruiseControl	82
5.4	Bugkilla	83
5.5	qftestJUI	84
5.6	Exacum	85
6	Testframework für Java	87
6.1	Was ist Java?	87
6.2	Besonderheiten beim Testen	87
7	Konzept für ein integriertes Java-Testtool	89
7.1	Anforderungen	89
7.2	Unit-/Integrationstest	89
7.3	Systemtest	90
7.4	Abnahme-/Akzeptanztest	91
8	Zusammenfassung / Ausblick	93

Abbildungsverzeichnis

Abbildung 2.1: Vor- und Nachbedingungen bei einer Programmanweisung (Quelle: [2]).....	15
Abbildung 2.2: Programmflussgraph (Quelle: [2]).....	21
Abbildung 3.1: Beispiel Vererbung	25
Abbildung 3.2: Beispiel Mehrfachvererbung.....	26
Abbildung 3.3: Beispiel Polymorphismus (Quelle: [12])	27
Abbildung 3.4: Ablaufverfolgung der JoJo Ausführung (Quelle: [1]).....	29
Abbildung 4.1: Stufenweiser Aufbau des Softwaretests.....	30
Abbildung 4.2: Generisches Zustandsmodell für Collections.....	36
Abbildung 4.3: Transitionsbaum für das quasi-modale Zustandsmodell.....	37
Abbildung 4.4: Legende Grafiken Integrationstest.....	41
Abbildung 4.5: Beispiel Big Bang Test (Quelle: [1])	42
Abbildung 4.6: Bottom-up Integration, 1. Schritt (Quelle: [1]).....	43
Abbildung 4.7: Bottom-up Integration, 2. Schritt (Quelle: [1]).....	44
Abbildung 4.8: Bottom-up Integration, 3. Schritt (Quelle: [1]).....	44
Abbildung 4.9: Bottom-up Integration, 4. Schritt (Quelle: [1]).....	44
Abbildung 4.10: Bottom-up Integration, Schlusskonfiguration (Quelle: [1]).....	45
Abbildung 4.11: Top-down Integration, 1. Schritt (Quelle: [1]).....	47
Abbildung 4.12: Top-down Integration, 2. Schritt (Quelle: [1]).....	47
Abbildung 4.13: Top-down Integration, 3. Schritt (Quelle: [1]).....	47
Abbildung 4.14: Top-down Integration, 4. Schritt (Quelle: [1]).....	48
Abbildung 4.15: Top-down Integration, Schlusskonfiguration (Quelle: [1])	48
Abbildung 4.16: Collaboration Integration, 1. Konfiguration (Quelle: [1])	51
Abbildung 4.17: Collaboration Integration, 2. Konfiguration (Quelle: [1])	51
Abbildung 4.18: Collaboration Integration, Schlusskonfiguration	51
Abbildung 4.19: Layer Integration, Architektur und Komponententestkonfiguration (Quelle: [1]).....	54
Abbildung 4.20: Layer Integration, Top-down Konfiguration (Quelle: [1])	54
Abbildung 4.21: Generische Stern Client/Server Architektur (Quelle: [1])	56
Abbildung 4.22: Client/Server Integration, Stufen und Konfigurationen (Quelle: [1]).....	57
Abbildung 4.23: 3-Schichten Client/Server Architektur (Quelle: [1]).....	57
Abbildung 4.24: Beispiel einer verteilten Architektur (Quelle: [1]).....	61

Abbildung 4.25: Schritte und Konfigurationen einer Integration von verteilten Remote Hosts (Quelle: [1]).....	62
Abbildung 4.26: Backbone Integration, 1. Schritt (Quelle: [1])	64
Abbildung 4.27: Backbone Integration, 2. Schritt (Quelle: [1])	65
Abbildung 4.28: Backbone Integration, 3. und 4. Schritt (Quelle: [1])	65
Abbildung 4.29: Backbone Integration, 5. und 6. Schritt (Quelle: [1])	66
Abbildung 4.30: Backbone Integration, 7. Schritt (Quelle: [1])	66
Abbildung 7.1: JUnit – erfolgreicher Testdurchlauf	79
Abbildung 7.2: JUnit – Testdurchlauf nicht erfolgreich	80
Abbildung 5.1: Testergebnisse JUnitEE (Quelle: [8])	81
Abbildung 5.2: Ergebnis Kompilierungsprozess CruiseControl (Quelle: [9]).....	83
Abbildung 5.3: Hauptfenster qftestJUI (Quelle: [25])	84
Abbildung 5.4: Hauptfenster Exacum (Quelle: [11]).....	85

Tabellenverzeichnis

Tabelle 2.1: Abdeckungsmaße für Programmflussgraphen (Quelle: [2])	21
Tabelle 4.1: On- und Off-Punkte für die <code>CustomerProfile</code> Invarianz	33
Tabelle 4.2: Auszug aus einer Domainmatrix für Invariant Boundaries	33
Tabelle 4.3: Generische quasi-modale Zustände	36
Tabelle 4.4: Generische quasi-modale Events	37
Tabelle 4.5: Zusammenfassung der Zustandsfehler und mögliche Gründe	39
Tabelle 4.6: Beispiel für eine Liefergegenstandskheckliste beim Abnahmetest (Quelle: [19])	74
Tabelle 5.1: Übersicht Testtools	75
Tabelle 7.1: Mögliche Kriterien für einen Umgebungstest	91
Tabelle 7.2: Mögliche Checkliste für den Abnahmetest einer Java-Applikation	92

1 Einleitung

1.1 Motivation

Durch ständige Weiterentwicklungen werden Softwareprodukte immer komplexer. Mit steigender Komplexität steigen jedoch auch mögliche Fehlerquellen. In den meisten Fällen werden in der Implementierungsphase nur ca. ein Drittel aller Programmierfehler gefunden und korrigiert.

Für die Gewährleistung einer hohen Fehlerfreiheit muss ein erheblicher Teil der Entwicklungszeit für den Softwaretest verwendet werden. Um effektiv zu testen, sollte ein Testablauf eingehalten werden, um alle möglichen Fehlerquellen zu überprüfen und gegebenenfalls zu korrigieren.

Objektorientierte Sprachen haben einen anderen Aufbau und eine andere Struktur als prozedurale Sprachen. Diese neuen Konzepte bringen mit Sicherheit einige Vorteile bei der Implementierung mit sich, allerdings beinhalten Sie auf der anderen Seite auch neue Fehlerquellen, die beim Testen berücksichtigt werden müssen.

Die objektorientierte Programmiersprache Java wird aufgrund ihrer vielfältigen Anwendungsmöglichkeiten heute häufig genutzt. Für die Fehlersuche existieren diverse Testtools auf dem Markt, die jedoch immer nur einen Teilbereich des Softwaretests abdecken. Das Ziel sollte es sein, mit einem Tool den gesamten Testablauf durchzuführen und die Ergebnisse der einzelnen Testphasen zu dokumentieren.

1.2 Aufbau der Arbeit

Die vorliegende Diplomarbeit stellt einen Versuch dar, ein Konzept für ein integriertes, Java-basiertes Testtool unter Bewertung bereits vorhandener Tools zu erstellen.

Im 2. Kapitel wird auf die allgemeinen Testmethoden eingegangen. Diese unterteilen sich in dynamische und statische Testmethoden, sowohl für prozedurale als auch für objektorientierte Software. Bei den statischen Testmethoden werden das Checklistenverfahren, die symbolische Programmierung sowie die Programmverifikation kurz erläutert. Bei den dynamischen Testmethoden wird auf den Black Box Test und den White Box Test eingegangen.

Das 3. Kapitel befasst sich mit den Besonderheiten beim Testen von objektorientierter Software.

Das nächste Kapitel befasst sich mit dem Testablauf. Hierzu gehören der Unittest, der Integrationstest, der Systemtest sowie der Abnahme-/Akzeptanztest.

Im 5. Kapitel wird auf ausgewählte Testtools eingegangen. Für den Unittest und den Integrationstest kann JUnit verwendet werden. JUnitEE und CruiseControl sind Beispiele für Tools, die nur den Integrationstest unterstützen.

Im 6. Kapitel wird auf die Besonderheiten der Programmiersprache Java in Bezug auf das Testen eingegangen.

Das 7. Kapitel stellt ein Konzept für ein integriertes Java-Testtool vor.

Im letzten Kapitel erfolgt eine Zusammenfassung der Arbeit.

2 Testmethoden

Das Testen einer Software dient der Fehlerfindung und somit dem Nachweis der Qualität einer Software. Nach [10] versteht man unter Softwarequalität die "Gesamtheit der Merkmale von Software, die erforderlich sind, dass sie die Funktionen, die sie für ihren sicheren und fehlerfreien Einsatz benötigt, erfüllt. Die Qualität von Software wird bestimmt durch den Entwicklungsprozess (Personal, Technologie, Management). Die Merkmale sollten voneinander unabhängig sein und gemeinsam alle relevanten Aspekte abdecken."

Durch einen Test kann jedoch nicht die 100%ige Fehlerfreiheit einer Software nachgewiesen werden.

Das Testen einer Software kann statisch und dynamisch erfolgen. Beim statischen Testen wird das Programm selber nicht ausgeführt. Die dynamischen Testmethoden beziehen sich auf das Programm in seiner abarbeitungsfähigen Form.

2.1 Statische Testmethoden

Statische Testmethoden beziehen sich auf den Quellcode eines Programms. Das Checklistenverfahren, die symbolische Programmtestung sowie die Programmverifikation werden im Folgenden kurz erläutert.

2.1.1 Checklistenverfahren

Beim Checklistenverfahren wird der Quellcode des Programms nach einem vorgegebenen Muster geprüft. Als Muster dienen Checklisten. „Inhalte dieser Prüfungen können die Einhaltung einfacher syntaktischer Regeln, wie beispielsweise die paarweise Verwendung der Klammerung, oder aber auch im Quellcode erkennbare, semantische Fehler bei der Abarbeitung des Programms sein“. (Quelle: [2])

Fragen für eine Checkliste können unter anderem sein:

- Werden Berechnungen mit Variablen unterschiedlichen Datentyps ausgeführt?
- Erfolgt ein Vergleich von Daten unterschiedlichen Datentyps?
- Kommt jede Schleife zum Ende?
- Kann eine Division durch Null vorkommen?

Von dem Ergebnis der Checkliste ist der weitere Testaufwand des Programms abhängig.

Ein Vorteil des Checklistenverfahrens ist, dass alles das geprüft wird, was auf der Liste steht. Dies ist aber wiederum auch ein Nachteil, da mit hoher Wahrscheinlichkeit nichts abgeprüft wird, was nicht auf der Liste zu finden ist. Als weiterer Nachteil ist die Fülle der Informationen, die abgeprüft werden, anzusehen.

2.1.2 Symbolische Programmtestung

Eine weitere statische Testmethode ist die symbolische Programmtestung, welche einer Ablaufverfolgung des Programms mit Symbolen entspricht.

Beispiel:

```

1     summe (a, b, c)
2     {
3     x = a + b;
4     y = b + c;
5     z = x + y - b;
6     return(z);
7     }
```

Bei einer symbolischen Belegung der Eintrittsgrößen müsste das obige Programm folgendes Ergebnis liefern:

$$\text{summe (a} \equiv \alpha, \text{ b} \equiv \beta, \text{ c} \equiv \gamma) \rightarrow z = (\alpha + \beta + \gamma)$$

Die symbolische Abarbeitung:

Anweisung	a	b	c	x	y	z
1	α	β	γ	?	?	?
3	-	-	-	$\alpha + \beta$	-	-
4	-	-	-	-	$\beta + \gamma$	-
5	-	-	-	-	-	$(\alpha + \beta) + (\beta + \gamma) - \beta$
6	-	-	-	-	-	$\text{return}(\alpha + \beta + \gamma)$

Die Korrektheit ist durch die Übereinstimmung mit dem geforderten Ergebnis gezeigt.

2.1.3 Programmverifikation

Zum Abschluss der statischen Testmethoden wird noch kurz auf die Programmverifikation eingegangen.

„Die Programmverifikation ist die formale Überprüfung, ob eine Realisierung eines Algorithmus als Programm mit der Algorithmenbeschreibung (im allgemeinen Spezifikation genannt) übereinstimmt.“ (Quelle: [7])

Bei der Programmverifikation wird eine Programmanweisung in der in Abbildung 2.1 beschriebenen Weise betrachtet:

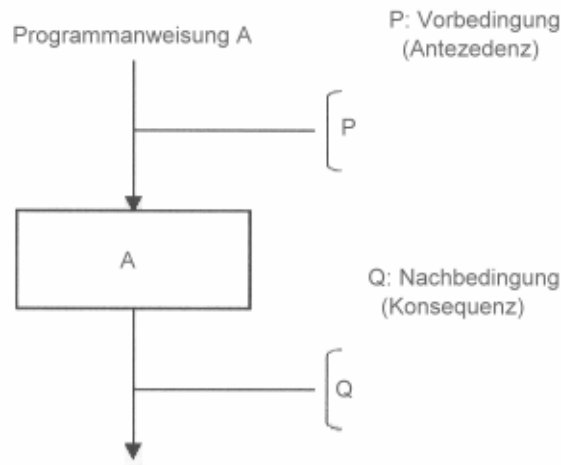


Abbildung 2.1: Vor- und Nachbedingungen bei einer Programmanweisung (Quelle: [2])

Die Vorbedingungen und Nachbedingungen können auch als Assertions (Zusicherungen) bezeichnet werden. Die Art der Assertions ist vom Paradigma abhängig. Unter einem Paradigma versteht man bei der Softwareentwicklung „die durch formale und informale Methoden auf der Modellierungsseite und durch eine Programmiersprachform auf der Realisierungsseite geprägte Art der Systemdarstellung und -implementation“ (Quelle: [2]). Das folgende Beispiel bezieht sich auf ein imperatives Paradigma, welches auf der strukturierten Programmierung beruht. Bei der strukturierten Programmierung wird das Programm in Prozeduren und Funktionen unterteilt.

Unter P versteht man die Vorbedingung, Q ist die Nachbedingung. Die Zusicherungsangaben lauten mit A_i als Anweisung und B als logischem Ausdruck:

- Sequenz

$$\{P_0\} A_1; \{P_1\} A_2 \{P_2\}; \dots; A_n \{Q\} \text{ mit } Q = P_0 \wedge P_1 \wedge \dots \wedge P_n$$

- Selektion

$$\{P_0\} \text{ case I of } A_1; A_2; \dots; A_n \text{ mit } Q = P_0 \wedge P_1$$

und speziell

$$\{P_0\} \text{ if B then } A_1 \{P_1\} \text{ else } A_2 \{P_2\}$$

$$\text{Mit } Q = (P_0 \wedge B \wedge P_1) \vee (P_0 \wedge \neg B \wedge P_2)$$

- Iteration

$$\{P_0\} \text{ while B; } A_1 \{Q\} \text{ mit } Q = P_0 \wedge \neg B \wedge P_1$$

sowie

$$\{P_0\} \text{ until B; } A_1 \{Q\} \text{ mit } Q = P_0 \wedge B \wedge P_1$$

Nach Dumke[2] wird die Programmverifikation wie folgt gezeigt:

- (1) „Im Quellprogramm wird nach jeder Anweisung die unmittelbar nach dieser Anweisung geltende Nachbedingung eingetragen.“
- (2) Innerhalb eines Zyklus unterteilen sich diese Nachbedingungen in Varianten und Invarianten. Diese Unterteilung ist wichtig, da nur sich im Zyklus verändernde Größen

(Varianten) für die Beendigung dieser Strukturform die Grundlagen bilden können. Neben den Berechnungs- und Verarbeitungsinhalten müssen wir beim Zyklus natürlich auch dessen Terminierung beweisen.

- (3) Es wird der durch das Programm zu leistende Zielausdruck in der obigen, aussagenlogischen Form postuliert.
- (4) Im Programm werden die Assertions schrittweise interpretiert und in der oben beschriebenen Form logisch verknüpft. Zyklen erfordern zumeist eine induktive Beweisführung.
- (5) Das Interpretationsergebnis wird mit dem Postulat verglichen und somit die formale Korrektheit bewiesen bzw. verifiziert.“

Im Folgenden wird ein einfaches Beispiel für eine Programmverifikation angegeben. Es sollen alle Anweisungen in einem Quellcode gezählt werden, die eine Bedingung bzw. bedingte Verzweigung enthalten. Dabei wird angenommen, dass im analysierten Quelltext nur eine bedingte Verzweigung pro Quelltextzeile auftreten kann. Daraus entsteht der folgende Quellcode mit den dazugehörigen Assertions:

Programmbeschreibung	Assertions
Beispiel seq	
Anzahl := 0	Anzahl _j = 0; j = 1
Zeilenr := 0	Zeilenr _i = 0; i = 1
Zyklus iter until (Zeilenr = n)	Zeilenr _i = n?
Nächste Zeile einlesen	Zeile _i definiert
Suche select (in Zeile ‘if’, ‘while’, ‘case’, ‘for’, ‘until’ oder ‘repeat’ vorhanden)	Verzweigungsanweisung in Zeile (genannt “Bedg”) ?
Anzahl := Anzahl + 1	Anzahl _{j+1} = Anzahl _j + 1
Suche end	
	Invariante:
	Anzahl _{j+1} = Anzahl _j + $\begin{cases} 1 & \text{wenn Bedg} \\ 0 & \text{sonst} \end{cases}$
Zeilenr := Zeilenr + 1	
	Variante:
	Zeilenr _{i+1} = Zeilenr _i + 1 mit 1 ≤ i ≤ n
Zyklus end	
Anzahl ausgeben	Anzahl _j drucken
Beispiel end	

Durch den obigen Algorithmus soll der Zielausdruck

$$\text{Anzahl} = \text{Anz}(\text{Bedg in Zeile}_i), 1 \leq i \leq n$$

realisiert werden.

Der folgende Beweis zeigt, dass das oben im Pseudocode dargestellte Programm genau diese Berechnungen realisiert. Da das Programm aus einer Sequenz von drei Anweisungsteilen besteht (Anfangswertfestlegung, Berechnungszyklus und Ausgabe) wird der Beweis nach Dumke [2] in 3 Schritten geführt:

1. Nach den ersten beiden Anweisungen 'Anzahl := 0' und 'Zeilenr := 0' gilt $\text{Anzahl}_1 = 0$ und $\text{Zeilenr}_1 = 0$
2. Beim Zyklus wird die Beweistechnik der vollständigen Induktion verwendet. Dabei werden zum einen das Berechnungsergebnis und zum anderen die Terminierung für den Zyklus gezeigt.

a. Anzahlberechnung:

- Erster Zyklusdurchlauf:

wenn „Bedg“ in Zeile, dann $\text{Anzahl}_j = 0 + 1 = 1$

wenn „Bedg“ nicht in Zeile, dann Anzahl_j unverändert ($= 0$)

- Induktionsannahme:

$$\text{Anzahl}_{j+1} = \text{Anzahl}_j + \begin{cases} 1 & \text{wenn Bedg} \\ 0 & \text{sonst} \end{cases} \quad \text{bis Zeile}_i \text{ für } 1 \leq i \leq n - 1$$

- Induktionsbehauptung:

$$\text{Anzahl}_{j+2} = \text{Anzahl}_{j+1} + \begin{cases} 1 & \text{wenn Bedg} \\ 0 & \text{sonst} \end{cases} \quad \text{bis Zeile}_{i+1} \text{ für } 1 \leq i \leq n - 1$$

- Induktionsbeweis:

aus der Interpretation des Zyklusdurchlaufens erhält man:

wenn „Bedg“ in Zeile_{i+1}, dann $\text{Anzahl}_{j+2} = \text{Anzahl}_{j+1} + 1$,

sonst $\text{Anzahl}_{j+2} = \text{Anzahl}_{j+1} + 0$

q.e.d.

b. Terminierung:

Hierbei ist zu zeigen, dass $\text{Zeilenr}_i = n$ erfüllt werden kann.

- Erster Zyklusdurchlauf:

das Einlesen der ersten Zeile beginnt mit der $\text{Zeilenr}_1 = 0$ und schließt mit dem Inkrementieren für Zeilenr_1 ab; das heißt es ergibt sich $\text{Zeilenr}_2 = 1$

- Induktionsannahme:

nach $n - 1$ eingelesenen Zeilen gilt $\text{Zeilenr}_n = n - 1$

- Induktionsbehauptung:

nach dem Einlesen einer weiteren, der letzten Zeile hat Zeilennr_{n+1} den Wert n

- Induktionsbeweis:

der Durchlauf durch den Zyklus inkrementiert den Zeilenindex $n - 1$ und ergibt somit $(n - 1) + 1 = n$

q.e.d.

3. Der letzte Teil besteht aus der Ausgabe. Der durch den Zyklus bestimmte Ausdruck ist das Endergebnis, da keinerlei Wertveränderung vorgenommen wird.

Damit steht die Gleichheit fest

$$Anzahl = \underset{i}{Anz (Bedg \text{ in Zeile}_i)} \equiv Anzahl_{j+1} \equiv Anzahl_j + \begin{cases} 1 & \text{wenn Bedg} \\ 0 & \text{sonst} \end{cases}, 1 \leq j \leq n$$

Mit dem obigen Beweis wird die formale Korrektheit des Programms nachgewiesen. Bei einer realen Programmabarbeitung können folgende Probleme bzw. Fehler auftreten:

- Aufgrund von Zahlenkonvertierungen können Rundungsfehler auftreten. Dies tritt jedoch insbesondere bei der gleichzeitigen Verarbeitung von ganzzahligen und reellen Zahlen auf. Reelle Zahlen können nur näherungsweise dargestellt werden, ganze nur in einem speziellen Intervall.
- Bei einem Zeichenkettenvergleich können zum Beispiel die Groß- und Kleinschreibung nicht ausreichend beachtet werden. Auch ein unzureichendes Ignorieren ist möglich.
- Die Ausgabeanweisung kann falsche Formatangaben enthalten, die ein unzuverlässiges abschneiden, runden oder konvertieren des Ergebnisses zur Folge haben kann.
- Durch eine Transformation durch einen Optimierungs-Compiler kann sich die ursprüngliche Form des Quelltextes verändern.

Trotzdem werden durch die Programmverifikation Fehler in der Programmlogik eingeschränkt.

2.2 Dynamische Testmethoden

Bei den dynamischen Testverfahren wird das Programm als Programmrealisierung, das heißt in seiner abarbeitungsfähigen Form, getestet. Hierzu gehören, als die am häufigsten vorkommenden, der Black Box Test und der White Box Test.

2.2.1 Black Box Test

Der Black Box Test wird auch als Funktionstest bezeichnet. Beim funktionalen Test können nicht realisierte Teile der Spezifikation entdeckt werden. Das Programm wird hierbei als geschlossene Einheit betrachtet und liefert aufgrund spezieller, vorgegebener Eingabewerte Ausgabewerte. Die speziellen Eingabewerte sind Testdaten, die aus der Spezifikation bzw.

dem Entwurf abgeleitet werden. Als Testdaten dienen Normalwerte, Extremwerte und Falschwerte. Die Programmstruktur ist für diesen Test irrelevant. Da nicht alle möglichen Testfälle realisiert werden können, sind Fehler nicht auszuschließen.

2.2.1.1 Äquivalenzklassenbildung

Da ein Programm nicht für alle möglichen Werte einer Eingabegröße getestet werden kann, wird der Wertebereich einer Eingabegröße in Teilintervalle unterteilt. Diese Teilintervalle sind äquivalent, das heißt sie besitzen eine gleichartige Eigenschaft. Die äquivalenten Teilbereiche werden auch als Äquivalenzklassen bezeichnet. Beispiele für Äquivalenzklassen sind:

Wertebereich	1. Teilbereich (Klasse gültiger Werte)	2. Teilbereich (Klasse ungültiger Werte)
Ganze Zahlen $a \dots n$	$a \leq \text{Wert} \leq n$	Wert $< a$ oder Wert $> n$
Buchstabenfolge	aBcDeF	A123B

2.2.1.2 Grenzwertanalyse

Die Grenzwertanalyse dient der Analyse der Grenzdaten.

Bei Integerwerten zum Beispiel sollte an den Grenzen immer der Minimumwert -1, der Minimumwert und der Minimumwert +1 getestet werden. Das gleiche gilt für den Maximumwert -1, den Maximumwert und den Maximumwert +1.

Je nach definierter Variable und abhängig von der Prozessorarchitektur müssen geeignete Testfälle entworfen werden, die einen Überlauf bei Integeroperationen überprüfen.

2.2.1.3 Ursache-Wirkungsgraph

Sowohl die Grenzwertanalyse als auch die Äquivalenzklassenbildung können keine Kombination von Eingabebedingungen untersuchen. Trotz Unterteilung der Eingabekombinationen in Äquivalenzklassen wird die Anzahl an Kombination sehr hoch. Um einen effizienten Test zu erhalten, sollte eine sinnvolle Auswahl an Eingabebedingungen erfolgen.

Der Ursache-Wirkungsgraph ist hierbei sehr hilfreich.

Unter einem Ursache-Wirkungsgraph versteht man eine formale Sprache, in die die Spezifikation übersetzt wird.

Nach [6] erfolgt die Entwicklung der Testfälle in folgender Form:

- Zerlegung der Spezifikation in kleine, handliche Stücke.
- Die Ursachen (Eingabebedingungen oder Äquivalenzklassen von Eingabebedingungen) und die Wirkungen (Ausgabebedingungen) der Spezifikation werden festgelegt.
- Der semantische Inhalt der Spezifikation wird analysiert und in einen booleschen Graphen, dem Ursache-Wirkungsgraphen, transformiert, der die Ursachen und Wirkungen verbindet. Bei einem booleschen Graphen werden den Knoten logische Verknüpfungen wie 'und', 'oder' oder die Negation zugeordnet.
- Kombinationen von Ursachen und/oder Wirkungen sind manchmal auf Grund syntaktischer oder kontextabhängiger Beschränkungen nicht möglich. Diese werden als Kommentare im Graph vermerkt.

- Der Ursache-Wirkungsgraph wird in eine Entscheidungstabelle umgesetzt. Jede Spalte der Tabelle stellt einen Testfall dar.
- Die Spalten der Entscheidungstabelle werden in Testfälle umgesetzt.

2.2.2 White Box Test

Im Gegensatz zum Black Box Test, bei dem die Testfälle so gewählt werden, dass möglichst kritische Eingabewerte getestet werden, erfolgt beim White Box Test die Wahl der Testfälle so, dass alle Befehle und Befehlsfolgen systematisch durchlaufen werden. Das Ziel hierbei ist, alle möglichen Pfade im Programm zu durchlaufen. Die Testdaten ergeben sich aus dem Quellprogramm bzw. aus seiner abstrakten Form als Programmflussgraph.

Ein Beispiel mit Pseudocode (Quelle [2]):

```

Beispiel      Seq
              Eingabe: N, L, R
              Zähler := 1
              Summe := 0
              Zyklus iter until Zähler = N
                  Eingabe: Zahl
                  Auswahl select L < Zahl < R
                      Summe := Summe + Zahl
                  Auswahl end
                  Zähler := Zähler + 1
              Zyklus end
              Ausgabe: Summe
Beispiel      end

```

Der Programmflussgraph stellt die Anweisungen als einfache Knoten dar. Die Anweisungen sind durch gerichtete Kanten gemäß dem Steuerfluss miteinander verbunden.

Der Programmflussgraph für das oben angeführte Beispiel hat folgende Form:

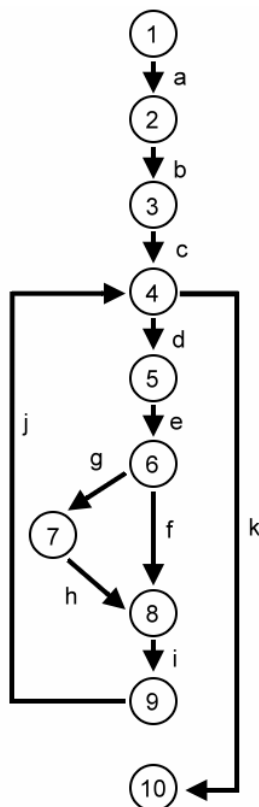


Abbildung 2.2: Programmflussgraph (Quelle: [2])

Abdeckungsmaße beschreiben die Vollständigkeit der Testaktivitäten. Auf Modulebene erfolgt die Abdeckung der Verzweigungen, auf Datenebene das Benutzen aller Daten in Testfällen und auf Programmebene das Benutzen aller Aufrufe aller Module.

Nach Dumke[2] gibt es folgende Abdeckungsmaße für Programmflussgraphen:

<i>Abdeckungsart</i>	<i>Bedeutung</i>
C0	Jede Anweisung wird einmal ausgeführt
C1	Jeder Zweig wird einmal durchlaufen
C2	Jeder Bedingungsteil wird einmal ausgeführt
C3	Alle Schleifen werden mehrmals wiederholt
C4	Alle unabhängigen Pfade werden wiederholt
C5	Alle unabhängigen Pfade werden ausgeführt
C6	Alle Vorwärtspfade werden ausgeführt
C7	Sämtliche Pfade werden ausgeführt

Tabelle 2.1: Abdeckungsmaße für Programmflussgraphen (Quelle: [2])

Für das oben angeführte Beispiel bedeutet dies:

- C0: Alle Knoten von 1 bis 10 werden einmal durchlaufen.
- C1: Zusätzlich zu C0 muss der Pfad f getestet werden.

- C2: In Anweisung 6 wird die Bedingung $L < \text{Zahl} < R$ gestellt. Da jeder Bedingungs-
teil einmal ausgeführt wird, muss sowohl $L < \text{Zahl}$ als auch $\text{Zahl} < R$ getestet werden.
- C3: Es werden mehrere Zahlen eingegeben, um einen mehrmaligen Zyklendurchlauf
zu erreichen. Hierdurch wird die Zyklensteuerung getestet.
- C4: Zu den unabhängigen Pfaden im Beispiel gehören unter anderem:
1 → 2 → 3 → 4 → 10 und
1 → 2 → 3 → 4 → 5 → 6 → 8 → 9 → 4 → 10
Bei dieser Abdeckungsart sind beide Pfade mehrmals wiederholt zu testen.
- C5: Die Ausführung der unabhängigen Programmpfade erfordert die paarweise Tes-
tung aller im Programm auftretenden Kombinationen dieser Pfade.(Quelle: [2])
- C6: Bei C6 sind alle möglichen Pfadkombinationen zu testen, wobei bei Programms-
prüngen nur die Vorwärtsrichtung zu berücksichtigen ist.
- C7: C7 beinhaltet die Testung aller möglichen Programmpfade.

3 Besonderheiten/Probleme beim Testen objektorientierter Software

Im Gegensatz zu konventionellen Systemen sind bei objektorientierten Systemen einige Besonderheiten zu beachten. Im folgenden Kapitel wird zunächst auf die Merkmale objektorientierter Systeme eingegangen. Daran schließen sich die Testgegenstände an. Anschließend wird auf die Besonderheiten beim Testen objektorientierter Systeme eingegangen.

3.1 Merkmale objektorientierter Software

Grundlage objektorientierter Software ist die Unterteilung des Gesamtsystems in einzelne Module. Ein Modul ist eine inhaltliche oder funktionale Einheit, die mehrfach verwendet werden kann. Diese Mehrfachverwendung soll vor allem zu einer Verringerung der Coderedundanz führen, wie sie in herkömmlicher, funktional programmierter Software häufig auftritt.

Das kleinste, wiederverwendbare Modul bei objektorientierter Software ist die *Klasse*. Eine Klasse stellt Attribute und Methoden bereit, die den Zustand und das Verhalten von *Objekten* definieren. Ein Objekt ist eine konkrete *Instanz* einer Klasse.

Bei objektorientierten Systemen sind die Methoden in der Regel so programmiert, dass die Klasse jeden potentiellen Zweck erfüllen kann. Damit sind diese funktional oft umfangreicher als benötigt. Da nicht alle Methoden von einer Anwendung genutzt werden, ist es nicht erforderlich, den gesamten Quellcode einer Klasse zu analysieren. Es reicht aus, die Methoden zu testen, die von der jeweiligen Anwendung verwendet werden.

Ein Objekt kann viele verschiedene Zustände annehmen. „Je komplexer ein Objekt, desto mehr Attribute und Methoden, desto mehr Zustände“ (Quelle: [13]) existieren. Für ein Objekt müssen alle Zustände und Zustandsübergänge getestet werden. Die Anzahl der Testfälle wächst somit exponentiell mit den verwendeten Attributen und Methoden.

Da die Ausführung einer Methode vom jeweiligen Objektzustand abhängen kann und dieser Zustand während der Abarbeitung auch verändert werden kann, entstehen Abhängigkeiten beim Aufruf von Methoden. Diese Abhängigkeiten sorgen für einen erhöhten Testaufwand, da eine Vielzahl möglicher Aufrufsequenzen getestet werden müssen.

Weiterhin hat die Modularisierung objektorientierter Systeme intermodulare Abhängigkeiten zur Folge. Durch Mechanismen wie Vererbung und die Verwendung öffentlicher Methoden werden Bezüge zwischen verschiedenen Modulen hergestellt, welche beim Testen berücksichtigt werden müssen.

Die Verwendung von „*Polymorphie* und *dynamischer Bindung*“ führen zu einer Vermehrung der potentiellen Ablaufpfade und damit der potentiellen Fehler“ (Quelle: [13]). Durch Vererbung werden unsichtbare und subtile Abhängigkeiten geschaffen. Die „Kapselung beschränkt die Sicht auf die Objektzustände und erschwert dadurch die Fehlererkennung. Zahlreiche Kollaborationen zwischen Objekten schaffen viele Schnittstellen, die wiederum zu Fehlern führen können“ (Quelle: [13]).

3.2 Testgegenstände

Bei objektorientierten Systemen sollten Methoden, Klassen, Komponenten, Pakete sowie das Gesamtsystem getestet werden.

Methoden sollten aufgrund ihrer klar definierten Eingangs- und Ausgangsdaten einzeln testbar sein. Nach [13] wird dieser unabhängige Test jedoch durch Wechselwirkungen zwischen den Methoden und deren Abhängigkeit vom Zustand der nächst größeren Einheit, der Klasse, erschwert.

Klassen lassen sich über den jeweiligen Zweck, den sie erfüllen, definieren. Sie können bezüglich ihrer Schnittstellen einzeln getestet werden. Schnittstellen schreiben vor, welche Operationen eine Klasse zur Verfügung stellen muss. Sie besitzen ausschließlich abstrakte Methoden, so dass die abgeleiteten Klassen alle Methoden überschreiben müssen. Unter Paketen versteht man eine Menge von Klassen.

Komponenten sind "Klassenmengen, in denen nur wenige Abhängigkeiten zu Klassen in anderen Komponenten bestehen (sollten)" (Quelle: [13]). Daher sind Komponenten besonders gut zum Testen geeignet. Komponenten bieten nach außen eine Schnittstelle an.

3.3 Kapselung

Die Kapselung macht dem Benutzer eines Objektes nur diejenigen Datenelemente und Methoden zugänglich, die für ihn relevant sind. Alle anderen bleiben für ihn verborgen. Die meisten objektorientierten Programmiersprachen verwenden hierfür Zugriffsklassen, welche die Zugriffsrechte für die Attribute und Methoden einer Klasse steuern.

Die meisten objektorientierten Programmiersprachen haben zur Steuerung der Zugriffsrechte Zugriffsklassen.

Als Beispiel soll auf die Zugriffsklassen der Programmiersprache Java genauer eingegangen werden. Java bietet zur Steuerung der Zugriffsrechte insgesamt vier verschiedene Zugriffsklassen. Drei von ihnen werden durch einen Modifizierer angegeben. Die Vierte ist voreingestellt, wenn explizit keine Zugriffsklasse angegeben wird:

- Die Zugriffsklasse *public* erlaubt unbegrenzten Zugriff. Von außen und von allen Unterklassen aus kann auf das Datenelement oder die Methode zugegriffen werden.
- Bei der Zugriffsklasse *protected* ist der Zugriff auf Klassen grundsätzlich möglich, wenn er aus dem Paket erfolgt, in dem die Klasse definiert ist. Das gilt sowohl für Unterklassen als auch für Nicht-Unterklassen. Weiterhin ist der Zugriff von Unterklassen aus anderen Paketen erlaubt.
- Die Zugriffsklasse *private* ist das Gegenteil von *public*. Auf private-Bestandteile kann nur in der Klasse zugegriffen werden, in der sie definiert sind. Es besteht kein Zugriff von Objekten, die nicht zu dieser Klasse gehören. Andere Exemplare der Klasse dürfen aber zugreifen, das heißt auf ein *private*-Datenelement darf von einem anderen Exemplar aus zugegriffen werden. Da *private*-Bestandteile nicht vererbt werden, sind sie nicht in Unterklassen definiert. Definiert eine Unterklasse eine Methode, deren Kopf formal mit dem einer *private*-Methode der Oberklasse übereinstimmt, so gilt dies als Neudefinition und nicht als überschreiben.
- Wird keine Zugriffsklasse explizit angegeben, so wird die Zugriffsklasse *friendly* automatisch verwendet. Datenelemente und Methoden sind in dem Paket sichtbar, in

dem die Klasse definiert ist. Das gilt sowohl für Klassen als auch für Unterklassen. Ein Zugriff außerhalb des Pakets ist nicht möglich.

Die Kapselung hat den Vorteil, dass die Objekte abgeschlossen und die Schnittstellen fest definiert sind. Die Objekte können unabhängig von anderen Objekten getestet werden.

Die Kapselung hat in Bezug auf das Testen einige Nachteile. Es müssen alle Kombinationen von Methodenausführungen getestet werden, da die Ausführung von Methoden vom aktuellen Objektzustand abhängt und dieser wiederum von vorher ausgeführten Methoden abhängig sein kann. Aufgrund der Kapselung erweist sich die Kontrolle der Objektzustände schwierig. Mittels so genannter friend-Methoden in der Programmiersprache C++ ist es möglich, diese Kapselung zu umgehen, da diese Methoden die Kapselung wieder aufheben. „Eine als friend gekennzeichnete Funktion ist keine Methode der Klasse, hat aber das Recht, auf klassenprivate Attribute zuzugreifen. Dazu ist in der Klassendeklaration die „befreundete“ Funktion als friend zu deklarieren [...]“ (Quelle: [21]).

Auch ist eine Testung der Schnittstellen in allen Variationen erforderlich. Da diese meist sehr komplex sind, ist der Aufwand für den Test sehr hoch.

3.4 Vererbung

Bei der Vererbung werden von einer erbenden Klasse alle Merkmale der vererbenden Klasse geerbt. Die vererbende Klasse wird meist als Basisklasse oder Superklasse bezeichnet, die erbende als Subklasse oder abgeleitete Klasse.

Die Vererbung wird meist angewandt, wenn man mehrere Klassen schreiben will, die konzeptionell zusammengehören. In der Basisklasse werden die identischen Merkmale der Klassen zusammengefasst.

Abbildung 3.1 verdeutlicht die Vererbung in einem Unified Modeling Language (UML)-Diagramm. Der Pfeil bedeutet „erbt von“. Ist C eine Unterklasse von B und B eine Unterklasse von A, dann ist auch C eine Unterklasse von A. Die Vererbung erfolgt somit auch transitiv.

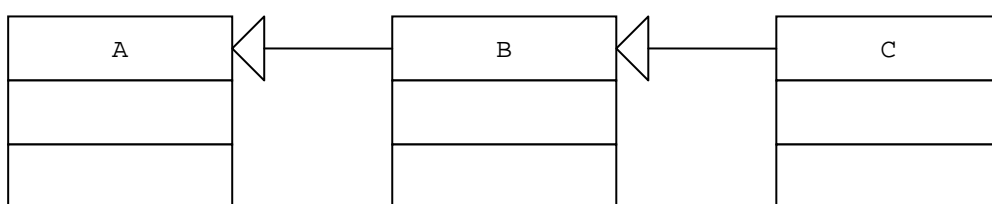


Abbildung 3.1: Beispiel Vererbung

Durch die Wiederverwendung wird Redundanz vermieden, da gemeinsame Codeteile in der Basisklasse nur einmal definiert werden.

Bei der Vererbung können unter anderem die folgenden Probleme auftreten:

- Bei der Vererbung können Fehler, die in den Basisklassen implementiert sind, auf Unterklassen übertragen bzw. vererbt werden. Deshalb ist es notwendig, die Basisklassen gründlich zu Testen, um eine Vererbung von Fehlern auszuschließen. Ist jedoch ein Fehler in einer Basisklasse behoben, so ist er in der Regel auch in den Subklassen bereinigt.

- Beim Testen sollte beachtet werden, dass Unterklassen nicht unabhängig von den Oberklassen getestet werden, da in der Klassenhierarchie Verweise zu der Oberklasse existieren. Als Beispielszenario dient noch einmal das in der Abbildung 3.1 dargestellte UML-Diagramm. C erbt von B und B erbt von A. Beim Testen von C müssen sowohl die Klasse B als auch die Klasse A mit einbezogen werden. Dies zeigt, dass es schwierig ist, Hierarchien hinreichend zu testen und somit hohe Anforderungen an den Testrahmen gestellt werden müssen.
- Als weiteres Problem ist anzusehen, dass die Struktur des Quelltextes nicht den Kontrollfluss widerspiegelt, „da Ablauflogik über Vererbungshierarchie verteilt ist“ (Quelle: [13]). Dies führt zu einer Erhöhung der Komplexität.
- Die Modifizierung der Basisklasse kann sich gravierend auf die Unterklassen auswirken. Es ist möglich, dass die Unterklassen neu angepasst werden müssen, und dieses zieht eine lange Kette von Veränderungen nach sich. Auch müssen sowohl die Basisklasse als auch alle erbenden Klassen erneut getestet werden.

3.4.1 Mehrfachvererbung

Bei der Mehrfachvererbung wird eine Klasse von mehr als einer Basisklasse abgeleitet. Objekte der erbenden Klassen können somit Objekte unterschiedlicher Oberklassen vertreten.

Abbildung 3.2 zeigt ein Klassendiagramm für eine Mehrfachvererbung. Die Klasse Wohnmobil erbt sowohl von der Klasse Auto als auch von der Klasse Wohnhaus, da sie die Eigenschaften von beiden Klassen besitzt.

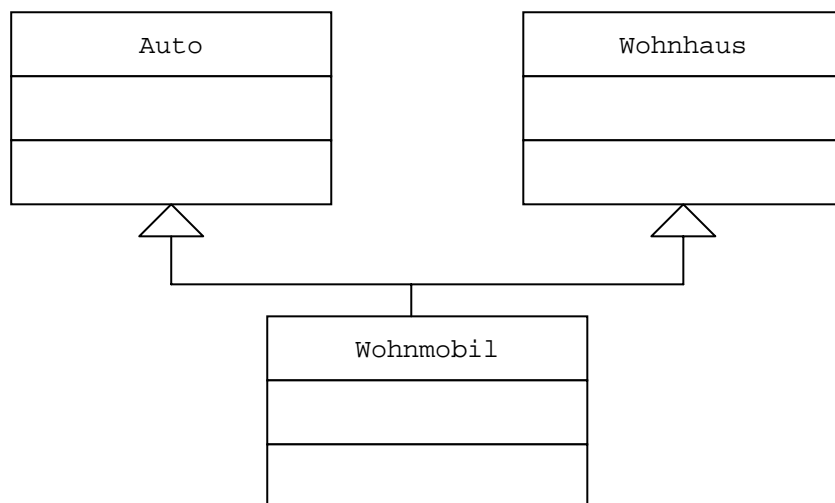


Abbildung 3.2: Beispiel Mehrfachvererbung

Nachteile können durch die Vererbung gleichnamiger Methoden entstehen. Klasse C erbt von Klassen A und B. C nutzt zunächst Methode A.m, wird dann aber mit B.m realisiert. Nach dieser Änderung muss die Klasse C erneut getestet werden. Weiterhin sind verschiedene Testfälle für A.m und B.m notwendig.

3.5 Abstrakte Klassen

Abstrakte Klassen sind einzigartig in den objektorientierten Programmiersprachen und leisten wertvolle Unterstützung für die Wiederverwendung.

Eine abstrakte Klasse unterstützt ein Interface ohne eine Implementierung. Abstrakte Klassen können (abstrakte) Methoden definieren, die die Subklassen erben. Die geerbten Methoden müssen von der Subklasse implementiert werden.

Der Nachteil von abstrakten Klassen ist, dass sie nicht instanziiert sind. Um ein konkretes Problem lösen zu können, muss eine konkrete Klasse abgeleitet werden.

3.6 Polymorphismus

„Polymorphie („Vielgestaltigkeit“) bedeutet, dass Funktionen bzw. Methoden einem bestimmten Typ bzw. einer bestimmten Klasse zugeordnet sind, und je nach Typ/Klasse eines bestimmten Objekts die zugehörige Funktion/Methode ausgeführt wird“ (Quelle: [24]). Somit wird die gleiche Nachricht von verschiedenen Objekten verstanden, aber von jedem Objekt anders interpretiert.

Als Beispiel für Polymorphie dient das in der Abbildung 3.3 dargestellte Klassendiagramm. Es wird eine Klasse `GeomFigur` definiert. Die Klassen `Kreis`, `Rechteck` und `Dreieck` erben von der Klasse `GeomFigur`. Die Methode `anzeigen()`, die in der Klasse `GeomFigur` definiert ist, wird polymorph behandelt, da jedes geometrische Objekt eine andere Form hat und somit auch jeweils eine andere Implementierung erfolgen muss, die die geometrische Figur anzeigt.

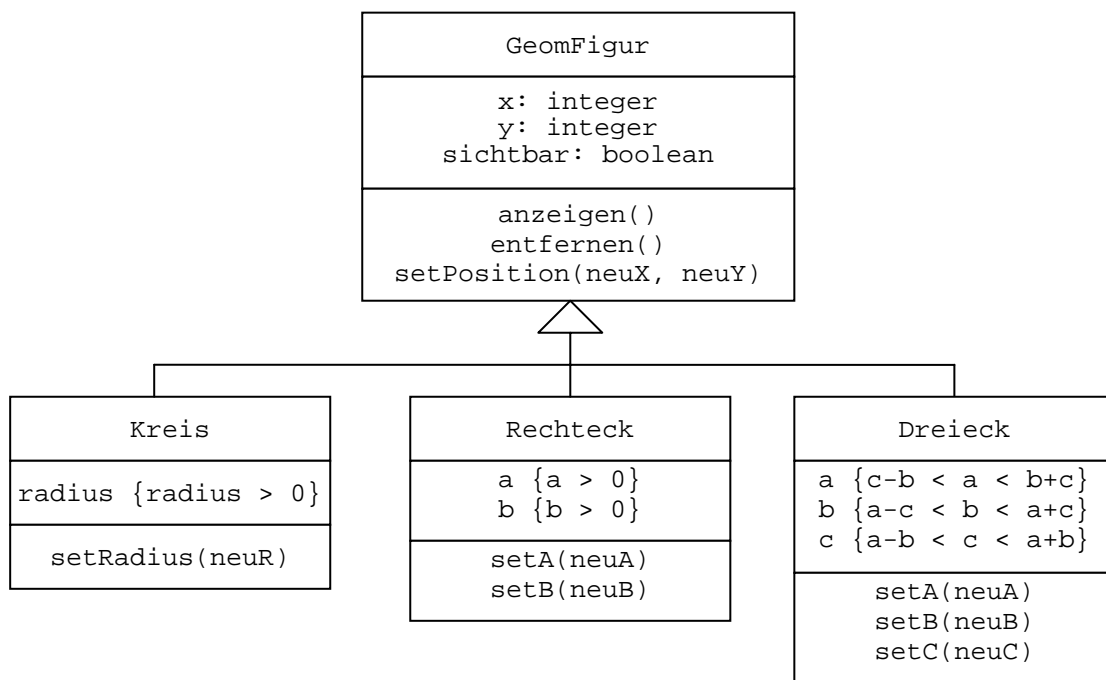


Abbildung 3.3: Beispiel Polymorphismus (Quelle: [12])

Bei der Polymorphie unterscheidet man zwischen dem statischen Binden und dem dynamischen Binden. Das statische Binden wird auch als Kompilationszeit-Polymorphie bezeichnet, das dynamische Binden als Laufzeit-Polymorphie. Die Art der Bindung wird durch die Programmiersprache festgelegt.

3.6.1 Statisches Binden

Beim statischen Binden muss ein Funktionsaufruf an eine Folge von auszuführenden Anweisungen gebunden werden. Der Aufruf einer Funktion (oder Methode, Operation) heißt statisch gebunden, wenn bereits der Compiler oder der Binder (Linker) die Funktion einbindet, also vor dem Programmstart.

Der Vorteil der statischen Bindung ist die schnelle Programmausführung im Gegensatz zur dynamischen Bindung, da der Programmablauf statisch festgelegt ist.

3.6.2 Dynamisches Binden

Bei der dynamischen Bindung bestimmt die Klasse des Objektes zur Laufzeit, welche Methode aufgerufen wird. Die Bindung erfolgt somit noch nicht zur Kompilierungszeit sondern erst zur Laufzeit. Der Programmablauf ist nicht mehr statisch aus dem Quelltext ableitbar. Bei der Entwicklung eines Tests weiß man unter Umständen nicht genau, welche Bedingungen zur Laufzeit auftreten können. Weiterhin existieren viele unterschiedliche Objekte mit unterschiedlichem internen Verhalten, aber einheitlicher Schnittstelle. Man weiß somit erst zur Laufzeit, welche Objekte verwendet werden und wie sie sich verhalten.

3.6.3 JoJo-Effekt

Der JoJo-Effekt verdeutlicht das Problem bei mehrfacher Wiederholung der Polymorphie. Mehrfache Polymorphie erschwert das Verständnis und die Nachvollziehbarkeit des dynamischen Programmverhaltens. Daraus resultieren wiederum eine Erhöhung von Fehlern sowie die Erschwerung der Entwicklung von Tests und die Lokalisierung der Fehler. Der JoJo-Effekt verdeutlicht den Versuch, Aufrufsequenzen nachzuvollziehen, wenn Methoden auf unterschiedlichen Hierarchieebenen überschrieben wurden.

Abbildung 3.4 zeigt die Aufrufsequenz einer Klassenhierarchie, bei der dieses Problem vorkommt:

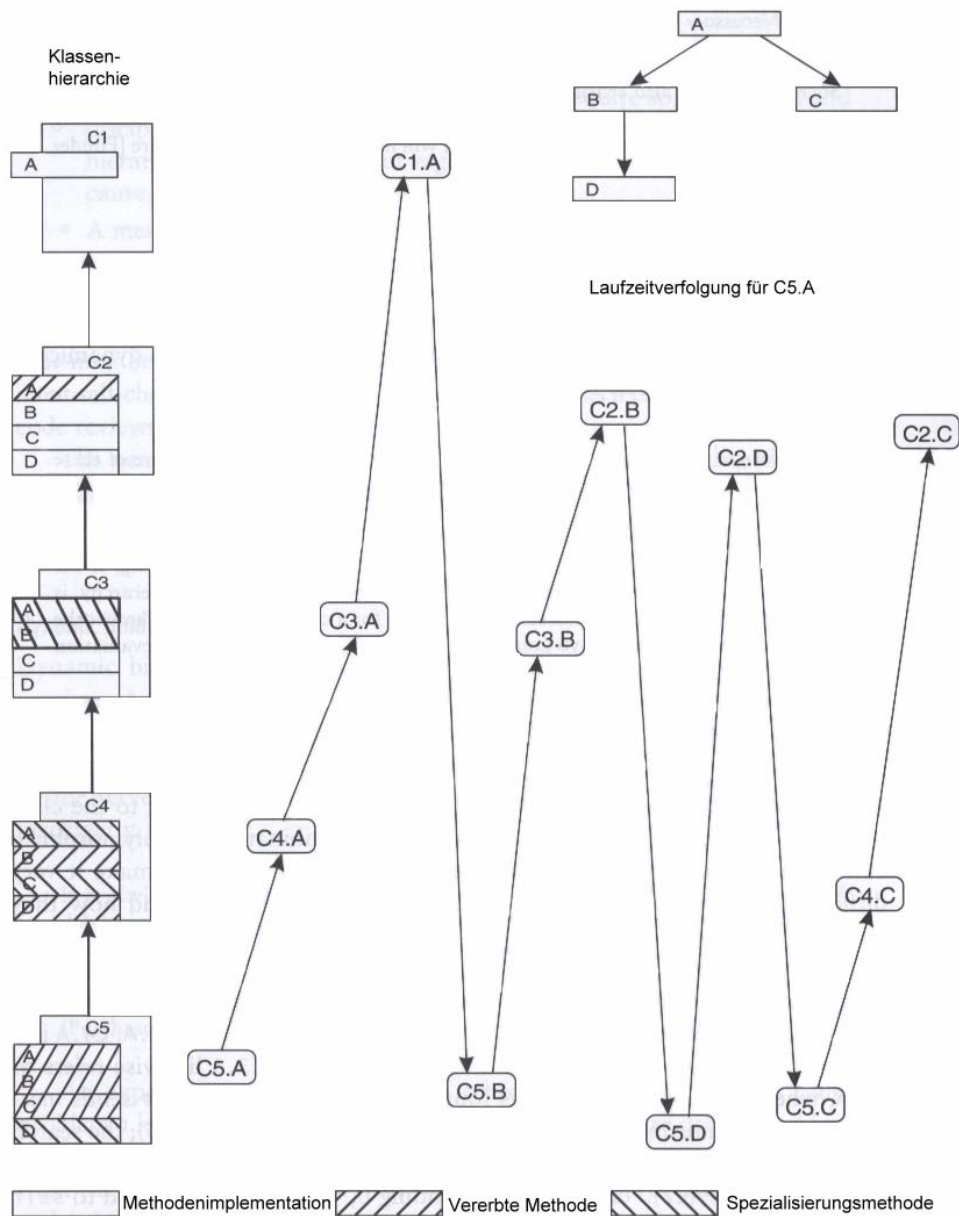


Abbildung 3.4: Ablaufverfolgung der JoJo Ausführung (Quelle: [1])

Die Methode A der Klasse C5 wurde von der Klasse C4 vererbt. C4.A ist eine Verfeinerung, das heißt bei C3.A wird nach einer Implementierung gesucht. C3.A verweist ebenso zu C1.A, wo die Implementierung gefunden und ausgeführt wird. Hieran erkennt man, dass es schwierig ist, die Aufrufsequenzen nachzuvollziehen.

4 Testablauf

Der Testablauf erfolgt stufenweise und aufbauend. Er beginnt mit dem Unittest. Hierbei werden einzelne Klassen oder Module getestet. Daran schließt sich der Integrationstest an. Der Integrationstest ist für den Test der Kommunikation und Interaktion zwischen den einzelnen Klassen zuständig. Als nächstes erfolgt der Systemtest. Dieser Test erfasst das Verhalten des Gesamtsystems und prüft es gegen die Systemspezifikation. Abschließend erfolgt der Abnahme- oder Akzeptanztest.

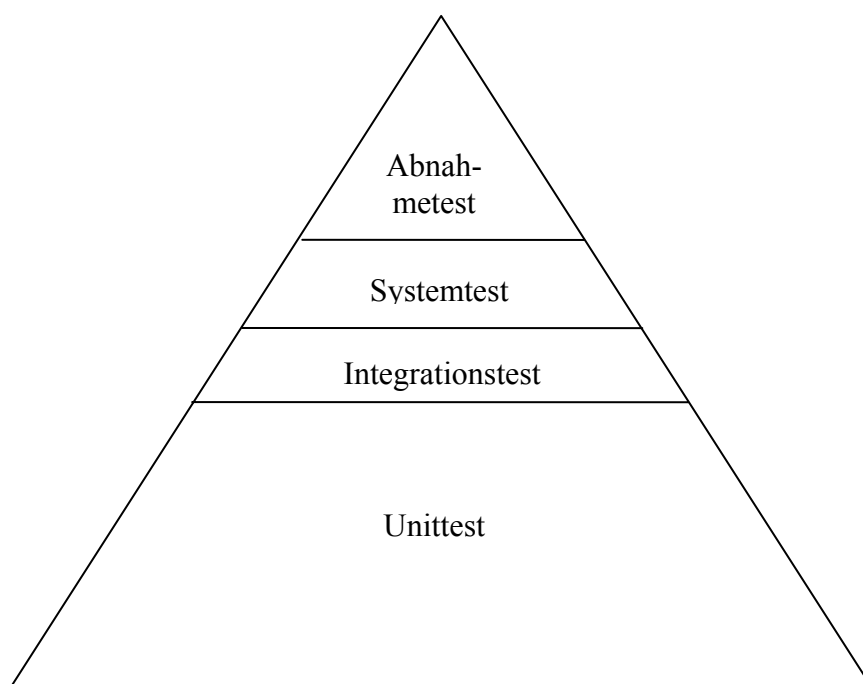


Abbildung 4.1: Stufenweiser Aufbau des Softwaretests

Die Zeitaufwände für die Tests sind der in Abbildung 4.1 gezeigten Pyramidenstruktur ähnlich. Für den Unittest ist ca. 70% der gesamten Testzeit zu veranschlagen. Der Integrationstest, Systemtest und Abnahmetest können in jeweils ca. 10% der Zeit abgearbeitet werden.

4.1 Unittest

Der Unittest wird auch als Klassen- oder Modultest bezeichnet. Dieser Test ermöglicht es dem Entwickler eigene Fehler zu finden. Das Ziel dieses Tests ist es, eine Rückkopplung über die Lauffähigkeit und Korrektheit des Codes zu bekommen, das heißt, ob das zu entwickelnde System lauffähig ist und die Anforderungen hinsichtlich der Funktionalitäten korrekt erfüllt werden. In den meisten Fällen werden dabei nur ein Drittel aller Programmierfehler gefunden. Hierbei handelt es sich meist um Codier- und Logikfehler.

Es gibt vier Arten von Klassen, die für das Testen unterschieden werden:

1. *Nonmodale Klassen*: „Nonmodale Klassen unterliegen keinen Einschränkungen bezüglich des Objektzustandes oder der Aufruffreihenfolge“ (Quelle: [3]). Zu den nonmodalen Klassen gehören Klassen, die nur Werte abspeichern und primitive get- und set-Methoden besitzen. Alle Methoden dieser Klassen können einzeln getestet werden.
2. *Unimodale Klassen*: Bei dieser Art von Klassen werden die Methoden nur in einer festgelegten Reihenfolge aufgerufen. Alle Methoden müssen in jeder Aufrufsequenz getestet werden.
3. *Quasi-modale Klassen*: Bei Quasimodalen Klassen können die Methoden nur bei bestimmten Objektzuständen aufgerufen werden. Quasimodale Klassen können durch einen Zustandsübergangsgraphen beschrieben werden. „Beim Test sollten sämtliche Zustandsübergänge getestet werden.“ (Quelle: [3])
4. *Modale Klassen*: Hier muss sowohl der Zustand der Objekte als auch die Reihenfolge von Methodenaufrufen berücksichtigt werden. „Der Test lässt sich wieder aus der Abdeckung aller Zustandsübergänge im zugehörigen Zustandsübergangsgraphen ableiten.“ (Quelle: [3])

Nach Binder [1] gibt es die unten aufgeführten Klassentests, auf die ab Kapitel 4.1.2. genauer eingegangen wird:

- *Nonmodaler Klassentest*: Entwicklung einer Testfolge für Klassen ohne sequentielle Einschränkungen.
- *Modaler Klassentest*: Entwicklung einer Testfolge für Klassen mit sequentiellen Einschränkungen.
- *Quasi-modaler Klassentest*: Entwicklung einer Testfolge für Klassen mit bestimmten, sequentiellen Einschränkungen.

Für die Klassentests werden so genannte *Invariant Boundaries* erstellt, welche Testfälle für komplexe Bereiche identifizieren. Auf diese wird im nächsten Abschnitt genauer eingegangen.

4.1.1 Invariant Boundaries

Das Ziel der Invariant Boundaries ist es, eine testeffiziente Kombination von Testwerten für Klassen, Interfaces und Komponenten, die auf komplexen und primitiven Datentypen beruhen, auszuwählen.

„Fehler hängen oft mit den Grenzbereichen von Variablenwerten zusammen. Unübliche, aber zulässige Wertekombinationen von Instanzvariablen werden auch oft falsch behandelt“ (Quelle: [14]). Es wird davon ausgegangen, dass unübliche, grenzwertige Kombinationen von Variablenwerten Fehler aufdecken.

4.1.1.1 Strategie

Eine Invariant Boundaries Testfolge wird in 3 Schritten entwickelt:

1. Definieren der Klasseninvariante.
2. Bestimmung der Grenzpunkte (on/off Punkte).
3. Werte für die in der Invariante nicht genannten Variablen bestimmen.

4.1.1.1.1 Definieren der Klasseninvariante

Die Entwicklung der Invariant Boundaries wird an der Java-Klasse `CustomerProfile` illustriert:

```
class CustomerProfile{
    Account account1 = new Account();
    Account account2 = new Account();
    Money creditLimit = new Money();
    short txCounter;
    //...
}
```

Es sei angenommen, dass `account1`, `account2`, `creditLimit` und `txCounter` Instanzen und Interfacevariablen sind. `Account` ist eine modale Klasse mit den abstrakten Zuständen `open`, `overdrawn`, `frozen`, `inactive` und `closed`. `Money` ist ein skalarer Datentyp mit einem Wertebereich von -10^{12} bis $+10^{12}$. Ein skalarer Datentyp ist durch eine geordnete Menge von Werten definiert. Die `txCounter` Instanzvariable ist ein nichtnegativer Integerwert des primitiven Javatyps `short`.

Die Klasseninvariante ist:

```
assert(
    ( txCounter >= 0 && txCounter <= 5000 ) &&
    ( creditLimit > 99.99 && creditLimit <= 1000000.00 ) &&
    (!account1.isClosed() || !account2.isClosed())
);
```

Bei dieser Invariante kann der Kunde maximal 5000 Transaktionen durchführen. Das Kreditlimit beträgt 1.000.000,00 Euro. Es ist erforderlich, dass beide Accounts gültig sind.

4.1.1.1.2 Bestimmung der Grenzpunkte (on/off Punkte)

Im nächsten Schritt werden die Grenzpunkte bestimmt.

Die Punkte im Wertebereich einer Variablen haben eine der folgenden Klassifikationen:

- On-Punkt: liegt auf einer Bereichsgrenze.
- Off-Punkt: liegt auf keiner Bereichsgrenze.
- In-Punkt: liegt innerhalb der Bereichsgrenzen.
- Out-Punkt: liegt außerhalb der Bereichsgrenzen.

Für jede Bereichsgrenze gibt es einen On-Punkt und eine beliebige Anzahl von Off-Punkten.

Nach [14] erfolgt die Auswahl der Punkte wie folgt:

- ein On-Punkt und ein Off-Punkt für jede relationale Bedingung.
- ein On-Punkt und zwei Off-Punkte für jede Gleichheit auf geordneten Typen.
- ein On-Punkt und ein Off-Punkt für jeden ungeordneten Typ.

- ein On-Punkt und mindestens ein Off-Punkt für jede abstrakte Zustandsinvariante. Eine Zustandsinvariante drückt eine Bedingung für einen Zustand aus.
- ein On-Punkt und ein Off-Punkt für jede nicht-lineare Grenze.
- gleiche Tests für aneinandergrenzende Unterbereiche nicht wiederholen.

Die folgende Tabelle zeigt eine Übersicht über die On- und Off-Punkte des `CustomerProfile` Beispiels:

Bedingung	On-Punkt	Off-Punkt
<code>txCounter >= 0</code>	0	-1
<code>txCounter <= 5000</code>	5000	5001
<code>creditLimit > 99.99</code>	99.99	100.00
<code>creditLimit <= 1000000.00</code>	1000000.00	1000000.01
<code>!account1.isClosed()</code>	True	False
<code>!account2.isClosed()</code>	True	False

Tabelle 4.1: On- und Off-Punkte für die `CustomerProfile` Invarianz

4.1.1.1.3 Werte für die in der Invariante nicht genannten Variablen bestimmen

Im letzten Schritt werden die kompletten Invariant Boundaries in einer *Domainmatrix* dargestellt.

Hierzu werden jeweils die On- und Off-Punkte für die einzelnen Variablen angegeben. Für das oben angeführte Beispiel ist in Tabelle 4.2 ein Auszug aus der Domainmatrix angegeben.

Boundary			Testfall				
Variable	Bedingung	Typ	1	2	3	4	5
txcounter	>= 0	On	0				
		Off		11			
	<= 5000	On			5000		
		Off				5001	
	Typisch	In					3500

Tabelle 4.2: Auszug aus einer Domainmatrix für Invariant Boundaries

4.1.1.2 Eingangskriterium

Die Eingangskriterien definieren die Kriterien, die erfüllt sein müssen, damit der Test durchgeführt werden kann.

Um für den nonmodalen, den quasimodalen und den modalen Klassentest Testfolgen zu erstellen, wird folgendes Pattern verwendet:

- Eine gültige Invariante ist vorhanden oder kann für die zu testende Implementierung entwickelt werden.

4.1.1.3 Ausgangskriterium

Das Ausgangskriterium bestimmt die Kriterien, die für einen erfolgreich beendeten Test erfüllt sein müssen.

In Bezug auf die Invariant Boundaries bedeutet dies, dass eine vollständige Domainmatrix entwickelt wurde. Für jede allgemeine Grenzbedingung gibt es einen On- und Off-Punkt und für genau gleiche Grenzbedingungen einen On-Punkt und zwei Off-Punkte.

4.1.2 Nonmodaler Klassentest

Der nonmodale Klassentest ist der einfachste aller Klassentests, da der Zustand der Objekte keinen Einfluss auf das Verhalten der Methoden hat.

„Nicht-modale Klassen lassen jede Folge von Methodenaufrufen zu, haben aber oft einen komplexen Datenzustand und eine komplexe Schnittstelle“ (Quelle: [14]). Das Ziel ist es, Methodenaufrufsequenzen zu finden, die Fehler aufdecken.

Da es keine Einschränkung der Aufrufsequenzen gibt, ist ein zustandsbasiertes Testen nicht erforderlich. Bei nonmodalen Klassen können unter anderem häufig die folgenden Fehler auftreten:

- zulässige Sequenz wird abgelehnt.
- zulässige Sequenz ergibt falschen Wert.
- Zustandsabfragen sind inkonsistent.
- zulässige Veränderung wird abgelehnt.

4.1.2.1 Strategie

Für nonmodale Klassen wird die folgende Teststrategie angewendet:

1. Entwicklung der Testfälle mit dem Pattern Invariant Boundaries.
2. Auswahl einer Aufrufabfolge-Strategie: *define-use*, *random* oder *suspect*.
3. Das zu testende Objekt wird einem Testfall ausgesetzt.
4. Rufe alle Zugriffsmethoden auf und vergleiche die Ausgaben mit dem Testfall.
5. Wiederhole die Schritte (3) und (4) bis alle Sequenzen nach (2) ausgeführt wurden.

Bei der *define-Use* Strategie wird ein definierender Methodenaufruf von Aufrufen aller Zugriffsmethoden gefolgt. Die *random* Strategie erzeugt zufällige Aufrufsequenzen. Bei der *suspect* Strategie werden verdächtige Sequenzen ausgeführt.

4.1.2.2 Eingangskriterium

Die minimale Ausführbarkeit der zu testenden Klasse wird durch Durchlaufen des *alpha-omega-Zyklus* nachgewiesen. Der *alpha-omega-Zyklus* überführt das zu testende Objekt vom *alpha-Zustand* in den *omega-Zustand*. Der *alpha-Zustand* repräsentiert die Objektdekla-

ration bevor das Objekt erstellt wurde, der omega-Zustand repräsentiert das Skelett nachdem das Objekt gelöscht oder zerstört wurde.

4.1.2.3 Ausgangskriterium

Die Überdeckung einer nicht-modalen Klasse ist gezeigt, wenn alle Define-Use-Paare ausgeführt wurden und alle Testfälle aus Invariant Boundaries mindestens einmal als Zustand angenommen worden sind. Unter der Überdeckung einer Klasse versteht man die Ausführung aller Methoden dieser Klasse.

Mindestens eine Zweigüberdeckung für jede Methode der zu testenden Klasse wurde geprüft. Um eine Zweigüberdeckung zu erreichen, muss jede Methode mindestens einmal durchlaufen worden sein.

4.1.2.4 Anforderungen

Die Größe von einem define-use Test wächst exponentiell mit n , wobei n die Sequenz ist. Die Anzahl von Tests beträgt m^{n+1} , wobei m die Anzahl von Methoden in der zu testenden Klasse ist.

4.1.3 Quasi-modaler Klassentest

Bei quasi-modalen Klassen können die Methoden nur bei bestimmten Objektzuständen aufgerufen werden.

Container und *Collection-Klassen* sind meist quasi-modale Klassen. Eine Containerklasse verwaltet und enthält Objekte aus anderen Klassen. Collections sind eine Sammlung von Objekten. Bei den Klassen interessiert oft nicht der genaue Datenzustand, sondern eine Abstraktion davon.

„Quasi-modale Fehler treten auf, wenn Klasseninvarianten, die sich auf alle Elemente eines Containers beziehen, verletzt werden“ (Quelle: [14]). Als Beispiel dient die folgende C++ Klasse `IntSet`:

```
class IntSet {
    IntSet();
    ~IntSet();

    IntSet&add(int); //einen Wert hinzufügen
    IntSet&removeInt(int); //einen Wert entfernen
    IntSet&Clear(int); //alle Werte entfernen
    int is_member(int); //ist das Argument ein
                        //vorhandener Wert?
    int extent(); //Anzahl der Elemente
    int is_empty(); //leer oder nicht?
}
```

4.1.3.1 Strategie

Der quasimodale Test besteht aus 2 Teilen:

1. Für die Klasse wird ein generisches Zustandsmodell erstellt. Auf das Modell wird in Kapitel 4.1.3.1.1 genauer eingegangen.
2. Invariant Boundaries werden für Parameter, die das Verhalten bestimmen, erzeugt und interessante Operationssequenzen werden getestet.

4.1.3.1.1 Generisches Zustandsmodell

Abbildung 4.2 zeigt ein generisches Zustandsmodell für Collections. Die zugehörigen Zustände sind in Tabelle 4.3 definiert. Die Zustände α und ω sind Platzhalter. Die Platzhalter werden genutzt, um den kompletten Zyklus von der Erstellung bis zur Löschung explizit darzustellen. Die generischen Events sind in Tabelle 4.4 dargestellt.

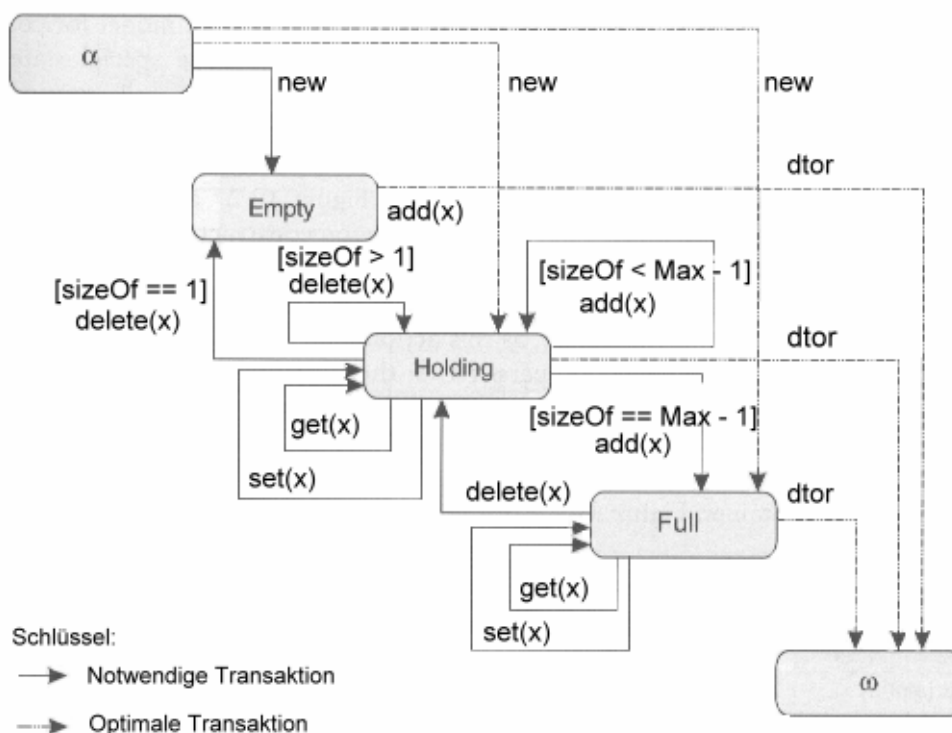


Abbildung 4.2: Generisches Zustandsmodell für Collections

Status	Definition
α (alpha)	Das Objekt ist definiert aber nicht deklariert.
Empty	Die Collection hat null Werte, ist somit leer.
Holding	Die Collection hat mindestens einen Wert, aber nicht mehr als die maximale Anzahl an Werten.
Full	Die Collection hat die Maximale Anzahl an Werten.
ω (omega)	Der Garbagestatus.

Tabelle 4.3: Generische quasi-modale Zustände

Event	Definition
New	Erstellt und initialisiert eine Instanz.
add(x)	Fügt Element x zur Collection hinzu.
set(x)	Ändert den Wert des existierenden Elements zu x, ohne den Wert aus der Collection zu entfernen.
get(x)	Gibt eine Kopie der Referenz zum Element x zurück, ohne die Collection zu verändern.
delete(x)	Entfernt Element x aus der Collection.
Dtor	Zerstört die Collection.

Tabelle 4.4: Generische quasi-modale Events

Die Abbildung 4.3 zeigt einen Transitionsbaum für das generische Zustandsmodell, welches in Abbildung 4.2 dargestellt ist.

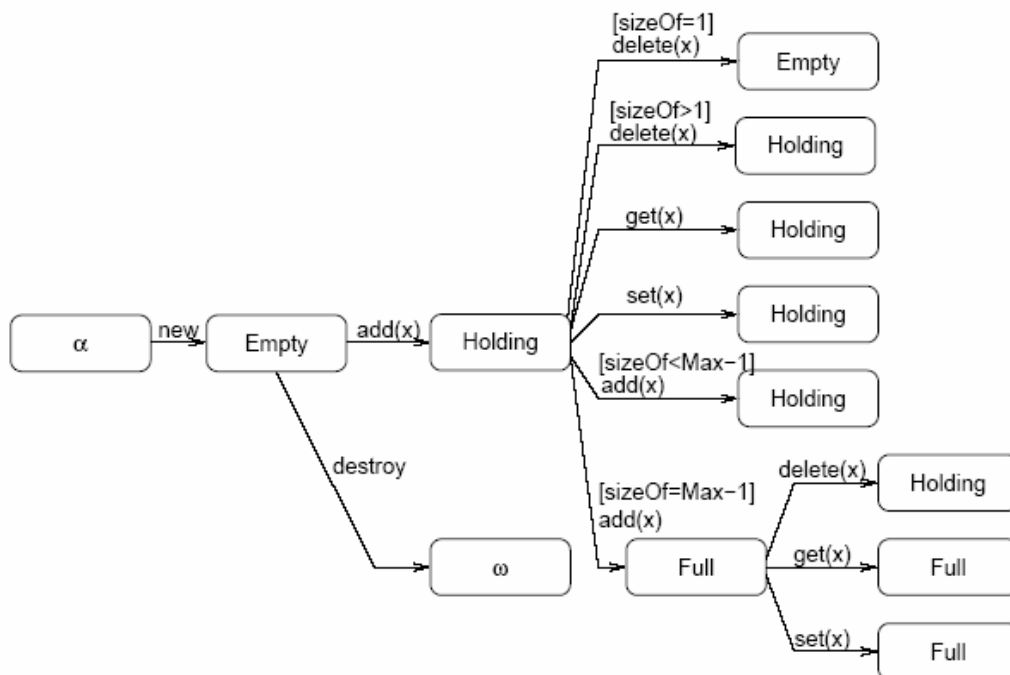


Abbildung 4.3: Transitionsbaum für das quasi-modale Zustandsmodell

Der Aufbau des Transitionsbaumes erfolgt nach den folgenden Regeln:

1. Der Ausgangszustand ist die Wurzel des Baumes.
2. Für jeden nicht terminierenden Blattknoten im Baum wird die Anzahl von auslaufenden Transitionen für den repräsentierten Zustand gezählt. Eine Kante und ein neuer Knoten werden für die jede auslaufende Transition gezeichnet. Jede neue Kante und jeder neue Knoten repräsentieren ein Event und den resultierenden Status, der durch die auslaufende Transition erreicht wird.
3. Für jede Kante und jeden Knoten aus Schritt 2 wird folgendes gezeichnet:
 - a. Kopiere das entsprechende Transitionsevent zu der neuen Kante.

- b. Ist der Status dieses Knotens bereits durch einen anderen Knoten repräsentiert oder ist es der Endzustand, dann ist dieser Knoten abgeschlossen. Es werden keine weiteren Transitionen von diesem Knoten ausgehen.
4. Wiederhole die Schritte 2 und 3 bis alle Blattknoten abgeschlossen sind.

4.1.3.1.2 Interessante Operationssequenzen

Im nächsten Schritt wird das collectionspezifische Verhalten getestet. Für den Entwurf der Aufrufsequenzen existieren die folgenden Collections:

- *Sequentielle Collections*: Die sequentiellen Collections enthalten eine variable Anzahl an Elementen. Diese Elemente können nur in einer festen Reihenfolge angesprochen werden, wie zum Beispiel bei Listen oder Puffern.
- *Geordnete Collections*: Bei den geordneten Collections werden die Elemente nach einer bestimmten Ordnung (zum Beispiel Stack, Queue und Bäume) hinzugefügt oder gelöscht.
- *Schlüsselcollections*: Bei den Schlüsselcollections wird der Zugriff über einen Primärschlüssel realisiert. Schlüsselcollections treten meist in einem Suchbaum oder bei einer Datenbankschnittstelle auf.

4.1.3.2 Eingangskriterium

Die minimale Ausführbarkeit der zu testenden Klasse wird durch Durchlaufen des alpha-omega-Zyklus nachgewiesen.

4.1.3.3 Ausgangskriterium

Mindestens eine Zweigüberdeckung für jede Methode der zu testenden Klasse wurde geprüft.

Die erzeugten „Testfälle überdecken alle Äste im Transitionsbaum und produzieren vollständige Mengen von Operationspaaren für illegale Übergänge“ (Quelle: [14]). Bei illegalen Übergängen wird der Aufruf akzeptiert, der eigentlich abgelehnt werden sollte.

Es kann ein Klassen-Flussgraph erstellt werden, um eine vollständige Überdeckung der alpha-omega Pfade zu gewährleisten. Der Klassen-Flussgraph ergibt sich aus der Reihenfolge der Ausführung von Klassen eines Programms, das heißt die Klassen die früh ausgeführt werden, erscheinen auch früh im Flussgraphen.

4.1.3.4 Anforderungen

Das quasi-modale Klassentest-Pattern hat die folgenden Anforderungen:

- Ein testbares Verhaltensmodell ist verfügbar oder kann entwickelt werden.
- Der Zustand der zu testenden Klasse ist beobachtbar.
- Ein geeigneter Testtreiber ist verfügbar. Auf den Begriff Testtreiber wird in Kapitel 4.2 genauer eingegangen.

4.1.4 Modaler Klassentest

Der modale Klassentest ist der schwierigste Klassentest, da die Klasse sowohl von der Reihenfolge der Methodenaufrufe als auch vom Zustand der Objekte abhängig ist.

Nach [14] existieren fünf Arten, um das Zustandsmodell einer Klasse fehlerhaft zu implementieren:

- fehlende Transition – ein Aufruf wird in einem zulässigen Zustand abgelehnt.
- falsche Aktion – inkorrekte Änderung der Instanzvariablen.
- falscher Folgezustand – Transition in einem falschen Zustand.
- fehlerhafter Zustand – es wird kein zulässiger Zustand erreicht.
- illegaler Übergang – Aufruf wird akzeptiert, der eigentlich abgelehnt werden sollte.

Nach [2] gibt es die folgenden Zustandsfehler und mögliche Gründe dafür:

Fehler		Mögliche Gründe		
		Lexikalisch	Dynamisch	Allgemein
Transformation	fehlt	X	X	...
	falsch	X	X	...
	extra	X	X	Kreatives Programmieren
Zustandstransformation	fehlt	X	X	Das zu testende System kann eine Sequenz nicht erzeugen.
	falsch	X	X	Das zu testende System kann eine Sequenz nicht erzeugen oder ein illegaler Übergang erlaubt eine unspezifische Sequenz.
	extra	X	X	Ein illegaler Übergang erlaubt eine unspezifische Sequenz.
Ausgabeaktion	fehlt	X	X	...
	falsch	X	X	...
	extra	X		Kreatives Programmieren.
Zustand	fehlt	X	X	...
	falsch	X	X	Fehlerhafte Methodenweisungen, falsche Nutzung des Clients.
	extra	X		Kreatives Programmieren.

Tabelle 4.5: Zusammenfassung der Zustandsfehler und mögliche Gründe

4.1.4.1 Strategie

Die Strategie des modalen Klassentests ist ähnlich wie die des quasi-modalen Klassentests. Zusätzlich erfordert der modale Klassentest noch die Beachtung der Transitionsbedingungen.

4.1.4.2 Eingangskriterium

Der modale Klassentest hat dasselbe Eingangskriterium wie der quasi-modale Klassentest.

4.1.4.3 Ausgangskriterium

Der modale Klassentest hat dieselben Ausgangskriterien wie der quasi-modale Klassentest.

4.1.4.4 Anforderungen

Das modale Klassentest-Pattern hat dieselben Anforderungen wie das quasi-modale Klassentest-Pattern.

4.2 Integrationstest

„Der Integrationstest testet die Kommunikation und Interaktion zwischen einzelnen Klassen, Modulen und Komponenten. Das Ziel des objektorientierten Integrationstests ist die Überprüfung des korrekten Zusammenwirkens von dienst anbietenden (Server) und dienstnutzenden (Client) Objekten unterschiedlicher Klassen, die nicht in einer Vererbungsbeziehung stehen. Die Integration von abgeleiteten Klassen und ihren Basisklassen ist bereits Aufgabe des Klassentests.“(Quelle: [3])

Für die Durchführung des Integrationstests bei objektorientierten Systemen gibt es mehrere Ansätze. Nach Binder werden folgende neun Patterns unterschieden [1]:

- *Big Bang Integration*: Alle Komponenten zusammen werden auf einmal getestet.
- *Bottom-up Integration*: Integration der Komponenten von den Blättern des Abhängigkeitsbaums zur Wurzel.
- *Top-down Integration*: Integration der Komponenten von der Wurzel des Abhängigkeitsbaums zu den Blättern.
- *Collaboration Integration*: Integrationsreihenfolge entsprechend des Zusammenwirkens und der Abhängigkeiten.
- *Backbone Integration*: Kombination von Top-down Integration, Bottom-up Integration und Big Bang Integration.
- *Layer Integration*: Einteilung des Systems in verschiedene Schichten, die entweder Top-down oder Bottom-up integriert werden.
- *Client/Server Integration*: Einteilung des Systems in Client- und Serverkomponenten, die schrittweise integriert werden.
- *Distributed Integration*: Integration verteilter Komponenten (Netzwerkanwendungen).
- *High-frequency Integration*: Häufige Wiederholung der Integration.

Die Wahl der Integrationsstrategie hat einen großen Einfluss auf die Anzahl an erforderlichen Testtreibern und Platzhaltern.

Ein Testtreiber, auch als Driver bezeichnet, „wird benötigt, um eine Systemkomponente zu testen, deren Dienst nicht direkt von der Benutzungsoberfläche aufgerufen und mit Parametern versorgt werden kann“ (Quelle: [5]).

Platzhalter werden auch als Dummies oder Stubs bezeichnet. Sie werden benötigt, „wenn eine zu testende Systemkomponente andere Systemkomponenten aufruft, die zwar spezifiziert,

aber noch nicht implementiert sind oder aus anderen Gründen nicht für den Test benutzt werden sollen“ (Quelle: [5]). Auch werden Platzhalter zur Simulation von nicht verwendeten Komponenten eingesetzt.

In diesem Kapitel wird auf die neun Pattern des Integrationstests genauer eingegangen. Für die folgenden Grafiken, wird die unten abgebildete Legende genutzt:

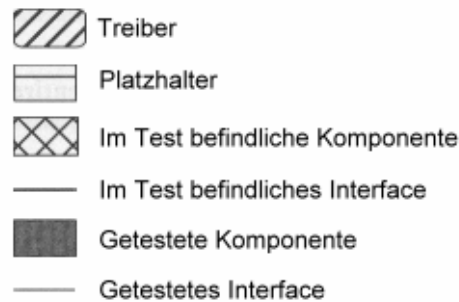


Abbildung 4.4: Legende Grafiken Integrationstest

4.2.1 Big Bang Integration

Die Big Bang Integration bringt alle Komponenten des zu testenden Systems, ohne auf Abhängigkeiten und Risiken zu achten, zusammen. Ein Systemanwendungstest wird angewandt, um die minimale Durchführbarkeit zu demonstrieren. Die Big Bang Integration findet in folgenden Situationen Anwendung:

- Das zu testende System läuft stabil und nur ein paar Komponenten wurden seit dem zuletzt durchgeführten Anwendungstest hinzugefügt oder geändert.
- Das zu testende System ist klein und prüfbar und jede einzelne Komponente hat adäquate Komponentenanwendungstests durchlaufen.

Die Big Bang Integration ist vielleicht der einzige durchführbare Ansatz für ein monolithisches System. Sind Komponenten so straff gekoppelt, können sie nicht separat getestet werden; es ist somit praktisch unmöglich, für Teilmengen der Komponenten Tests zu entwickeln und diese auch durchzuführen. Dies ist ein häufiges Ergebnis in unstrukturierten Systemen, die mit konventionellen Programmiersprachen entwickelt wurden, objektorientierte Implementierungen mit schlechtem oder keinem Design, oder der Ansatz einer ad hoc Wartung in beiden Fällen.

Sind diese Gegebenheiten nicht vorhanden, so schafft die Big Bang Integration mehr Probleme als sie löst. Wird ein ungetestetes, komplexes System mittels Big Bang integriert, so kann beim Auftreten eines Fehlers dieser aufgrund des umfangreichen Codes nur schwer lokalisiert werden.

4.2.1.1 Strategie

Die Big Bang Integration verzichtet auf eine inkrementelle Integrationsstrategie. Stattdessen wird eine Testfolge angewandt, um die minimale Funktionsfähigkeit des bereits kompilierten Gesamtsystems zu demonstrieren. Abbildung 4.5 zeigt diese Konfiguration.

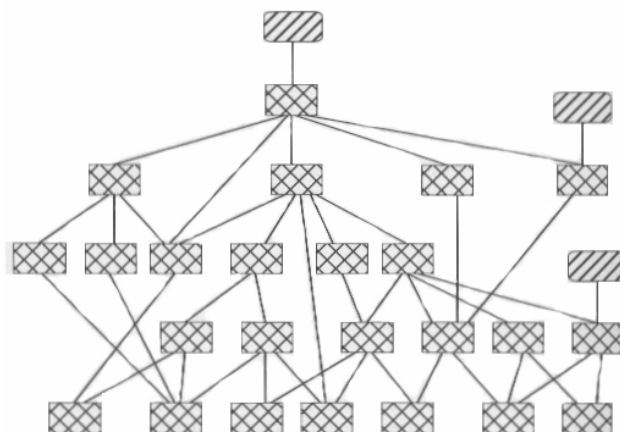


Abbildung 4.5: Beispiel Big Bang Test (Quelle: [1])

4.2.1.2 Eingangskriterium

Alle Komponenten haben den Komponentenanwendungstest durchlaufen.

Die VM, die in der Testumgebung genutzt wird, läuft stabil.

Eine physische und funktionale Prüfung sowie eine Prüfung der Umgebung wurden ausgeführt und es wurden keine Abweichungen gefunden, die den Integrationstest behindern oder beeinflussen.

4.2.1.3 Ausgangskriterium

Der Test ist beendet, wenn die Testfolge erfolgreich durchlaufen wurde.

4.2.1.4 Vorteile

Ein Vorteil der Big Bang Integration sind die vielen möglichen parallelen Testaktivitäten in der Modulphase. Das Testen der einzelnen Klassen erfolgt parallel und unabhängig voneinander.

Der Big Bang Test trägt zu einer schnellen Fertigstellung des Integrationstests bei. So kann der Big Bang Test bei Systemen, bei denen nur wenige Änderungen vorgenommen wurden, angewandt werden. Da davon auszugehen ist, dass die Module und das komplette System vor den Änderungen hinreichend getestet wurden, gestaltet sich die Fehlersuche aufgrund der geringen Änderungen als unproblematisch.

Auch bei kleinen und gut strukturierten Systemen, bei denen die Einzelkomponenten gründlich getestet wurden, ist die Big Bang Integration empfehlenswert.

4.2.1.5 Nachteile

Aufgrund der großen Anzahl von Komponenten kann der Test unter Umständen unstrukturiert und unsystematisch ablaufen. Als schwierig erweist sich die Fehlersuche, da aufgetretene Fehler nur sehr schwer zu lokalisieren sind. Auch die Konstruktion der Testfälle wird erschwert.

Treten bei der Integration keine Fehler auf, so wird für den Testaufwand nur wenig Zeit benötigt. Bei mehr als 3 Fehlern ist die Big Bang Integration laut [1] eine schlechte Wahl und es sollte auf weitere Integrationstestarten zurückgegriffen werden.

4.2.2 Bottom-up Integration

Bei der Bottom-up Integration werden zuerst die Komponenten implementiert und getestet, welche die geringsten Abhängigkeiten zu anderen Komponenten aufweisen. Hierzu kann eine Baumstruktur oder Schichtenstruktur verwendet werden. Bei der Baumstruktur werden zuerst die Blätter des Baumes implementiert, bei der Schichtenstruktur die Komponenten der untersten Schicht. Im Anschluss daran werden die nächst höheren Komponenten implementiert, getestet und integriert. So kommen schrittweise immer mehr Komponenten hinzu, bis zum Schluss das ganze System implementiert ist und getestet ist.

4.2.2.1 Strategie

4.2.2.1.1 Testmodell

Als erstes wird ein Abhängigkeitsbaum entwickelt. Die zugehörigen Testfälle für jede Komponente können mit jedem zugehörigen Testdesignpattern entwickelt werden. Zusätzlich zu der Prüfung der Komponenten müssen die Subkomponenten beobachtet werden. Die Anwendung der Testfälle für jeden Treiber ist auf die zu testenden Komponenten begrenzt, das heißt, der Treiber ist nicht bestrebt die Interfaces von Unterkomponenten zu prüfen. Stattdessen prüft der Treiber die Komponenten, die die Interfaces von Unterkomponenten verwenden. Dadurch erfolgt eine Ausführung und Prüfung dieser Interfaces.

4.2.2.1.2 Testprozedur

Bei einem Baum der Höhe n müssen n Integrationsschritte erfolgen. Im ersten Schritt werden die Blattkomponenten kodiert. Testtreiber werden für die Blattkomponenten implementiert und die Blattkomponenten werden anhand der Treiber getestet (Abbildung 4.6).

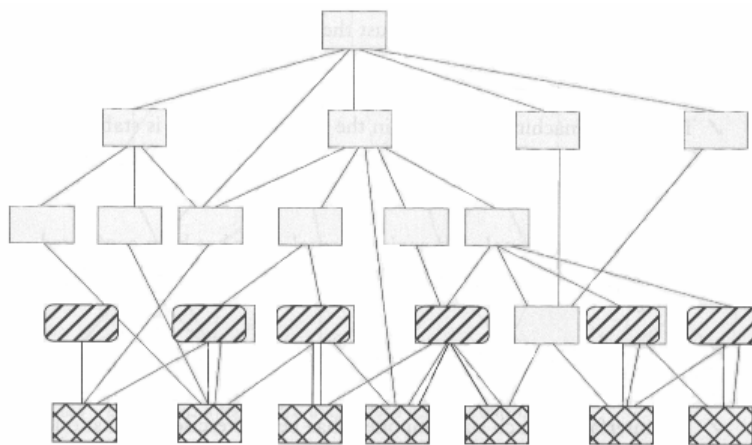


Abbildung 4.6: Bottom-up Integration, 1. Schritt (Quelle: [1])

Anschließend werden Komponenten der nächst höheren Ebene implementiert. Diese verwenden dabei die Komponenten, die bereits im vorherigen Schritt implementiert und getestet wurden. Die Abbildungen 4.7, 4.8 und 4.9 verdeutlichen das Vorgehen.

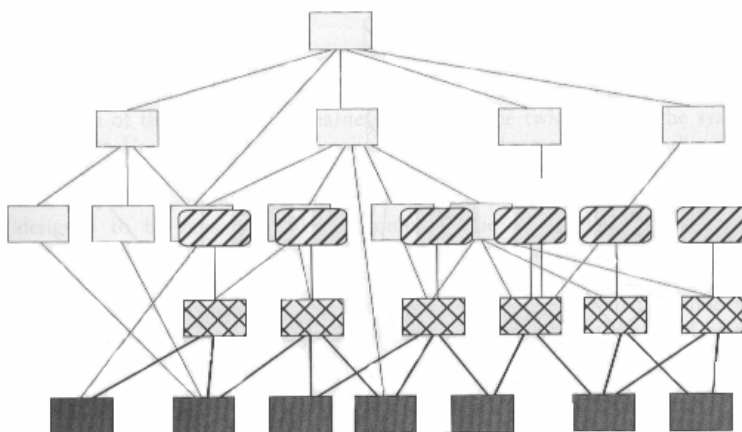


Abbildung 4.7: Bottom-up Integration, 2. Schritt (Quelle: [1])

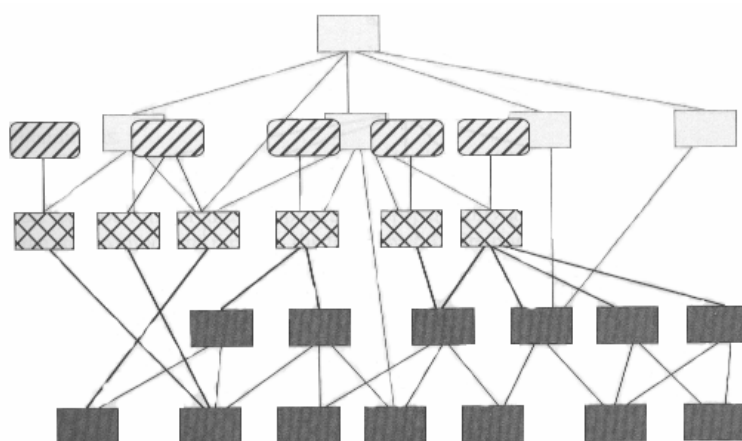


Abbildung 4.8: Bottom-up Integration, 3. Schritt (Quelle: [1])

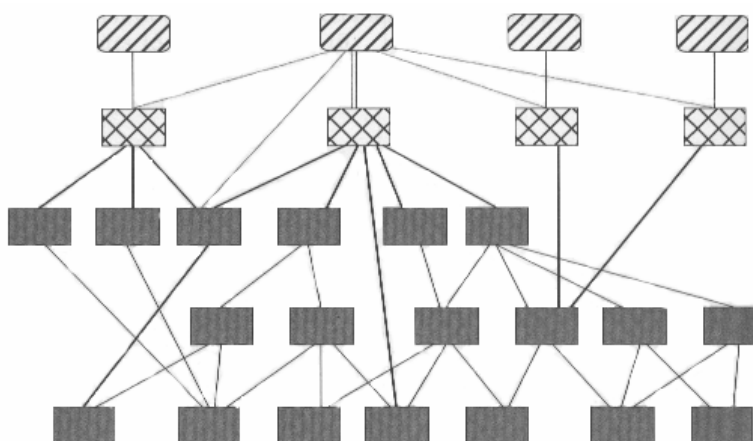


Abbildung 4.9: Bottom-up Integration, 4. Schritt (Quelle: [1])

Im letzten Schritt wird die Wurzelkomponente getestet, wie in Abbildung 4.10 verdeutlicht wird. Mit Prüfung der Wurzelkomponente wird auch die Prüfung des gesamten Systems abgeschlossen.

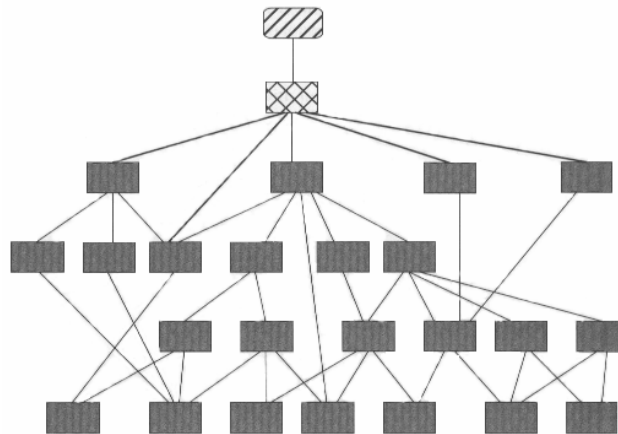


Abbildung 4.10: Bottom-up Integration, Schlusskonfiguration (Quelle: [1])

4.2.2.2 Treiber und Platzhalter

Die Bottom-up Integration benötigt einen Treiber für jede Komponente. Für Komponenten, die eine Wurzel eines Unterbaumes in der Abhängigkeitsbeziehung sind, werden ebenfalls Treiber benötigt. Ein Bottom-up Treiber sollte einer unabhängigen Komponente oder einer Menge an Komponenten entsprechen.

Jede erfolgreiche Testkonfiguration sollte gespeichert und mit einer Versionsnummer versehen werden, bevor neue Testkonfigurationen entwickelt werden. Dies verringert den Aufwand, wenn der Test aufgrund von Designänderungen oder Fehlerkorrekturen erneut durchgeführt werden muss.

4.2.2.3 Eingangskriterium

Die Virtual Machine, die in der Testumgebung genutzt wird, läuft stabil.

Die Komponenten, die in jedem Schritt integriert werden, haben minimale Funktionalitäten.

Eine physische und funktionale Prüfung sowie eine Prüfung der Umgebung wurden ausgeführt und es wurden keine Abweichungen gefunden, die den Integrationstest behindern oder beeinflussen. Die physische Prüfung, dient der Überprüfung des Quellcodes, die funktionale Prüfung dient der Prüfung der Funktionsfähigkeit des Systems.

4.2.2.4 Ausgangskriterium

Jede Treiberkomponente erfüllt das Austrittskriterium für ihr Testpattern.

Das Interface zu jeder Subkomponente wurde wenigstens einmal geprüft.

Der Integrationstest ist fertig ausgeführt, wenn alle Wurzelkomponenten ihre Testfolgen durchlaufen haben.

4.2.2.5 Vorteile

Aufgrund der Implementierung von „unten nach oben“ sind keine Platzhalter erforderlich.

Das Testen und die Integration beginnen, sobald die erste Blattkomponente fertig implementiert ist. Die Arbeit kann parallel erfolgen. Somit können verschiedene Entwickler unabhängig voneinander an verschiedenen Teilbäumen arbeiten.

Ein weiterer Vorteil ist die einfache Erstellung der Testbedingungen. Auch die einfache Interpretierbarkeit der Testergebnisse wirkt sich vorteilhaft aus. Des Weiteren kann eine bewusste Fehlereingabe zur Überprüfung von Ausnahmebehandlungen erfolgen. Die frühe Prüfung des Zusammenwirkens von Software, Systemsoftware und Hardware wirkt sich weiterhin positiv aus, so dass eventuelle Änderungen an der zu entwickelnden Komponente in einem frühen Stadium erfolgen können.

4.2.2.6 Nachteile

Die Entwicklung der Treiber erfordert die höchsten Kosten der Bottom-up Integration. Die Größe des Codes für die Treiber kann leicht doppelt so groß werden, wie die des zu modellierenden Systems. Werden die Treiber für eine Wiederverwendung vorgesehen und entwickelt, kann im Endeffekt von einem Kostenausgleich ausgegangen werden.

Als großer Nachteil ist anzusehen, dass das Gesamtsystem erst sehr spät vorhanden ist. Dies wiederum kann dazu führen, dass Fehler in der Produktdefinition spät erkannt werden und zu umfangreichen Änderungen führen können.

Wird eine schon getestete Komponente geändert, gefixt oder erweitert, so sollte die Testkonfiguration, in der die Komponente zuerst getestet wurde, dementsprechend überarbeitet werden und wieder anlaufen. Die Änderung muss sich schrittweise in der Testkonfiguration fortpflanzen. Dieser Prozess kann sich als sehr fehleranfällig, teuer und zeitaufwendig erweisen.

4.2.3 Top-down Integration

Die Top-down Integration ist die Umkehrung der Bottom-up Integration. Sie prüft zuerst die in der Baumhierarchie am weitesten oben stehenden Systemkomponenten. Die weitere Integration erfolgt schrittweise.

4.2.3.1 Strategie

4.2.3.1.1 Testmodell

Die Kontrollspitze des zu modellierenden Systems kann in einem Kollaborationsdiagramm oder in einem Sequenzdiagramm dargestellt werden.

4.2.3.1.2 Testprozedur

Die Kontrollhierarchie wird in einem Abhängigkeitsbaum dargestellt. Anschließend wird ein Plan für die Implementierung und das Testen erstellt.

Im ersten Schritt wird die erste Komponente bzw. die Komponenten des höchsten Kontrolllevels entwickelt und getestet. Die Server der Komponente werden als Platzhalter oder Proxies implementiert. Abbildung 4.11. verdeutlicht den ersten Schritt.

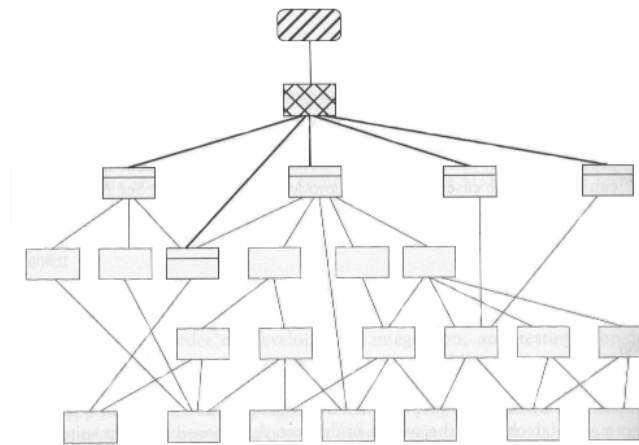


Abbildung 4.11: Top-down Integration, 1. Schritt (Quelle: [1])

Im nächsten Schritt werden die Platzhalter für die Server durch eine Implementierung ersetzt und im nächst kleineren Level werden Platzhalter für Server gesetzt. Dieses Vorgehen verdeutlichen die Abbildungen 4.12, 4.13 und 4.14.

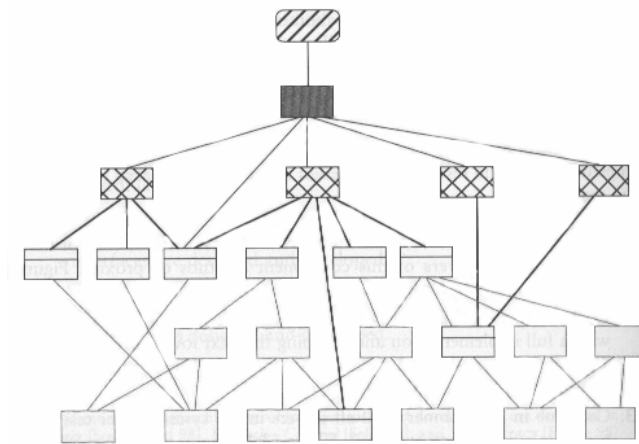


Abbildung 4.12: Top-down Integration, 2. Schritt (Quelle: [1])

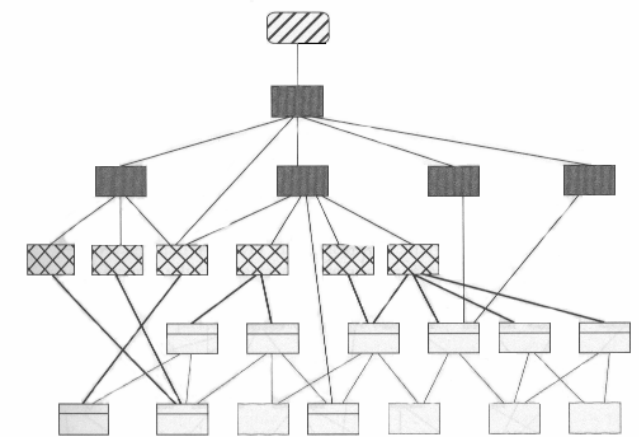


Abbildung 4.13: Top-down Integration, 3. Schritt (Quelle: [1])

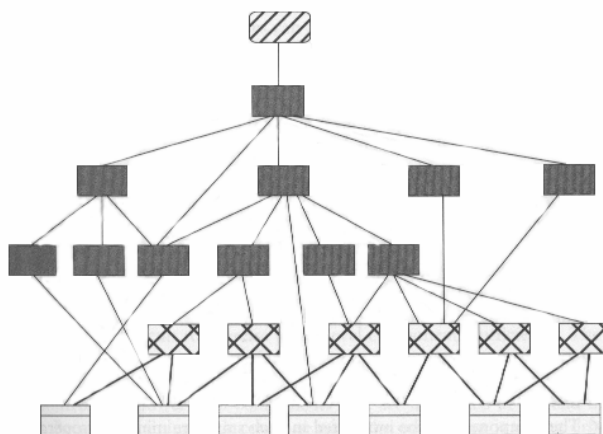


Abbildung 4.14: Top-down Integration, 4. Schritt (Quelle: [1])

Schritt 2 wird so lange ausgeführt, bis alle Server implementiert und überprüft sind.

Abbildung 4.15 zeigt die Schlusskonfiguration.

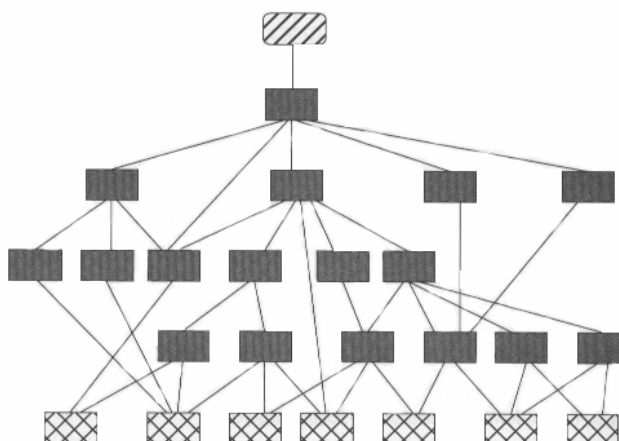


Abbildung 4.15: Top-down Integration, Schlusskonfiguration (Quelle: [1])

4.2.3.2 Treiber und Platzhalter

Die Top-down Integration erfordert einen einzigen Treiber für die Kontrollspitze. Ein Platzhalter wird für jede Komponente im Layer benötigt, die sich unterhalb des aktuellen Integrationsfocuses befindet.

Wie schon bei der Bottom-up Integration sollte auch bei der Top-down Integration eine Versionskontrolle für die Testkonfigurationen erstellt werden, um ein wiederholtes Testen mit verschiedenen Testkonfigurationen zu erleichtern.

4.2.3.3 Eingangskriterium

Die Virtual Machine, die in der Testumgebung genutzt wird, läuft stabil.

Die Komponenten, die in jedem Schritt integriert werden, haben minimale Funktionalitäten, das heißt, sie sollen das Austrittskriterium für den Komponentenanwendungstest erfüllen.

Eine physische und funktionale Prüfung sowie eine Prüfung der Umgebung wurden ausgeführt und es wurden keine Abweichungen gefunden, die den Integrationstest behindern oder beeinflussen.

4.2.3.4 Ausgangskriterium

Jede Treiberkomponente erfüllt das Austrittskriterium für ihr Testpattern.

Das Interface zu jeder Subkomponente wurde wenigstens einmal geprüft.

Der Integrationstest ist fertig ausgeführt, wenn ein übersetzter Quelltext existiert, der alle Blattkomponenten enthält, die die Systemanwendungstestfolgen durchlaufen haben.

4.2.3.5 Vorteile

Das Testen und die Integration beginnen unmittelbar nachdem die top-level Komponenten kodiert sind.

Die Komponenten können parallel entwickelt werden. Mehrere Entwickler können unabhängig an verschiedenen Leveln der Komponenten und Platzhalter arbeiten. Die Implementierung von low-level Komponenten und device-abhängigen Komponenten kann hinausgeschoben werden. Diese Flexibilität ist sinnvoll, wenn die Ziel-Umgebung sich noch in der Entwicklung befindet oder Veränderungen daran vorgenommen werden.

Bei der Top-down Integration entsteht frühzeitig ein Simulationsmodell, das einen Teil der Funktionalitäten des endgültigen Systems enthält. Somit kann das Modell verändert und verbessert werden. Auch kann eine Auswahl an Alternativen frühzeitig erfolgen.

Eine Verzahnung von Entwurf und Implementierung ist ebenfalls möglich. Hierzu müssen zuerst die oberste Schicht entworfen und die Komponenten implementiert werden. Die implementierten Komponenten werden mit Platzhaltern getestet. Im Anschluss daran wird die nächste Schicht entworfen, implementiert etc.

4.2.3.6 Nachteile

Die Entwicklung der Platzhalter und die Wartung verursachen die meisten Kosten bei der Top-down Integration.

Da Platzhalter ein notwendiger Teil von jedem Test sind, erfordert das Erstellen eines Tests eine große Anzahl an Platzhaltern, die kodiert werden müssen, um das gewünschte Ergebnis zu liefern. Komplexe Testfälle können eine Aufnahme von Platzhaltern für jeden Testfall erfordern. Steigt die Anzahl an Platzhaltern für eine Testkonfiguration, wird es schwieriger, die notwendige Kontrolle und Konsistenz zu wahren.

Eine unvorhergesehene Anforderung in einer lower-level Komponente kann Änderungen an einigen top-level Komponenten erforderlich machen. Wird eine schon getestete Komponente geändert, gefixt oder erweitert, so sollte die Testkonfiguration, in der die Komponente zuerst getestet wurde, dementsprechend überarbeitet werden und wieder anlaufen. Diese Änderung muss sich schrittweise in der Testkonfiguration fortpflanzen. Dieser Prozess kann sich als sehr fehleranfällig, teuer und zeitaufwendig erweisen.

Es ist schwierig, lower-level Komponenten hinlänglich zu testen. Die Distanz zwischen dem getesteten Interface und den integrierten Komponenten nimmt mit jedem Level zu. Dies resultiert in einem Verlust der Testkontrolle über lower-level Komponenten, die am Ende des Zyklus integriert werden. Es ist schwierig oder unmöglich, eine hohe Überdeckung zu erlangen, wenn die Komponente eine hohe Distanz zum Treiber hat.

Die Interoperabilität von allen Komponenten des zu modellierenden Systems ist nicht getestet, solange nicht die letzte Komponente ihren Platzhalter ersetzt und die Testreihe durchlaufen wurde.

„Mit zunehmender Integrationstiefe wird die Erzeugung bestimmter Testsituationen in tiefer angeordneten Komponenten schwieriger. Das Zusammenwirken von zu prüfender Software, Systemsoftware und Hardware wird spät geprüft.“ (Quelle: [5])

4.2.4 Collaboration Integration

Bei der Collaboration Integration erfolgt die Reihenfolge der Integration entsprechend dem Zusammenwirken und den Abhängigkeiten der Komponenten. Die Integration ist abgeschlossen, wenn alle Komponenten und Interfaces durchlaufen wurden, jedoch wird nicht der Test aller Zusammenhänge erwartet. Folgende Anzeichen weisen auf die Verwendung einer Collaboration Integration hin:

- Das zu modellierende System hat klar definierte Zusammenhänge, die alle Komponenten und Interfaces im Anwendungsbereich abdecken.
- Aufgrund von technischen Risiken und kritischen Kollaborationen ist es wichtig, so schnell wie möglich zu zeigen, dass die Kollaborationen zusammenarbeiten.

4.2.4.1 Strategie

4.2.4.1.1 Testmodell

Die Komponenten, die zu einer Kollaboration gehören, werden nach der Zugehörigkeit zu einem Entwicklungsthread oder zu kritischen Performanzzielen ausgewählt.

4.2.4.1.2 Testprozedur

Als Testprozedur für die Collaboration Integration gibt Binder [1] folgendes an:

1. Entwicklung eines Abhängigkeitsbaumes für das zu modellierende System. Bilde alle Abhängigkeiten im Abhängigkeitsbaum ab, bis alle Komponenten und Interfaces abgedeckt sind.
2. Wähle eine Reihenfolge, um die Abhängigkeiten darzustellen. Es existieren verschiedene Heuristiken, um eine Reihenfolge festzulegen.
 - a. Beginne mit der Einfachsten und Ende mit der Komplexesten.
 - b. Beginne mit der Abhängigkeit, die die geringste Anzahl an Platzhaltern benötigt oder die die Anzahl an Platzhaltern für andere Abhängigkeiten minimiert.
3. Entwickle eine Testfolge für die erste Abhängigkeit. Wenn möglich, entwickle einen Testtreiber, der hinzugefügte Abhängigkeiten unterstützt. Abbildung 4.16 zeigt die erste Konfiguration.

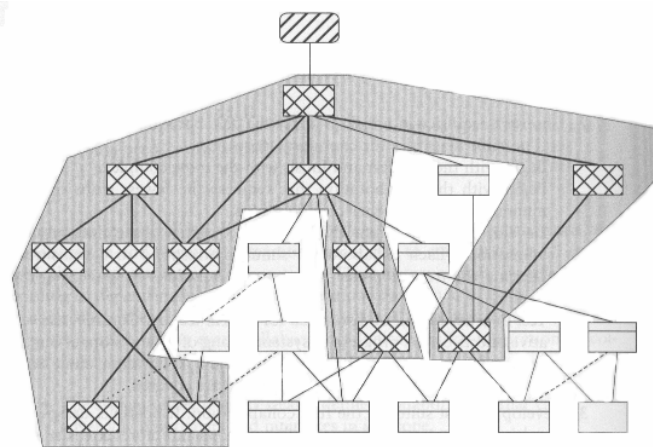


Abbildung 4.16: Collaboration Integration, 1. Konfiguration (Quelle: [1])

4. Lasse die Testfolge laufen und stoppe, wenn die erste Abhängigkeit durchlaufen wurde. Ändere das Testdesign, wenn notwendig. Fahre so lange fort, bis alle Abhängigkeiten durchlaufen wurden. Abbildung 4.17 zeigt die zweite Konfiguration, Abbildung 4.18 verdeutlicht die Schlusskonfiguration.

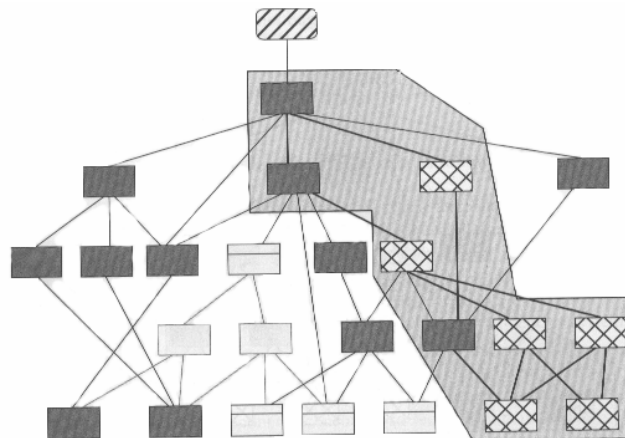


Abbildung 4.17: Collaboration Integration, 2. Konfiguration (Quelle: [1])

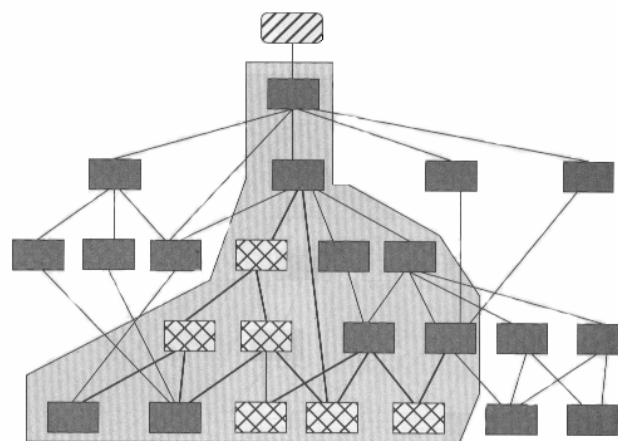


Abbildung 4.18: Collaboration Integration, Schlusskonfiguration

4.2.4.2 Treiber und Platzhalter

Dieses Integrationspattern erfordert einen einzigen Treiber, der die Interfaces für die Kollaboration koppelt. In Applikationen mit einer GUI ist der Treiber typischerweise ein Widget oder eine Sequenz von Widgets. Platzhalter werden für ungetestete Komponenten benötigt, die nicht mit anderen Komponenten zusammenwirken bzw. von ihnen abhängig sind. Ist jede Collaboration getestet, so werden die relevanten Platzhalter ersetzt.

Wie schon bei der Bottom-up und bei der Top-down Integration sollte auch bei der Collaboration Integration eine Versionskontrolle für die Testkonfigurationen erstellt werden, um ein wiederholtes Testen mit verschiedenen Konfigurationen zu erleichtern.

4.2.4.3 Eingangskriterium

Die Virtual Machine, die in der Testumgebung genutzt wird, läuft stabil.

Die Komponenten der Kollaboration haben minimale Funktionalitäten, das heißt, sie sollen das Austrittskriterium für ihr zugehöriges Komponentenanwendungstestpattern erfüllen.

Eine physische und funktionale Prüfung sowie eine Prüfung der Umgebung wurden ausgeführt und es wurden keine Abweichungen gefunden, die den Integrationstest behindern oder beeinflussen.

4.2.4.4 Ausgangskriterium

Jede getestete Kollaboration erfüllt das Austrittskriterium für ihr Testpattern.

Alle Komponenten- und Unterkomponentennachrichten im System wurden wenigstens einmal geprüft. Dieser Schritt erfordert nicht notwendigerweise, dass alle Kollaborationen getestet werden.

4.2.4.5 Vorteile

Die Abdeckung der Interfaces kann mit ein paar Schritten erreicht werden. Es ist nicht notwendig, alle Zusammenhänge zu überprüfen, um eine Integrationsabdeckung zu erlangen, wenn ein Teil der Zusammenhänge alle Interfaces abdeckt.

Testfolgen der Collaboration Integration sind minimal an die zusammenhängenden Komponenten gekoppelt, da nur ein einziges Testinterface genutzt wird. Wenn die zusammenhängenden Komponenten aufgrund von unbeständigen Anforderungen und Bug fixes Probleme bereiten, so kann diese geringe Kopplung die Testfolge reduzieren.

Als vorteilhaft bei der Collaboration Integration erweist sich, dass das Testen und die Integration bereits beginnen, wenn die ersten zusammenhängenden Komponenten implementiert sind. Als Ergebnis hieraus kann die Leistungsfähigkeit später als in der Top-down Integration aber früher als in der Bottom-up Integration demonstriert werden.

Dieses Pattern minimiert die Treiberentwicklungskosten aus denselben Gründen wie die Top-down Integration. Werden lower-level Komponenten hinzugefügt und getestet, so werden upper-level Komponenten erneut ausgeführt und in den Test einbezogen.

4.2.4.6 Nachteile

Auch die Collaboration Integration hat Nachteile.

Spezifizierte Kollaborationen können nicht komplett sein. Das Kollaborationsdiagramm beschreibt einen aufgerufenen Pfad oder eine Familie von aufgerufenen Pfaden, die brauchbar

für die zu testende Implementierung und die Applikation sind. Jedoch müssen nicht alle brauchbaren Pfade modelliert werden.

Da die Zusammenhänge nicht separat getestet werden, ist dieses Pattern auch eine Art Big Bang Integration. Auf die Big Bang Integration wurde im Kapitel 4.2.1. genauer eingegangen.

Auch wird zunächst eine große Anzahl an Platzhaltern bei initialen Zusammenhängen benötigt.

Das Testen von lower-level Komponenten erweist sich als schwierig, da sich die Distanz zwischen dem Testinterface und den zu integrierenden Komponenten als so lang erweisen kann, dass es schwierig ist, Testfälle zu gestalten.

4.2.5 Layer Integration

Bei der Layer Integration erfolgt eine Einteilung des Systems in verschiedene Schichten (Layer). Die unterschiedlichen Schichten werden entweder Top-down oder Bottom-up integriert.

4.2.5.1 Strategie

4.2.5.1.1 Testmodell

Die Layer Integration kombiniert Elemente der Top-down Integration und der Bottom-up Integration. Das Testdesign sollte die Layer identifizieren und ermitteln, welches Integrationspattern für jeden Layer angewandt wird. Unter einem Layer ist hierbei eine Ebene oder Schicht des Systems zu verstehen. Das Testdesign und den Ablauf des Tests innerhalb eines Layers bestimmen die folgenden Pattern:

- Der erste und möglicherweise zweite Layer werden Top-down getestet.
- Der mittlere Layer ist Bottom-up entwickelt.
- Die Primärkomponenten des untersten Layers werden isoliert getestet und das Testdesign entspricht ihrer spezifischen Zuständigkeit.

Alle Kontrollthreads wurden für den obersten Layer getestet und sollten nochmals getestet werden, wenn eine lower-level Testfolge hinzugefügt wurde.

4.2.5.1.2 Testprozedur

Die Layer Integration kann Top-down oder Bottom-up erfolgen. Ein Top-down Ansatz benötigt die folgenden Schritte:

1. Jeder Layer wird isoliert getestet. Treiber können verwendet werden, wenn sie benötigt werden. Das Austrittskriterium für das verwendete Pattern sollte erreicht werden. Abbildung 4.19 stellt den Layer Isolationstest dar.

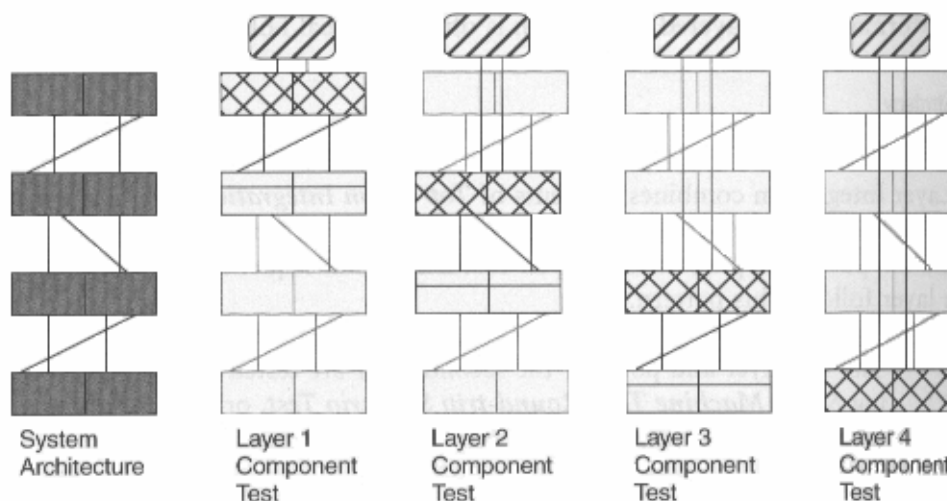


Abbildung 4.19: Layer Integration, Architektur und Komponententestkonfiguration (Quelle: [1])

2. Im 2.Schritt wird die Top-down Integration der Layer vollzogen. Die resultierenden Testfolgen und Testtreiber sollten so gewählt werden, dass sie für spätere Integrationen wieder verwendbar sind.
3. Der Top-down Ansatz wird beim nächsten Layer angewandt. Platzhalter werden bei secondlevel Teilsystemen entfernt und die Interfaces implementiert. Die Kontrolltestfolgen werden erweitert, um die neu implementierten Komponenten zu erreichen und der Test wird durchgeführt.
4. Nachdem die Interfaces in jedem Layer durchgegangen worden sind, werden alle Platzhalter entfernt und die Interfaces des nächsten Layers implementiert.

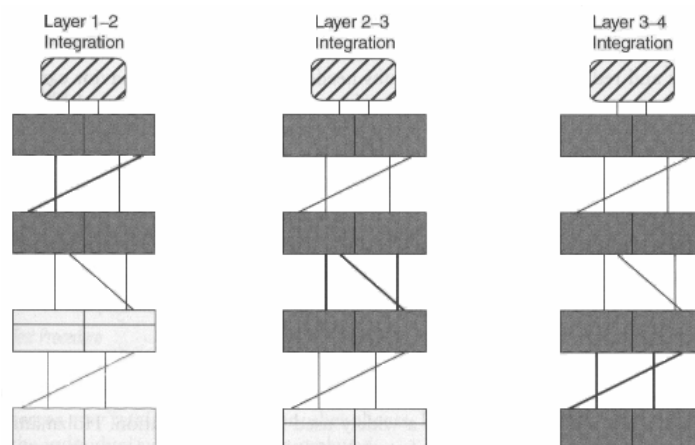


Abbildung 4.20: Layer Integration, Top-down Konfiguration (Quelle: [1])

Abbildung 4.20 zeigt die Testkonfigurationen für einen Top-down Integrationsansatz. Bei dem Bottom-up Integrationsansatz wäre die Reihenfolge umgekehrt.

4.2.5.2 Treiber und Platzhalter

Dieses Pattern erfordert Treiber und Platzhalter für jeden Layer. Die Treiber für den obersten Layer sollten so gestaltet sein, dass sie das ganze System prüfen können.

Auch bei der Layer Integration sollte jede erfolgreiche Testkonfiguration gespeichert und mit einer Versionskontrolle versehen werden, bevor neue Testkonfigurationen entwickelt werden. Dies verringert den Aufwand, wenn der Test aufgrund von Designänderungen oder Fehlerkorrekturen erneut durchgeführt werden muss.

4.2.5.3 Eingangskriterium

Die Virtual Machine, die in der Testumgebung genutzt wird, läuft stabil. Ist die Virtual Machine noch in der Entwicklung, so sollte die Backbone Integration in Betracht gezogen werden. Auf die Backbone Integration wird in Kapitel 4.2.8 genauer eingegangen.

Die Komponenten der Kollaboration haben minimale Funktionalitäten, das heißt, sie sollen das Austrittskriterium für ihr zugehöriges Komponentenanwendungstestpattern erfüllen.

Eine physische und funktionale Prüfung sowie eine Prüfung der Umgebung wurden ausgeführt und es wurden keine Abweichungen gefunden, die den Integrationstest behindern oder beeinflussen.

4.2.5.4 Ausgangskriterium

Jede Treiberkomponente erfüllt das Austrittskriterium für ihr Testpattern.

Jede Kollaboration, die zwei oder mehr Layer durchläuft, wurde wenigstens einmal geprüft.

4.2.5.5 Vorteile und Nachteile

Die Vorteile und Nachteile der Layer Integration sind dieselben wie bei der Top-down Integration, wenn die Top-down Variante verwendet wird. Wird die Bottom-up Methode angewandt, so sind die Vor- und Nachteile gleich der Bottom-up Integration.

Die Top-down Variante erfordert Platzhalter für lower Layer, aber nur einen einzigen Treiber. Wird ein Stack von Layern als Teilsystem verwendet, so kann dieser Ansatz ein lauffähiges Interface für die Integration der Clients des top-Layers zum frühestmöglichen Zeitpunkt bereitstellen. Die Realisierbarkeit des Stacks kann nicht demonstriert werden, solange der unterste Layer nicht integriert ist. Da geschichtete Applikationen oft in zeitkritischen Applikationen verwendet werden, kann die Performance des Stacks solange nicht gezeigt werden, bis der Integrationszyklus beendet wurde.

Die Bottom-up Variante benötigt Treiber für jeden Layer und reduziert die Anzahl an Platzhaltern. Das gesamte Verhalten des Systems kann so lange nicht demonstriert werden, bis der oberste Layer integriert ist.

4.2.6 Client/Server Integration

Bei der Client/Server Integration erfolgt eine Einteilung des Systems in Client- und Serverkomponenten, die schrittweise integriert werden. Gleichzeitiges Entwickeln und Testen ist möglich.

Das zu testende System ist eine Variante der Client/Server Architektur. Im Gegensatz zu einer Top-down Integration existiert kein einziger Kontrollpunkt. Der Kontrollpunkt entscheidet über den Ablauf der Aktionen (zum Beispiel Input, Output und Resourceallokation). Die

Server reagieren auf die Nachrichten des Clients und die Clients reagieren auf Nachrichten der Serverumgebung. Jede Komponente des Systems hat Ihre eigene Kontrollstrategie. Komponenten, die einen einzigen Kontrollpunkt besitzen, können durch Bottom-up, Top-down oder Collaboration Integration integriert werden.

4.2.6.1 Strategie

4.2.6.1.1 Testmodell

Der Testplan muss sowohl für den Client als auch für den Server festgelegt werden. Die Client/Server Interaktionen sollten mit einem geeigneten Testpattern modelliert werden. In einem 2-Schichten Stern besteht die Basisintegrationsstrategie darin, die Client/Server-Paare zu überprüfen. Ein 2-Schichten Stern besteht aus einem zentralen Punkt, dem Server. Der Server ist mit jedem Client verbunden. Die Clients untereinander sind nicht vernetzt. In einem 3-Schichten Stern hingegen werden die Client-Server und Server-Server Konfigurationen, gefolgt durch die Client-Server-Server Konfiguration geprüft. Ein 3-Schichten Stern besteht aus 2-Schichten Sternen, dessen Server mit einem zentralen Server verbunden sind. Aufgrund dieser Architektur erfolgt eine Prüfung der Client-Server-Server Konfiguration.

4.2.6.1.2 Testprozedur

Jeder Client wird mit einem Platzhalter für den Server getestet. Der Server wird mit Platzhaltern für jeden Clienttyp getestet. Anschließend werden Paare von Clients mit dem aktuellen Server getestet. Der Server kann einige Platzhalter für andere Clients weiter verwenden. Zum Schluss werden alle Platzhalter entfernt und die individuellen Anwendungsfälle werden getestet.

Abbildung 4.21 zeigt den generischen 2-Schichten-Stern. Bei einem großen System ist es nicht praktikabel, jeden einzelnen Client individuell zu integrieren. Stattdessen werden Clientgruppen gebildet. Die Mitglieder einer Clientgruppe sollten alle dasselbe Server-Interface besitzen. Bei der 2-Schichten-Stern Architektur wird die folgende 3-Schritt Prozedur für jede Clientgruppe angewandt:

1. Repräsentativer Client + Server Platzhalter
2. Server + Client Platzhalter
3. Repräsentativer Client + Server

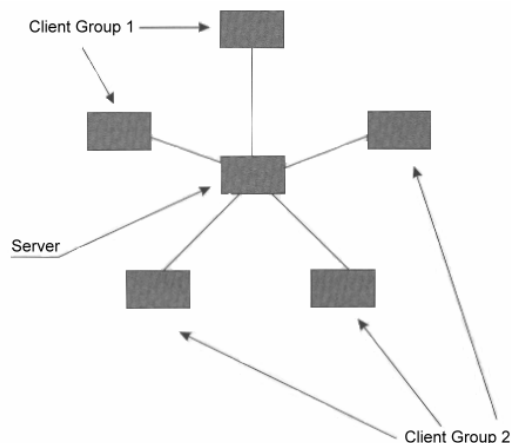


Abbildung 4.21: Generische Stern Client/Server Architektur (Quelle: [1])

Abbildung 4.22 zeigt die Konfigurationen, die den oben angegebenen Stufen entsprechen.

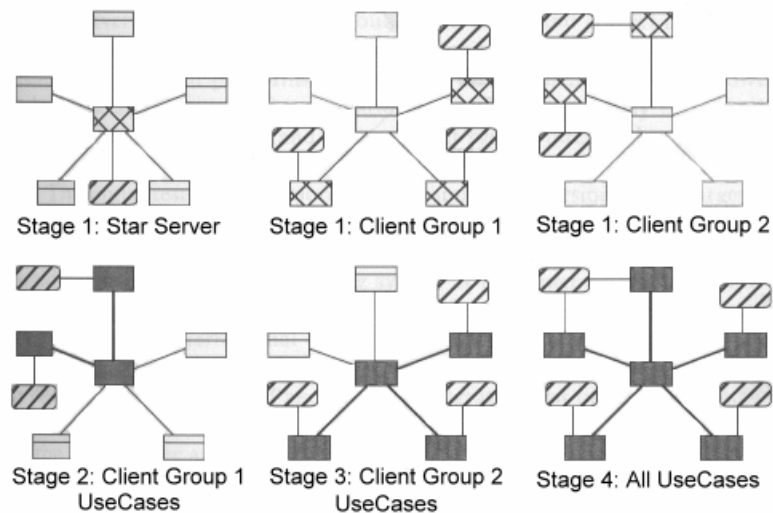


Abbildung 4.22: Client/Server Integration, Stufen und Konfigurationen (Quelle: [1])

Die Client/Server Integration kann auch bei einer 3-Schichten oder n-Schichten Client/Server Architektur angewandt werden. Abbildung 4.23 illustriert eine 3-Schichten-Stern Architektur.

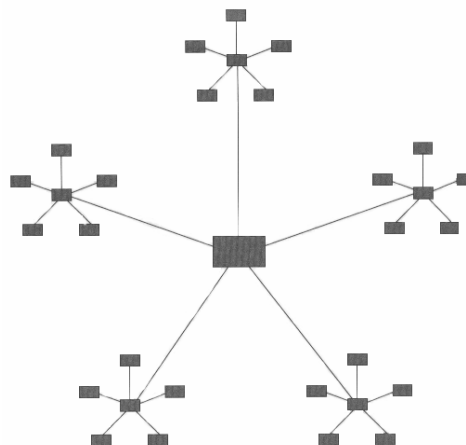


Abbildung 4.23: 3-Schichten Client/Server Architektur (Quelle: [1])

Bei einer 3-Schichten Client/Server Stern Integration, bei der ein Object Request Broker (ORB) genutzt wird, ist der Ablauf ungefähr der folgende:

1. Clientgruppe + ORB
2. Server + ORB
3. Clientgruppe + ORB + Server Proxy
4. Server + ORB + Client Proxy
5. Client + ORB + Server

Der Object Request Broker wird bei CORBA-Anwendungen (Common Object Request Broker Architecture) benötigt. Er sorgt dafür, dass die lokale Anwendung und die Anwendung auf dem Server problemlos miteinander kommunizieren können.

4.2.6.2 Treiber und Platzhalter

Das Interface der zu testenden Komponente entscheidet über die Art der Treiber, die genutzt werden. Es sollten Testskripte entwickelt werden, die 2 Arten der Wiederverwendung unterstützen. Erstens sollten sie wiederholbar den Regressionstest bei der Integration mehrerer Komponenten unterstützen. Regressionstests werden eingesetzt, um nach einer Veränderung des Quellcodes eines Systems festzustellen, ob sich unerwünschte Seiteneffekte ergeben haben. Dabei ist es unerheblich, ob die Codeänderung aus einem Hinzufügen von neuen Features oder aus dem Beheben eines Fehlers herrührt. Als zweites sollten sie so erweiterbar sein, dass sich der Systemanwendungstest an den letzten Integrationstest anschließt.

Jede erfolgreiche Testkonfiguration sollte gespeichert und mit einer Versionskontrolle versehen werden, bevor neue Testkonfigurationen entwickelt werden. Dies verringert den Aufwand, wenn der Test aufgrund von Designänderungen oder Fehlerkorrekturen erneut durchgeführt werden muss.

Client/Server Systeme basieren oft auf unterschiedlichen Plattformen. So können zum Beispiel Mainframe MVS Server, Windows Clients, Solaris Server und Java Clients verwendet werden. Das primäre Ziel des Client/Server Integrationstests besteht darin, Kompatibilitätsprobleme aufzudecken und zu lösen. Die Integration über mehrere Plattformen erfordert eine kontrollierbare Testumgebung, die alle Plattformen der Zielumgebung umfasst. Debugging- und Test-Tools sind typischerweise plattformspezifisch. Daher ist ein kontrolliertes Debugging und Testen des Systems, welches verschiedene Plattformen umfasst, schwierig.

4.2.6.3 Eingangskriterium

Die Virtual Machine, die in der Testumgebung genutzt wird, läuft stabil. Ist die Virtual Machine noch in der Entwicklung, so sollte die Backbone Integration in Betracht gezogen werden.

Die Komponenten der zu testenden Kollaborationen haben minimale Funktionalitäten, das heißt, sie sollen das Austrittskriterium für ihr zugehöriges Komponentenanwendungstestpattern erfüllen.

Eine physische und funktionale Prüfung sowie eine Prüfung der Umgebung wurden ausgeführt und es wurden keine Abweichungen gefunden, die den Integrationstest behindern oder beeinflussen.

Ein Multiplattform Testtool wurde installiert und ist lauffähig.

Eine Multiplattform Versionskontrolle wurde installiert und wird genutzt.

4.2.6.4 Ausgangskriterium

Jede Treiberkomponente erfüllt das Austrittskriterium für ihr Testpattern.

Das Interface zu jeder Subkomponente wurde wenigstens einmal geprüft.

Die Testfolgen werden in einer Umgebung durchlaufen, die isomorph in Hinblick auf die Zielumgebung ist und deren Test- und Zielumgebung dieselben Eigenschaften haben.

4.2.6.5 Vorteile

Bei der Client/Server Integration werden die Probleme der Big Bang-Integration vermieden. Die Reihenfolge der Client und Server Integration hat wenige Beschränkungen, so dass die Integration nach Priorität oder Risiko aufgeteilt werden kann.

Weiterhin können Treiber und Testfälle wieder verwendet und erweitert werden. Dieser Ansatz unterstützt kontrolliertes, wiederholtes Testen.

4.2.6.6 Nachteile

Ein großer Nachteil der Client/Server Integration sind die Kosten für die Entwicklung der Treiber und Platzhalter. Die Anwendungsfälle können erst spät im Testzyklus getestet werden.

4.2.7 Distributed Services Integration

Unter der Distributed Services Integration versteht man eine Integration verteilter Komponenten. Hierzu gehören zum Beispiel Netzwerkanwendungen.

Ein verteiltes System besteht aus einer Menge autonomer Rechner, welche im Folgenden auch als Knoten bezeichnet werden. Diese Rechner sind gekoppelt und kommunizieren über Nachrichten miteinander.

4.2.7.1 Strategie

4.2.7.1.1 Testmodell

Der Integrationstest von verteilten Systemen befasst sich mit der Prüfung, ob alle Interfaces zwischen den Knoten minimal betriebsfähig sind. Dabei ist es schwierig, eine Testfolge zu finden, die das gesamte Systemverhalten abdeckt. Im schlechtesten Fall ist jeder Knoten mit jedem anderen verbunden und jedes Interface ist einzigartig. Mit den Knoten A, B und C existieren die Interfaces AB, BA, BC, CB, CA und AC. Bei n Knoten ergeben sich damit $n*(n-1)$ Interfaces. Ist die Richtung der Interfaces irrelevant, so reduziert sich deren Anzahl auf $(n*(n-1))/2$. Bei dieser Konfiguration ist der Testaufwand proportional zur Anzahl der Interfaces

Es gibt verschiedene Sequenzen, wie Komponenteninterfaces getestet werden können. Einige Optionen sind im Folgenden kurz erläutert:

- *Risikogesteuert*: Beginne mit den Komponenten und Interfaces, die am problematischsten erscheinen. Das Hauptinteresse liegt auf Komponenten, die von anderen Komponenten, Objekten, der Middleware oder anderen Ressourcen abhängig sind, die
 - Individuell instabil und unerprobt sind.
 - Nicht gezeigt haben, dass sie vorher zusammenarbeiten.
 - Komplexe Businessregeln implementieren.
 - Eine komplexe Implementierung haben (zum Beispiel das 10 fache der Codegröße wie alle anderen Komponenten).
- *Risikoabgeneigt*: Beginne mit den Komponenten und Interfaces, die am unproblematischsten erscheinen. Versuche so früh wie möglich, eine minimale Durchführbarkeit zu demonstrieren. Das Hauptinteresse liegt auf Komponenten, die von anderen Komponenten, Objekten, der Middleware oder anderen Ressourcen abhängig sind, die
 - Individuell sind und sich bewährt haben.
 - Gezeigt haben, dass sie vorher zusammenarbeiten.
 - Einfache Businessregeln implementieren.

- Eine geringe Implementierungskomplexität oder -größe besitzen.
- Ohne Änderungen wieder verwendet werden können.
- *Abhängigkeitsgesteuert*: Beginne mit den Komponenten und Interfaces, die isoliert ausgeführt werden können oder die geringste Abhängigkeit zu anderen Komponenten haben. Ermittle ihre Durchführbarkeit. Bestimme als nächstes Anwendungsfälle, die diese Komponenten unterstützen oder fast unterstützen. Beziehe die zusätzlichen Komponenten mit ein, die diesen Anwendungsfall unterstützten und teste den minimalen Anwendungsfall. Fahre auf diese Art und Weise fort, bis alle Interfaces geprüft sind.
- *Prioritätsgesteuert*: Beginne mit den Komponenten und Interfaces, die erforderlich sind, um ein hohes Dringlichkeitspotential zu demonstrieren. Zeige die Durchführbarkeit dieser Features. Selektiere als nächstes andere Features oder Interfaces, die die anderen Interfaces mit möglichst wenigen Schritten testen.

4.2.7.1.2 Testprozedur

Die generellen Ziele und Strategien für den Integrationstest von verteilten und nicht verteilten Systemen sind meist die Selben. Die Kontrolle und Überwachung der Komponenten eines Remote Hosts kann sich als zeitaufwendig und schwierig gestalten. Treffen auch noch verschiedene operationale Systeme aufeinander, so entstehen oft signifikante Kompatibilitätsprobleme.

Läuft ein Test nicht in einem verteilten System, so soll die Ursache schnell gefunden werden. Schlug der Test aufgrund der Konfiguration der Testumgebung fehl? Oder ist es ein Fehler in der Implementierung? Die folgenden Prinzipien sollen nach [1] bei der Determinierung helfen:

- *Die Testfolge sollte in separat ausführbare Testfolgen unterteilt werden, die Host/Knoten-unabhängig und Host/Knoten-abhängig sind.* Alle Testfolgen müssen automatisiert und komplett wiederholbar sein.
- *Für jeden typischen Remote Host sollte eine Konfigurationsspezifikation entwickelt werden.* Eine typische Remote Host Konfiguration repräsentiert eine Klasse von möglichen Remote Hosts. Während des Integrationstests ist es selten notwendig, jede physische Instanz eines Remote Hosts auszuführen. Trotzdem sollte jeder Hosttyp mit einer deutlich verschiedenen Hardware/Software Plattform und deutlich verschiedener Zuteilung von Applikationskomponenten als eine distinkte Konfiguration betrachtet werden.
- *Auch sollte geprüft werden, ob die Testware, die die Tests implementiert, auf allen typischen Remote Hosts läuft.* Es sollte versucht werden, jeden Remote Host in der Testumgebung zu installieren und zum Laufen zu bringen. So einfach sich dieser Prozess auch anhört, sind es meist mehrer Monate Arbeit für eine nicht geringe Anzahl an Programmierern. Die Schwierigkeit der Konfiguration der Testumgebung sollte hierbei nicht unterschätzt werden.
- *Die Testfolge und die Testware sollten zuerst mit der einfachsten möglichen Konfiguration debuggt werden.* Die grundlegenden, verteilten Objektinterfaces sollten zuerst auf einer einzigen Maschine oder einer beständigen, homogenen Client/Server Architektur ausgeführt werden. Dies erleichtert das Debugging und resultiert in einer grundlegenden Testfolge. Ist die grundlegende Testfolge für diese Konfiguration stabil, so können weitere hinzugefügte Hosts/Knoten getestet werden.

- *Füge immer nur einen typischen Remote Host zur Testkonfiguration hinzu.* Durchlaufe zuerst den hostunabhängigen Grundlagentest. Durchlaufe anschließend hostabhängige Tests. Füge hostspezifische Tests hinzu.
- *Stoppe, wenn alle Interfaces und Typen von Remote Hosts ausgeführt wurden.* Der Integrationstest soll nicht versuchen, die gesamte Funktionalität nachzuweisen und zu prüfen oder die Performance zu überwachen. Es kann sinnvoll sein, einen minimal aufspannenden Baum für die logischen Netzwerkinterfaces anzulegen.

Abbildung 4.24 zeigt 6 typische Remote Hosts. Einige haben lokal kontrollierte Interfaces (eine einfache Linie), andere nutzen das Internet (Wolken).

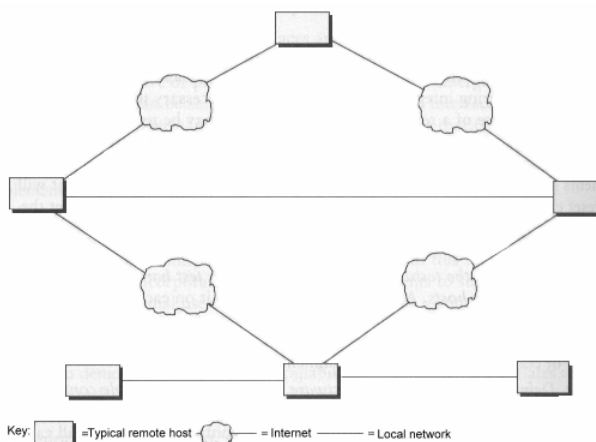


Abbildung 4.24: Beispiel einer verteilten Architektur (Quelle: [1])

Die „einer nach dem anderen“ Abfolge ist in Abbildung 4.25 dargestellt. Einige Platzhalter sind notwendig. Dieses Diagramm enthält einen einzigen Treiber der zeigt, dass die Tests von einer Plattform kontrolliert werden. Zusätzliche Testware und Treiber werden wahrscheinlich für jeden Remote Host benötigt, sind aber der Einfachheit halber nicht im Diagramm aufgeführt. Im Schritt 6 sind alle typischen Remote Hosts geprüft, aber nicht alle Interfaces wurden ausgeführt.

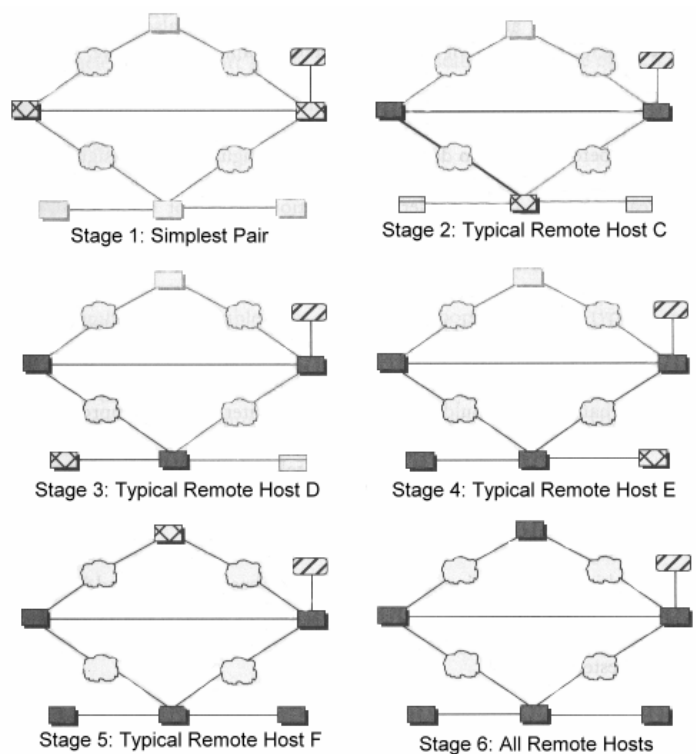


Abbildung 4.25: Schritte und Konfigurationen einer Integration von verteilten Remote Hosts
(Quelle: [1])

4.2.7.2 Treiber und Platzhalter

Bei den Treibern und Platzhaltern kann auf die Überlegungen der Client/Server Integration zurückgegriffen werden. Wie bei der Client/Server Integration sollten auch bei der Distributed Services Integration die Testskripte so entwickelt werden, dass sie die 2 Arten der Wiederverwendung, wie in Kapitel 4.2.6.2 beschrieben, unterstützen.

Verschiedene Tools sind vorhanden, die ein exaktes Binärimage einer Festplatte von einem Remote Host generieren. Diese Utilities sind hilfreich, um einen Remote Host in einen bestimmten Zustand zu versetzen bevor ein Testdurchgang erfolgt.

Auch bei der Distributed Services Integration sollte jede erfolgreiche Testkonfiguration gespeichert und mit einer Versionskontrolle versehen werden, bevor neue Testkonfigurationen entwickelt werden.

4.2.7.3 Eingangskriterium

Die Virtual Machine, die in der Testumgebung genutzt wird, läuft stabil. Ist die Virtual Machine noch in der Entwicklung, so sollte die Backbone Integration in Betracht gezogen werden.

Die Komponenten der zu testenden Kollaborationen haben minimale Funktionalitäten. Sie sollen das Austrittskriterium für ihr zugehöriges Komponentenanwendungstestpattern erfüllen.

Eine physische und funktionale Prüfung sowie eine Prüfung der Umgebung wurden ausgeführt und es wurden keine Abweichungen gefunden, die den Integrationstest behindern oder beeinflussen.

Ein Multiplattform Testtool wurde installiert und ist lauffähig.

Eine Multiplattform Versionskontrolle wurde installiert und wird genutzt.

Die typischen Remote Host Konfigurationen wurden spezifiziert, installiert und getestet.

Die ausgewählte Testware wurde bei jedem Remote Host installiert und konfiguriert.

4.2.7.4 Ausgangskriterium

Das Interface zu jeder Komponente in jedem typischen Remote Host wurde wenigstens einmal geprüft.

Die Testfolgen werden in einer Umgebung durchlaufen, die isomorph in Hinblick auf die Zielumgebung ist.

4.2.7.5 Vorteile

Ein Vorteil der Distributed Services Integration ist die Erschlagung der Probleme der Big Bang Integration. Die Reihenfolge der Client und Server Integration hat wenige Beschränkungen und die Integration kann nach Priorität oder Risiko aufgeteilt werden. Weiterhin können Treiber und Testfälle wieder verwendet und erweitert werden. Dieser Ansatz unterstützt kontrolliertes, wiederholbares Testen.

4.2.7.6 Nachteile

Die Nachteile der verteilten Integration liegen in der kostenintensiven Treiber- und Platzhalterentwicklung sowie im Aufbau der gesamten Testumgebung. Erst in der Mitte des Entwicklungszyklus sind Tests möglich. Der Aufbau einer Testumgebung, die typische Remote Hosts enthält, kann sich als schwierig, teuer und zeitaufwendig erweisen.

4.2.8 Backbone Integration

Unter einem Backbone versteht man eine zentrale Kommunikations- und Transportschicht.

Die Backbone Integration kombiniert Elemente der Top-Down Integration, der Bottom-up Integration und der Big Bang Integration, um die straffe Interoperabilität zwischen leicht gekoppelten Teilsystemen nachzuweisen.

Eingebettete Systemapplikationen und ihre Infrastruktur (zum Beispiel Backbones) werden oft zeitgleich entwickelt. Die Applikationssysteme können nicht ohne die Backbones operieren. Meist werden durch die Backbones Services bereitgestellt, die notwendig für Testdurchgänge und die Applikationen selber sind.

Das Bestreben, die gesamten Backbones durch Platzhalter zu realisieren, würde unpraktisch sein bzw. es wären sehr komplexe Platzhalter erforderlich. Die Platzhalter würden somit entweder sehr kostspielig oder mit einer zweifelhaften Genauigkeit abgebildet werden.

4.2.8.1 Strategie

4.2.8.1.1 Testmodell

Das Problem beim Erstellen von Tests ist die Identifizierung von Komponenten, welche die Applikationskontrolle sowie die Backbone- und Applikationssysteme unterstützen. Die Reihenfolge des Testens basiert auf der folgenden Analyse:

- Die erste und möglicherweise zweite Kontrollstufe werden top-down getestet. Interfaces zum Backbone und anderen Subsystemen werden durch Proxies oder Platzhalter ersetzt.
- Die Applikationssubsysteme sind bottom-up entwickelt und nutzen Testdesignpattern, die die Komponentenzuständigkeit kennzeichnen.
- Die Backbonekomponenten werden isoliert unter einem Treiber getestet und das Testdesign entspricht ihrer spezifischen Zuständigkeit.

Die getesteten Komponenten (Subsysteme) werden dann mittels Big Bang getestet. Diese Big Bang Testfolge sollte alle Kontrolltests, die für die Kontrollkomponente entwickelt wurden, enthalten. Weiterführende Tests sollten entwickelt werden um Szenarien abzudecken, die bei den applikationsspezifischen Subsystemen implementiert sind.

4.2.8.1.2 Testprozedur

Nach Binder [1] wird folgende Testprozedur durchlaufen:

1. Jede Backbone-Komponente wird adäquat und isoliert geprüft. Soweit erforderlich, sollten Treiber und Platzhalter, wie in Abbildung 4.26 gezeigt, genutzt werden.

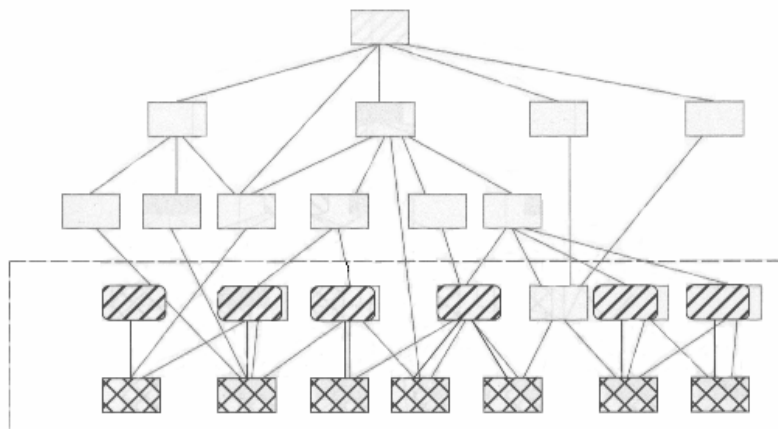


Abbildung 4.26: Backbone Integration, 1. Schritt (Quelle: [1])

2. Eine Top-down Integration bei den Kontrollkomponenten der Applikation wird durchgeführt. Abbildung 4.27 verdeutlicht dieses Vorgehen. Die resultierende Testfolge und die resultierenden Testtreiber sollten so gewählt werden, dass sie für spätere Integrationen wieder verwendet werden können.

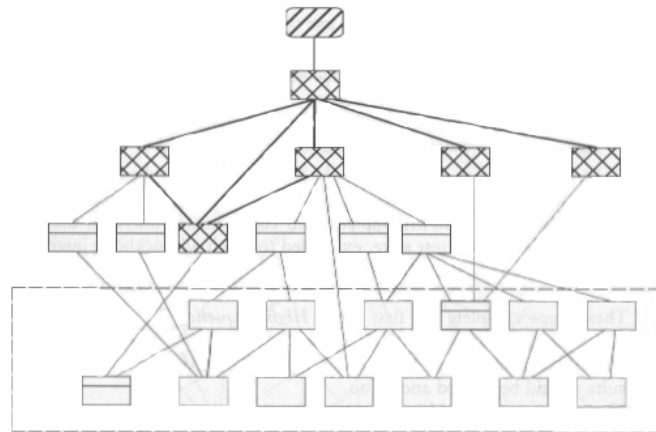


Abbildung 4.27: Backbone Integration, 2. Schritt (Quelle: [1])

3. Ein Big Bang wird auf das Backbone angewandt. Es wird geladen, getestet und wieder heruntergefahren. Dieser Vorgang wird mehrere Male wiederholt.
4. Das Backbone wird unter einem Treiber getestet. Abbildung 4.28 zeigt die Konfiguration für die Schritte 3 und 4.

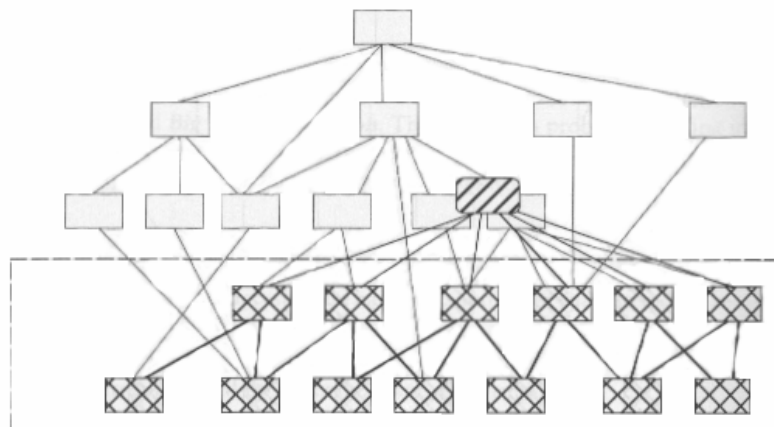


Abbildung 4.28: Backbone Integration, 3. und 4. Schritt (Quelle: [1])

5. Das Kontrollsystem soll mit dem Backbone wieder aufgebaut werden. Die Kontrollkomponenten und das Backbone werden mittels Big Bang zusammengefügt und die Interaktionen mittels der Kontrolltestfolge geprüft.
6. Der Top-down Ansatz wird für die nächste Stufe genutzt. Die Platzhalter für die second-level Komponenten werden entfernt und die Interfaces werden implementiert. Die Kontrolltestfolge wird auf alle neu implementierten Komponenten ausgedehnt und die Tests werden wiederholt. Abbildung 4.29 zeigt die Konfiguration für die Schritte 5 und 6.

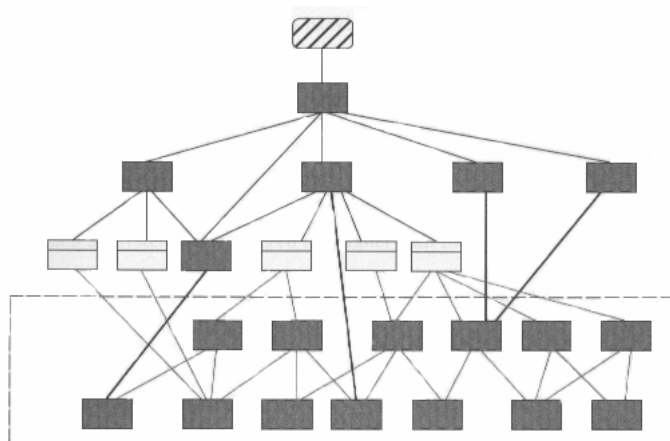


Abbildung 4.29: Backbone Integration, 5. und 6. Schritt (Quelle: [1])

7. Der Top-down Ansatz wird für die noch verbliebenen Stufen angewandt. Alle noch vorhandenen Platzhalter werden von den noch übrig geblieben Teilsystemen entfernt und die Interfaces implementiert. Wie schon in Schritt 6 wird auch im letzten Schritt der Kontrolltest auf die neu implementierten Komponenten ausgedehnt und die Tests laufen wieder an. Die Ausdehnung der Kontrolltests ist erforderlich, um alle Backbone Interfaces zu erreichen, wie in Abbildung 4.30 dargestellt ist.

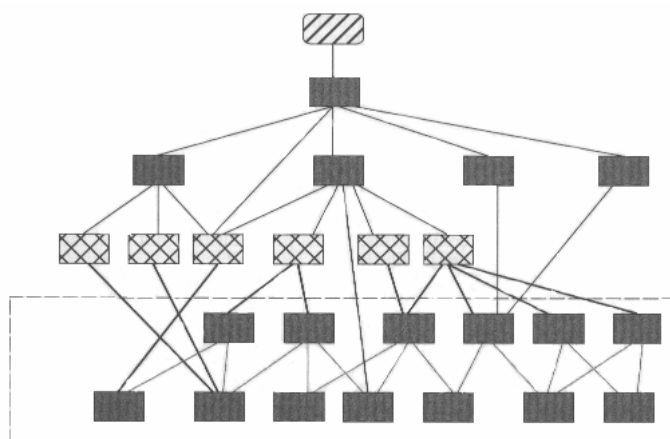


Abbildung 4.30: Backbone Integration, 7. Schritt (Quelle: [1])

4.2.8.2 Treiber und Platzhalter

Dieses Pattern benötigt Treiber und Platzhalter für das Integrationspattern, das für jede Konfiguration angewandt wird: unter anderem Platzhalter für die Clients der Kontrollkomponente, Treiber für die Applikation-Cluster Komponenten.

Wie schon bei den vorangegangenen Pattern sollte auch bei der Backbone Integration eine Versionskontrolle für die Testkonfigurationen erstellt werden, um ein wiederholtes Testen mit verschiedenen Testkonfigurationen zu erleichtern.

4.2.8.3 Eingangskriterium

Die Systemarchitektur und alle Komponenteninterfaces sind entwickelt worden und haben einen sorgfältigen Review durchlaufen.

Das Backbone und die Applikation haben ungefähr dieselbe Größe und Komplexität.

4.2.8.4 Ausgangskriterium

Alle Komponenten erfüllen das Austrittskriterium für die angewandten Testpattern.

Jedes Unterkomponenteninterface wurde wenigstens einmal geprüft.

Die gesamte Erweiterung durchläuft eine Systemanwendungstestfolge.

4.2.8.5 Vorteile

Die Backbone Integration mindert die Nachteile, die bei der Top-down Integration und der Bottom-up Integration auftreten durch Kürzung der Punkte, an denen die Effektivität verloren geht. Die Top-down Integration wird nur bei den höheren Kontrollebenen genutzt und die Bottom-up Integration beschränkt sich auf die Teilsysteme der Applikation.

Alternativ kann zum Testen eines Backbone ein Backbonesimulator entwickelt werden. Obwohl dieser Ansatz gewöhnlich für eingebettete Systeme verwendet wird, besitzen Simulatoren oft eine geringe Wiedergabetreue der Backbone. Das Testen mit einem Simulator kann Fehler identifizieren, die ein Ergebnis des Simulators sind und andere verfehlen, da der Simulator sie nicht triggern kann. Durch die Nutzung von aktuellen Backbones werden diese Probleme und die Entwicklungskosten eines Simulators vermieden.

Die Integration beginnt im Mittelpunkt der Entwicklung. Die minimale Systembedienbarkeit wird auf halbem Wege demonstriert, um das Risiko einer Fehlentwicklung am Ende des Zykluses zu reduzieren. Die Entwicklung und das Testen von Komponenten und Teilsystemen können parallel nach der ersten großen Integration erfolgen.

4.2.8.6 Nachteile

Bei der Backbone Integration ist eine sorgsame Analyse der Systemstruktur und der Abhängigkeiten notwendig. Auch ist sowohl die Entwicklung von Treibern als auch von Platzhaltern unerlässlich. Dieser Ansatz setzt voraus, dass die Bedingungen für ein erfolgreiches Big Bang durch adäquates Testen der Backbone Komponenten erzielt wird. Ist diese Voraussetzung nicht erfüllt, so sind Probleme wie bei der Big Bang Integration wahrscheinlich.

4.2.9 High-frequency Integration

Für den Fall einer schnellen, iterativen Entwicklung von Erweiterungen und deren Integration in bestehende Systeme kann die High-frequency Integration eingesetzt werden. Diese besteht in einer kontinuierlichen Wiederholung der Integration. Folgende Gegebenheiten sind hierfür notwendig:

- Viele bedeutende funktionale Erweiterungen können ohne ein Häufigkeitsintervall erstellt werden. Das heißt, wenn die Integrationshäufigkeit täglich ist, dann ist es sinnvoll, die Erweiterungen größtmäßig so aufzuteilen, dass diese an einem Tag von einem Programmierer realisiert werden können.

- Die Testfolge wird parallel mit dem Quellcode entwickelt. Die Entwicklung der Integrationstestfolge kann nicht aufgeschoben werden. Stattdessen muss sie für den ersten Integrationszyklus verfügbar sein und für jeden erfolgreichen Zyklus gewartet werden.
- Die High-frequency Integration sollte automatisiert werden. Es muss ein zuverlässiges Hilfsmittel für den wiederholten Durchlauf der Integrationstestsuite existieren. Dieses Ziel kann unter anderem mit der Entwicklung von Testtreiberklassen erreicht werden.
- Ein Konfigurationsmanagementtool sollte installiert und genutzt werden. Ohne ein effektives Konfigurationsmanagement würde die Zunahme an Versionen in der High-frequency Entwicklung nicht mehr handhabbar sein.

Die High-frequency Integration ist somit eine Wiederholung der Integration.

4.2.9.1 Strategie

4.2.9.1.1 Testprozedur

Die High-frequency Integration hat 3 Hauptschritte.

Im ersten Schritt erzeugt der Entwickler Codedeltas. Die Codedeltas sollen integriert und mit den Testfolgen verbunden werden. Die nachfolgend aufgelisteten Aufgaben müssen dabei ausgeführt werden:

- Schreibe oder ändere den Code.
- Schreibe oder ändere eine Testfolge für den Code.
- Schreibtischtest, Review, oder überprüfe die geänderte Komponente.
- Schreibtischtest, Review, oder überprüfe die Testfolge.
- Durchlaufe alle Formen der statischen Analyse und löse alle Fehler. Checke die geänderte Komponente ein, um Konflikte im Konfigurationsmanagementsystem mit anderen Versionen oder geänderten Packages zu finden. Nutze alle verfügbaren Quellcodeanalyser.
- Kompiliere den veränderten Code.
- Durchlaufe die Testfolge.
- Hat die Komponente alle Tests durchlaufen, checke den geänderten Code und die Testfolge in den Entwicklerintegrationszweig ein.

Im zweiten Schritt aggregiert der Kompiliererstester die Änderungen des Entwicklers und führt eine Integrationstestfolge aus:

- In einem vereinbarten Intervall stoppt die für die Integration verantwortliche Person die Übernahme der Deltas und kompiliert ein neues System.
- Wenn die Kompilierung erfolgreich abgeschlossen wurde, können die Testfolgen durchlaufen werden. Diese Testfolgen enthalten unter anderem neu entwickelte Tests.

Im dritten Schritt werden Ergebnisse erwartet. Ein effektiver High-frequency Integrationsprozess sollte für die folgenden Fragestellungen eine konkrete Antwort haben:

- Wer ist für die Wartung der existierenden Testfolgen verantwortlich?
- Wie ist der Kompilierungs-Zyklus? Wo ist die Deadline, um eine Änderung zu akzeptieren?

- Wer führt die Übersetzung des Quelltextes und den Integrationstest durch? Unter welchen Umständen?
- Welche Aktionen werden ausgeführt, wenn das Kompilieren aufgrund eines Verweisfehlers fehlschlägt oder ein Test nicht durchgeführt wird? Zu welcher Version kehrt das System zurück? Wie wird das Problem ermittelt und wie wird einer Person oder Organisation die Aufgabe der Fehlerkorrektur zugewiesen?

4.2.9.2 Treiber und Platzhalter

In der Regel wird das Übersetzen des Quelltextes nachts durch Skripte oder Batchjobs ausgeführt. Die gesamte Testfolge sollte dahingehend automatisiert sein, dass sie häufige Wiederholungen unterstützt. Die Fähigkeit, einen nicht durchgeführten Test anzuzeigen und weiterzumachen, ist notwendig für diese Art von Testen. Da die Testfolgen oft nachts automatisch durchlaufen, ist das Vermögen nach einem nicht durchlaufenen Test oder einer abgestürzten Applikation weiterzumachen, kritisch. Deshalb müssen die Testtools in der Lage sein, existierende Tests genauso wieder durchzuführen wie neu entwickelte Tests.

Auch bei der High-frequency Integration sollte eine Versionskontrolle für die Testkonfigurationen erstellt werden, um ein wiederholtes Testen mit verschiedenen Testkonfigurationen zu erleichtern.

4.2.9.3 Eingangskriterium

- Die Virtual Machine, die in der Testumgebung genutzt wird, läuft stabil.
- Eine automatisch wieder anlaufende Testfolge wurde für das Kompilieren entwickelt und der auszuführende Test wurde entwickelt und getestet.
- Die High-frequency Integration kann beginnen, wenn genug Komponenten zur Integration vorhanden sind. Typischerweise entspricht dieser Schritt einer kompletten Kollaboration oder einer Vervollständigung der high-level Kontrollobjekte.
- Die Komponenten unterstehen einer automatisierten Konfigurationsmanagementkontrolle und ein erfolgreich übersetzter Quelltext kann aus dieser Konfiguration generiert werden.
- Eindeutige Kriterien und Prozeduren für das Hinzufügen einer Komponente zum übersetzten Quelltext wurden festgelegt, verifiziert und allen interessierten Parteien mitgeteilt.
- Eine Abschaltung für den Zyklus wird festgesetzt. Es sollte zur selben Zeit (Stunde, Tag, Woche) sein, wenn der Rhythmus, der für dieses Pattern erforderlich ist, festgesetzt wird.

Wurde der Prozess einmal gestartet, so muss die Prozedur für das Hinzufügen von Komponenten von allen Entwicklern befolgt werden.

- Die eingereichte Komponente sollte einen Komponentenanwendungstest durchlaufen haben.
- Die eingereichte Komponente muss die Konfigurationskontrollprozedur für das Kompilieren befolgen.
- Die eingereichte Komponente muss die eingeführten Prozeduren für das Einchecken und die Aktualisierung befolgen. Funktioniert eine Prozedur nicht, sollte sie geändert werden.

4.2.9.4 Ausgangskriterium

Die High-frequency Integration ist beendet, wenn die Entwicklungserweiterung abgeschlossen ist und das zu testende System die Integrationsfolge für die letzte Erweiterung durchlaufen hat. Dabei existieren 2 mögliche Fälle:

- Die Testfolgen der High-frequency Integration werden entwickelt, um eine abgedeckte Integration zu erzielen. Dies ist der Fall, wenn der Kompilierungstest Tests enthält, die durch die Nutzung der Top-down Integration, Bottom-up Integration, Collaboration Integration, Client/Server Integration und so weiter entwickelt wurden. Wenn die letzte Testfolge das Austrittskriterium für solch ein Pattern erfüllt, ist das zu testende System bereit für einen Systemtest.
- Die Testfolgen der High-frequency Integration werden designed, um eine minimale Interoperabilität zu demonstrieren, aber es wird keine Abdeckung der Integration angestrebt. Da das zu testende System ausgeführt wurde, ist es einfach, eine geeignete Integrationsabdeckung zu erzielen. Nach Binder [1] wird die Collaboration Integration Abdeckung vermutlich am besten sein. Wenn dieses Austrittskriterium erfüllt ist, ist das zu testende System bereit für einen Systemtest.

4.2.9.5 Vorteile

Die High-frequency Integration offeriert signifikante Vorteile:

- Es erfordert Verantwortung, um Quellcode und Testfolgen zu entwickeln und zu warten. Die Erstellung von Testfolgen und die Erstellung von Quellcode haben dieselbe Wichtigkeit. Das ist eine effektive Fehlerpräventionsstrategie.
- Große Fehler, Auslassungen und unkorrekte Annahmen werden in den meisten Fällen früh aufgedeckt.
- Da meist die neuesten Hinzufügungen zum übersetzten Quelltext mit Fehlern verbunden sind, ist Debugging einfacher.
- Das gesamte Entwicklerteam ist auf ein System fokussiert, dass arbeitet, anstatt auf ein System, an dem gearbeitet wird. Dieser Ansatz verbessert die Moral, weil das Team konkrete Ergebnisse früh sehen kann.
- Obwohl einige Platzhalter notwendig sein können, sind explizit keine erforderlich.
- Entwicklung und Integrationstest können bis zu einem hohen Grade überlappen.

4.2.9.6 Nachteile

Um den Quellcode und die Testfolgen zu entwickeln und zu warten, ist viel Verantwortung nötig. Der extra Aufwand ist nötig, da viele Nutzer für dieses Pattern einfache Testfolgen entwickeln. Diese Testfolgen haben fragliche Effektivität, besonders wenn die Testfolgen von Entwicklern erstellt wurden, die eine erhebliche Strafe zahlen müssen, wenn sie das Kompilieren unterbrechen.

4.3 Systemtest

Der Systemtest erfasst das Verhalten des Gesamtsystems und prüft es gegen die Systemspezifikation. Die Testumgebung sollte der Produktivumgebung möglichst sehr nahe kommen.

Anstelle von Platzhaltern und Treibern sollte die tatsächliche Hardware und Software zum Einsatz kommen.

Der Systemtest umfasst die folgenden Tests, auf die in diesem Kapitel genauer eingegangen wird:

- Umgebungstest
- Funktionstest
- Test nichtfunktionaler Anforderungen.

4.3.1 Umgebungstest

Beim Umgebungstest wird geprüft, ob das Anwendungssystem zur Zielumgebung passt.

Der Umgebungstest wird nochmals in *den Test der Systemumgebung* und den *Test der Organisationsumgebung* unterteilt.

Der Test der Systemumgebung ist für die Prüfung der „Kompatibilität und Interoperabilität des Systems mit der Systemumgebung“ (Quelle: [13]) zuständig. Unter einer Systemumgebung werden hierbei die Hardware-Konfiguration, die Basissoftware und die Middleware verstanden.

Der Test der Organisationsumgebung prüft die „Kompatibilität und Interoperabilität des Systems mit der Organisationsumgebung“ (Quelle: [13]). Unter einer Organisationsumgebung versteht man die organisatorischen Randbedingungen für den Einsatz eines Produkts. Hierbei ist auch die Frage zu beantworten, ob das System zu den Geschäftsprozessen passt. Auch Fragen hinsichtlich der Berücksichtigung des Datenschutzes, der Zugriffsrechte, der Datensicherung etc. sind zu beantworten.

4.3.2 Funktionstest

Der Funktionstest prüft, ob das System die funktionalen Anforderungen gemäß der Spezifikation erfüllt. Wesentliche Bestandteile des Funktionstests sind der Blackbox-Test und der Feature-Test.

Beim Blackbox-Test, der im Kapitel 2.2.1 genauer erläutert wurde, liegt der Schwerpunkt in der Überprüfung des User Interfaces und der Programmausgaben. Der Feature-Test dient der Prüfung von produktspezifischen Merkmalen (Features), wie zum Beispiel Menü-Optionen.

Für den Funktionstest wird ein genauer Testplan erstellt, in welchem die zu verwendenden Ressourcen sowie die zu testenden Funktionen spezifiziert werden. Oftmals werden dabei nicht alle einzelnen Funktionen getestet, sondern es erfolgt ein Test anhand eines repräsentativen Beispiels. Der Funktionstest beantwortet die Frage, ob Funktionen und Features den Anforderungen des Anwenders und der Spezifikation entsprechen. Eventuelle Fehler und Schwächen sollten in einer Funktionsliste dokumentiert werden, um sie schnellstmöglich zu beheben.

4.3.3 Test nichtfunktionaler Anforderungen

Bestandteil des Systemtests ist ebenfalls der Test von nichtfunktionalen Anforderungen. Dieser lässt sich nach [13] wie folgt untergliedern:

- Beim *Lasttest* bzw. *Volumentest* wird die Frage geklärt, ob das System den maximalen Anforderungen bezüglich Datenmenge, Rechenleistung etc. standhält.
- Der *Performancetest* ist für die „Messung der Verarbeitungsgeschwindigkeit bzw. Antwortzeit für bestimmte Anwendungsfälle“ verantwortlich. Die Messung erfolgt in der Regel in Abhängigkeit einer steigenden Last, zum Beispiel durch Clients.
- Beim *Stresstest* erfolgt die „Beobachtung des Systemverhaltens bei Überlastung“.
- Beim *Test der Datensicherheit* wird getestet, ob ein unberechtigter Systemzugang oder Datenzugriff möglich ist.
- Weiterhin erfolgt ein *Test der Stabilität und Zuverlässigkeit* im Dauerbetrieb.
- Der *Test auf Robustheit* prüft die Fehlerberhandlung und das Wiederanlaufverhalten nach einer Fehlbedienung oder einer Fehlprogrammierung.
- Beim *Test auf Kompatibilität/Datenkonvertierung* erfolgt eine „Prüfung der Verträglichkeit mit vorhandenen Systemen“. Als Beispiel dient hier der Import und Export von Datenbeständen.
- Es erfolgt ebenfalls ein *Test unterschiedlicher Systemkonfigurationen*. Bei diesem Test werden zum Beispiel unterschiedliche Betriebssystemversionen und die Landessprache getestet.
- Der *Test auf Benutzungsfreundlichkeit* prüft die „Erlernbarkeit und Angemessenheit der Bedienung“. Auch auf die Verständlichkeit der Systemausgaben wird großen Wert gelegt.

4.4 Abnahme-/Akzeptanztest

Der Abnahme bzw. Akzeptanztest ist der letzte auszuführende Test im Testablauf. Dabei „wird das Blackbox-Verfahren angewendet, d.h. der Kunde betrachtet nicht den Code der Software, sondern nur das Verhalten der Software bei spezifizierten Handlungen (Eingaben des Benutzer, Grenzwerte bei der Datenerfassung, etc.). Getestet wird anhand eines Prüfprotokolls, ob die in dem Pflichtenheft festgelegten Anforderungen erfüllt werden“ (Quelle: [18]).

4.4.1 Erforderliche Einzeltests für die Abnahme

Der Abnahmetest beinhaltet nach [19] eine Reihe von einzelnen Tests:

- Ein „*Test auf Benutzerakzeptanz* ist dann sinnvoll, wenn der Anwender der Software nicht gleich der Kunde ist“ (Quelle: [20]). Das System wird nicht vom Auftraggeber, sondern von dem oder den späteren Anwendern in Hinblick auf die Erwartungen getestet. Zu den Erwartungen zählen zum Beispiel die Benutzerfreundlichkeit und die übersichtliche sowie verständliche Ergebnisdarstellung.
- „Beim *Test auf vertragliche Akzeptanz* prüft der Kunde, ob die vorliegende Software den Vertrag bzgl. den dort vereinbarten Systemeigenschaften erfüllt“ (Quelle: [20]). Hierbei erfolgt eine Untersuchung der Software in Hinsicht auf Mängel aus Sicht des Kunden. Es ist wichtig, dass die Abnahmekriterien im Vertrag vorher genau festgelegt wurden.

- Beim *Feldtest* wird eine „Vorabversion der Software als sog. Beta-Release ausgewählten Kunden zur Verfügung gestellt“ (Quelle: [20]). Diese Kunden setzen dieses Produkt versuchsweise ein und geben dem Hersteller ein Feedback über die aufgetretenen Fehler während der Nutzung der Software. Bevor die Testversion an die Kunden herausgegeben wird, sollte ein Systemtest von den Softwaretestern der Entwicklungsfirma durchgeführt werden. Auf den Systemtest wurde in Kapitel 4.3 genauer eingegangen.
- Beim *Zielrechnertest* wird geprüft, ob das System auf der Hard- und Softwareumgebung des Kunden läuft.
- Ist die Dokumentation nach den Vorstellungen des Kunden erstellt, so ist der *Dokumentationstest* erfolgreich abgeschlossen.
- Beim *Oberflächentest* wird die Frage geklärt, ob die Oberfläche dem Wunsch des Kunden entspricht.

4.4.2 Fehlerklassifikation

Um die aufgetretenen Fehler besser einzuordnen, ist eine Fehlerklassifikation sinnvoll. Hierfür schlägt [19] die folgende Unterteilung vor:

- Bei *abnahmehinderlichen Fehlern* sind wesentliche Teile, die für den Betrieb und die Nutzung des Systems erforderlich sind, nicht funktionsfähig.
- Bei *schweren Fehlern* können “wesentliche Teile des Systems, die nicht zwingend zum Betrieb, aber zur Nutzung des Systems notwendig sind“ (Quelle: [19]) nicht verwendet werden.
- Bei *mittleren Fehlern* können „Teile des Systems, die unmittelbar weder für den Betrieb noch für die Nutzung des Systems notwendig sind“ (Quelle: [19]), nicht benutzt werden. Das System kann mit geringen Einschränkungen betrieben werden.
- Zu den *leichten Fehlern* zählen Mängel in der Dokumentation, in der Darstellung und im Layout.

4.4.3 Testdurchführung

Der Abnahmetest findet in der Betriebsumgebung statt. Vorausgesetzt wird hierbei, dass das System in einem betriebsbereiten Zustand ist. Der Abnahmetest wird vom Auftraggeber mit seinen Daten durchgeführt. Schon bei der Installation des Systems sollte nach den Anweisungen in der Dokumentation vorgegangen werden, um die Handhabbarkeit dieser zu überprüfen. Weiterhin ist es wichtig, die gelieferten Gegenstände auf Vollständigkeit zu prüfen. Am einfachsten ist dies mit Checklisten möglich. Das Fehlen einzelner Gegenstände kann zum Abbruch des Tests führen. Man spricht dabei von einem sogenannten k.o.-Kriterium. Tabelle 4.6 zeigt ein Beispiel für eine Checkliste:

Liefergegenstand	k.o.-Kriterium	Geliefert
<i>Hardware</i>		
Komponenten	Ja	Ja
Betriebssystem	Ja	Ja
Datenbank	Ja	Ja
Kommunikationssoftware	Ja	Ja
<i>Anwendungssoftware</i>		
Ausführbarer Code	Ja	Ja
Quellcode	Nein	Ja
<i>Dokumentation</i>		
Benutzerhandbuch	Ja	Ja
Installationshandbuch	Ja	Ja
Konfigurationshandbuch	Ja	Ja
Betriebshandbuch	Ja	Ja
Testfälle des Lieferanten	Ja	Ja

Tabelle 4.6: Beispiel für eine Liefergegenstandsscheckliste beim Abnahmetest (Quelle: [19])

Die während des Tests aufgetretenen Fehler sollten mit ihrer zugehörigen Fehlerklasse in einer Tabelle festgehalten werden. Dadurch können die Fehler schneller identifiziert und beseitigt werden. Die Klassifikation der Fehler sollte vertraglich fixiert sein und kann sich an die Klassifikation von Kapitel 4.4.2 anlehnen. Es ist wichtig, vertraglich festzuhalten, was zum Scheitern des Abnahmetests führt. Nach [19] könnte eine Klassifikation für die Verweigerung der Abnahme wie folgt aussehen:

- Bei *abnahmehinderlichen Fehlern* kann die Abnahme verweigert werden.
- Treten mehr als 3 *schwere Fehler* auf, kann die Abnahme ebenfalls verweigert werden.
- Bis zu 10 *mittlere Fehler* dürfen auftreten. Treten mehr auf, kann die Abnahme hier ebenfalls verweigert werden.
- Bei den *leichten Fehlern* kann nach mehr als 30 Fehlern die Abnahme verweigert werden.

Wurden alle spezifizierten Gegenstände geliefert und ist der Test des Systems beim Kunden erfolgreich verlaufen, gilt der Abnahmetest als bestanden.

5 Übersicht Test-Tools

Auf dem Markt sind verschiedene Tools für den Unittest, den Integrationstest, den Systemtest und den Abnahmetest zu finden. In Tabelle 5.1 sind die wichtigsten und am häufigsten verwendeten Tools mit ihrer Zugehörigkeit zu der jeweiligen Testmethode dargestellt. Auf diese Tools wird in diesem Kapitel genauer eingegangen.

	Unittest	Integrationstest	Systemtest	Abnahmetest
JUnit	X	X		
JUnitEE		X		
CruiseContol	X			
Bugkilla			X	X
qftestJUI	X			
Exacum	X			

Tabelle 5.1: Übersicht Testtools

5.1 JUnit

Das Java-Framework JUnit ist ein Tool zum Schreiben und Ausführen von automatischen Unit Tests. Ein Framework ist eine „Menge von kooperierenden Klassen, die einen wieder verwendbaren Entwurf für einen bestimmten Anwendungsbereich implementieren“ (Quelle: [16]). Da die Tests direkt in Java programmiert werden, ist das Testen mit JUnit so einfach wie das Kompilieren. „Die Testfälle sind selbstüberprüfend und damit wiederholbar“ (Quelle: [15]). JUnit kann für den Test von Methoden, Klassen bis hin zu Komponenten verwendet werden.

JUnit wurde als OpenSource Software veröffentlicht. Unter <http://www.junit.org/> kann das Tool heruntergeladen werden.

Nach [3] bietet JUnit:

- „einen einheitlichen Rahmen zur Spezifikation, Implementierung, Organisation, Ausführung und Kontrolle des Tests,
- Unabhängigkeit einzelner Testfälle,
- beliebige Zusammenfassbarkeit von Testfällen,
- sofortige Erkennbarkeit des Testergebnisses und
- Trennung von Anwendungs- und Testcode“.

5.1.1 Aufbau JUnit Framework

Im Folgenden werden die JUnit Klassen vorgestellt, die die meiste Verwendung finden.

5.1.1.1 Assert

Mit JUnit können Werte und Bedingungen getestet werden, die für einen erfolgreichen Testabschluss erfüllt sein müssen. „Die Klasse `Assert` definiert dazu eine Menge von `assert`-Methoden, die unsere Testklassen aus JUnit erben und mit denen wir in unseren Testfällen eine Reihe unterschiedlicher Behauptungen über den zu testenden Code aufstellen können.“ (Quelle: [15]).

- `assertTrue(boolean condition)`: Diese Methode gibt an, ob die Behauptung wahr ist.
- `assertEquals(Object erwartet, Object aktuell)`: Bei dieser Methode wird geprüft, ob zwei Objekte gleich sind.
- `assertEquals(int erwartet, int aktuell)` prüft, ob zwei Integer-Werte gleich sind.
- `assertEquals(double erwartet, double aktuell)`: Bei dieser Methode wird die Gleichheit von zwei Fließkommazahlen geprüft.
- `assertNull(Object wert)`: Die `assertNull()`-Methode prüft, ob eine Objektreferenz null ist.
- `assertNotNull(Object wert)`: Bei dieser Methode wird geprüft, ob eine Objektreferenz nicht null ist.
- `assertSame(Object erwartet, Object aktuell)`: Diese Methode prüft, ob zwei Referenzen auf das gleiche Objekt verweisen.

5.1.1.2 AssertionError

„Die im Testcode durch `assert`-Anweisungen kodierte Behauptungen werden von der Klasse `Assert` automatisch verifiziert. Im Fehlerfall bricht JUnit den laufenden Testfall sofort mit dem Fehler `AssertionFailedError` ab“ (Quelle: [15]).

Die Fehlermeldung kann für alle `assert`-Methoden angepasst werden, das heißt, der Tester gibt einen Erklärungstext an, der im Fehlerfall ausgegeben wird. Anstelle von `assertTrue(boolean condition)` kann die Methode `assertTrue(String nutzerFehlermeldung, boolean condition)` verwendet werden.

5.1.1.3 TestSuite

Mit der Klasse `TestSuite` können mehrere Tests hintereinander ausgeführt werden. Bei JUnit können beliebig viele Tests in einer `TestSuite` zusammengefasst und ausgeführt werden. In einer statischen `suite`-Methode wird definiert, welche Tests zusammen ausgeführt werden sollen. „Eine Suite von Tests wird dabei durch ein `TestSuite` Objekt definiert, dem wir beliebig viele Tests und selbst andere `TestSuite` hinzufügen können“ (Quelle: [15]).

Um eine `TestSuite` zu definieren, wird ein `TestSuite` Exemplar gebildet. Mittels der `addTestSuite` Methode werden verschiedene Testfallklassen hinzugefügt. „Jede Testfallklasse definiert implizit eine eigene `suite` Methode, in der alle Testfallmethoden eingebunden

werden, die in der betreffenden Klasse definiert wurden“ (Quelle: [15]). Dieser Teil wird von JUnit automatisch mittels Java Reflection erledigt.

5.1.2 Beispiel

Als Beispiel für einen Test mit JUnit dient die folgende Java-Klasse Euro:

```
public class Euro {
    // der Eurobetrag
    private double betrag;

    /** Initialisierung des Konstruktors (Speichert den
     * übergebenen Eurobetrag).
     * @param betrag der Eurobetrag
     */
    public Euro(double betrag) {
        //Betrag speichern
        this.betrag = betrag;
    }
    /** Gibt den Eurobetrag zurück.
     * @return den Eurobetrag.
     */
    public double getBetrag() {
        return this.betrag;
    }
}
```

Die folgende Klasse Test testet, ob die in der Methode testEurobetrag() angegebene Behauptung wahr ist.

```
import junit.framework.*;
public class Test extends TestCase {

    /** Testet ob die in der Methode angegebene Behauptung
     * wahr ist.
     */
    public void testEurobetrag() {
```

```
        //Instantiierung Eurobetrag
        Euro two = new Euro(2.00);
        //prüft ob die Behauptung wahr ist
        assertTrue(2.00 == two.getBetrag());
    }
    /** Startet den Test
     * @param args die Argumente
     */
    public static void main(String[] args) {
        //startet die Testklasse 'Test'
        junit.swingui.TestRunner.run(Test.class);
    }
}
```

Die Klasse `Euro` wird mit dem Wert 2.00 instantiiert. Die Behauptung lautet, dass 2.00 den gleichen Wert wie `two.getBetrag()` hat, also 2.00 gleich 2.00 ist. Diese Behauptung ist richtig. In Abbildung 7.1. zeigt JUnit anhand des grünen Balkens, dass der Test erfolgreich durchlaufen wurde.

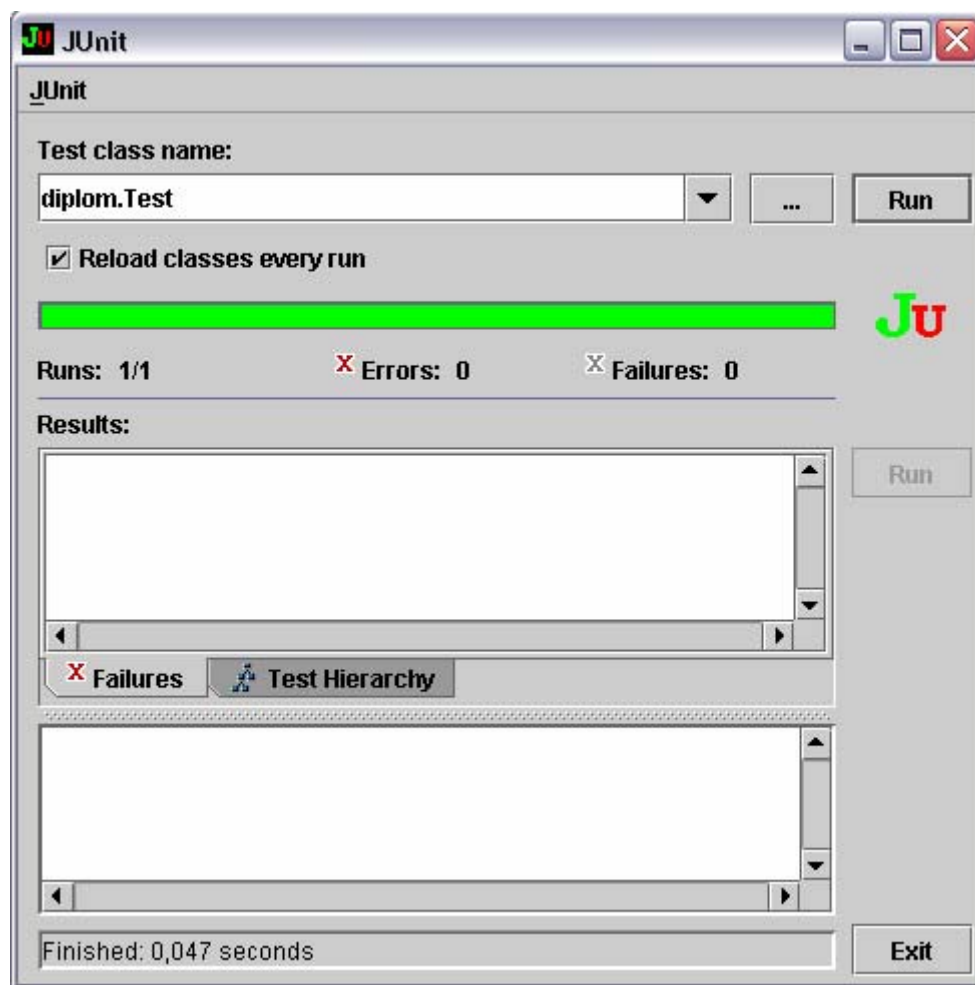


Abbildung 7.1: JUnit – erfolgreicher Testdurchlauf

Der Klasse Test wird die folgende Methode hinzugefügt:

```
/** Testet, ob die Rundung der Werte korrekt erfolgt. */
public void testRundung() {
    //Instantiierung Eurobetrag
    Euro gerundeterWert = new Euro(1.995);
    //prüft ob die Fließkommazahl 2.00 dem gerundeten
    //Wert entspricht
    assertTrue("2.00 entspricht nicht dem gerundeten
    Wert", 2.00 == gerundeterWert.getBetrag());
}
```

Diese Methode prüft, ob der gerundete Wert dem Wert entspricht, mit dem die Klasse Euro instantiiert wurde. Dieser Test schlägt fehl, da die Klasse Euro noch kein Rundungskonzept enthält. Abbildung 7.2. zeigt, dass der Test nicht erfolgreich war, da ein Fehler (Failure) aufgetreten ist. Die spezifizierte Fehlermeldung wird unter dem Punkt ‚Results‘ angezeigt:

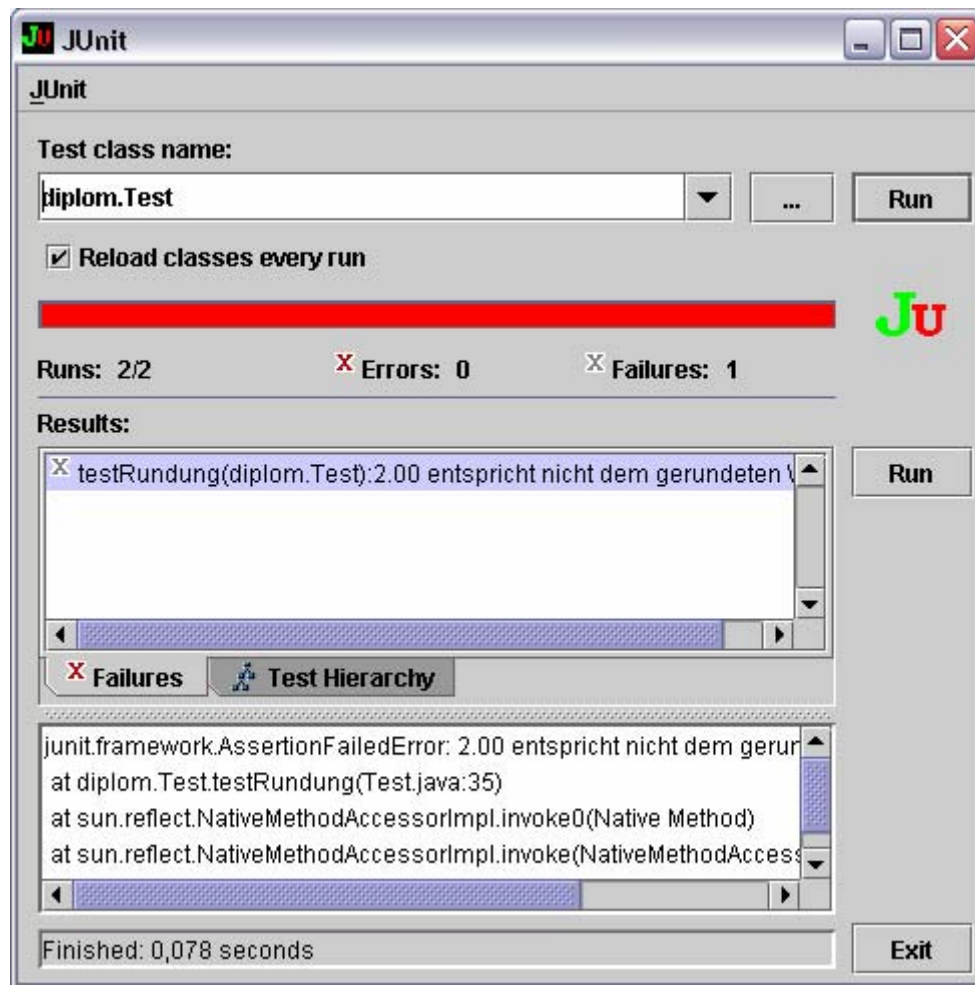


Abbildung 7.2: JUnit – Testdurchlauf nicht erfolgreich

Wird im Konstruktor der Klasse Euro anstatt `,this.betrag = betrag;'` `,this.betrag = Math.round(betrag);'` geschrieben, so wird eine Rundung des Wertes nach mathematischen Regeln vorgenommen und der Test wird erfolgreich abgeschlossen.

Um nicht jeden Test einzeln ausführen zu müssen, können die Tests „gesammelt“ und zusammen ausgeführt werden. In Kapitel 5.1.1.3 wurde auf die Klasse `TestSuite` genauer eingegangen. Hierzu ist eine Methode nötig, in der alle auszuführenden Tests zu der Testsuite hinzugefügt werden:

```
import junit.framework.*;

public class AlleTests {

    public AlleTests() {
    }

    /** Fügt alle auszuführenden Tests einer TestSuite hinzu
        und gibt diese zurück.
```

```
* @return eine Suite von allen auszuführenden Tests
*/
public static junit.framework.Test suite() {
    //instantiiert eine neue TestSuite
    TestSuite suite = new TestSuite();
    //fügt die auszuführenden Tests hinzu
    suite.addTestSuite(Test.class);
    return suite;
}
}
```

5.2 JUnitEE

Das auf JUnit basierende Testframework JUnitEE ermöglicht das Ausführen von Unittests innerhalb eines Application Servers. Die Mitgabe der Tests an die Gesamtapplikation erfolgt als Web Archive. Auf diese Weise ist ein Testen im Produktivsystem auf Abruf möglich. Die Steuerung der Tests erfolgt über ein Servlet. Die Ergebnisse werden im HTML-Format ausgegeben. „Mit JUnitEE als Framework hat man alle Hilfsmittel zur Verfügung, um das Umfeld der zu testenden Komponenten entsprechend zu konfigurieren, z.B. durch Setzen von JNDI Properties“ (Quelle: [17]).

Abbildung 5.1 zeigt ein Beispiel für ein Testergebnis:

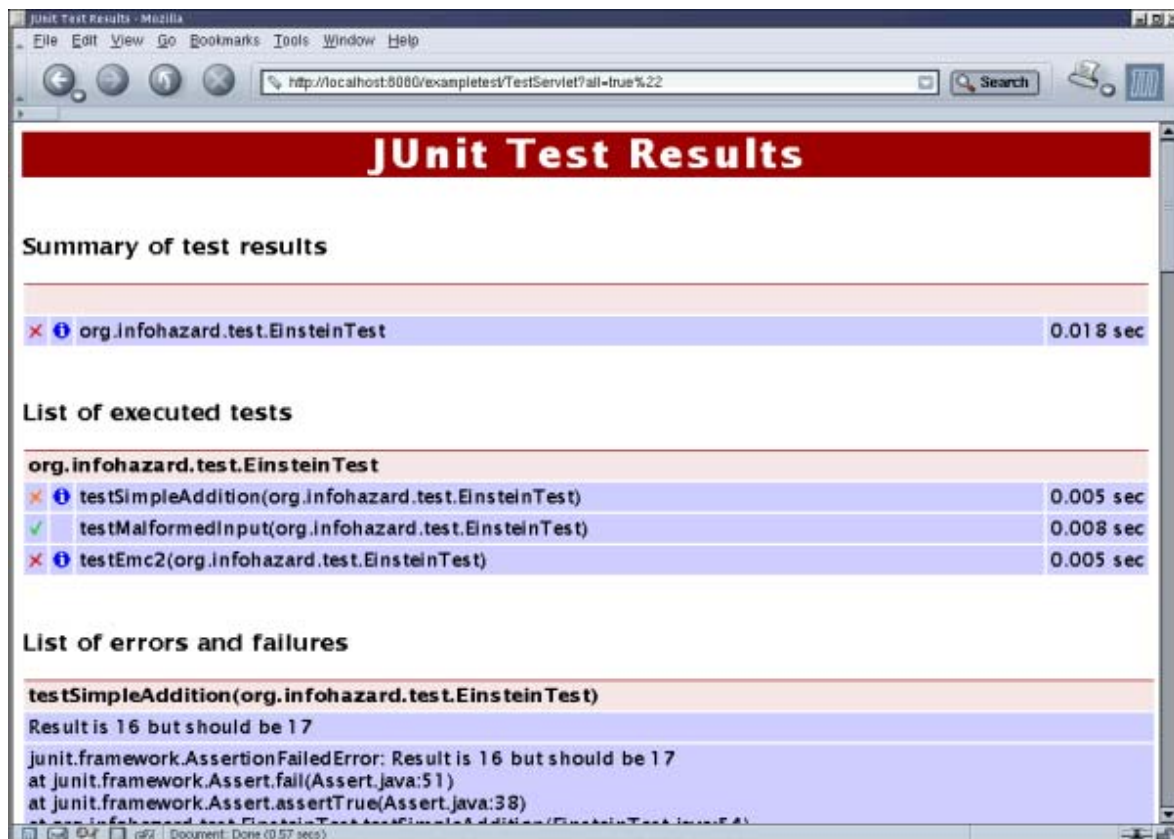


Abbildung 5.1: Testergebnisse JUnitEE (Quelle: [8])

Die Übersicht der Testergebnisse ist in 3 Punkte unterteilt:

- Eine Übersicht zeigt eine Liste von allen ausgeführten Testfolgen und ihrem Ergebnis.
- Für jede Testfolge gibt es eine Liste von ausgeführten Tests und ihren jeweiligen Ergebnissen.
- Für jeden fehlgeschlagenen Test existiert eine detaillierte Beschreibung für den Grund des Fehlers.

Weitere Informationen zu JUnitEE sind unter www.junit.org/ zu finden.

5.3 CruiseControl

CruiseControl ist ein OpenSource Werkzeug zur kontinuierlichen Integration von Systemen. Bei der kontinuierlichen Integration erfolgt meist mehrmals täglich eine Übersetzung der fertigen Systemeinheiten.

CruiseControl besteht aus zwei Kernteilen:

- der Build-Loop und
- der Build Results JSP.

Die Build Loop ist ein Hintergrundprozess, der periodisch prüft, ob Änderungen im zentralen Quellcode durchgeführt wurden. Ist dies der Fall, so erfolgt nach Quelle [17] eine Abarbeitung der im Folgenden genannten Punkte:

- Interaktion mit der Versionsverwaltung um die neueste Version zu erhalten.
- Aufruf aller notwendigen Compile, Link und Package Vorgänge.
- Aufruf aller notwendigen Testläufe.
- Benachrichtigung aller beteiligten Personen über die Resultate des letzten Builds.

Die Konfiguration der Build Loop erfolgt über eine XML-Datei. Wahlweise ist auch eine Verwaltung der Build Loop über ein JMX basiertes Webinterface möglich. Neben der möglichen „Mailbenachrichtigung wird zusätzlich nach jedem Build eine Log Datei erzeugt, die von der Build Results JSP zur Darstellung einer Ergebnis-Historie verwendet wird“ (Quelle: [17]).

Der Kompilierungsprozess kann bei CruiseControl zeitabhängig oder ereignisabhängig gestartet werden. Hierzu wird das Projekt aus dem Sourcecode-Repository ausgecheckt, kompiliert und deployed. Bei aufgetretenen Fehlern wird der Entwickler benachrichtigt, um bis zu einem festgesetzten Termin das Problem zu beheben und die Datei neu einzuchecken oder die Datei zu entfernen.

Abbildung 5.2 zeigt das Ergebnis eines Kompilierungsprozesses. Die linke Seite zeigt, ob CruiseControl noch am Kompilieren des Projektes ist. Weiterhin sind Links dargestellt, die auf Details zu vorangegangenen Kompilierungen verweisen. Auf der rechten Seite der Grafik sind die Ergebnisse der Kompilierung dargestellt. Hierzu zählen Kompilierungsfehler, Testergebnisse und Details darüber, welche Dateien sich seit der letzten Kompilierung geändert haben.

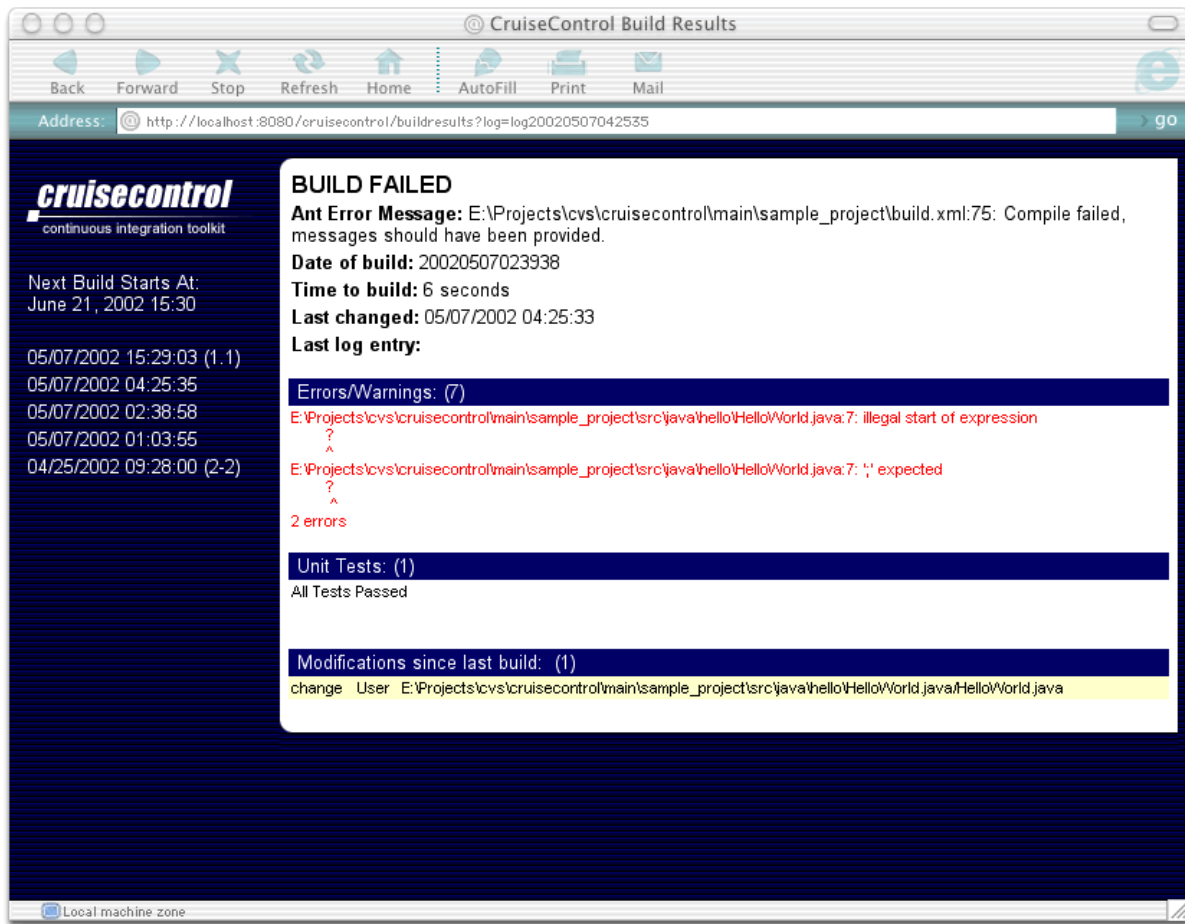


Abbildung 5.2: Ergebnis Kompilierungsprozess CruiseControl (Quelle: [9])

Weitere Information über CruiseControl sind unter <http://cruisecontrol.sourceforge.net/> zu finden.

5.4 Bugkilla

Die OpenSource Software Bugkilla unterstützt die Erstellung, Pflege und Ausführung von automatisierten, funktionalen Akzeptanztests für eine J2EE Web-Anwendung. Eine Automatisierung erfolgt bei folgenden Gebieten:

- Spezifikation von Testfällen.
- Unterstützung der Instantiierung von Testfallspezifikationen.
- Management und Erstellung der Testdaten.
- Kontrolle der Terminierungskriterien.
- Testadministration.
- Analyse, Validierung und Messung von Tests.
- Durchführung des Tests.

Mehr über Bugkilla ist unter <http://bugkilla.sourceforge.net/> zu finden.

5.5 qftestJUI

„qftestJUI ist ein Werkzeug zur Erstellung, Ausführung und Verwaltung von automatisierten Tests für Java/Swing Anwendungen mit grafischer Benutzeroberfläche (GUI)“ (Quelle: [25]). Es läuft unter Windows und Unix mit den JDKs von Sun, IBM und Blackdown in den Versionen 1.1 bis 1.5

„qftestJUI registriert die Reaktionen der Anwendung auf die simulierten Aktionen, z.B. das Öffnen eines neuen Fensters in Folge eines Mausklicks auf einen Knopf“ (Quelle: [23]). Weiterhin können zum Beispiel Tabellenwerte ausgelesen werden und mit Vorgaben verglichen werden. qftestJUI ist in Java programmiert und kann so für plattformunabhängige Tests verwendet werden.

Tests lassen sich einfach über die graphische Benutzeroberfläche erstellen. Die Aufnahme eines manuellen Tests wird mittels eines Mausklicks gestartet. Es werden Maus und Tastaturaktionen sowie die Reaktion der Anwendung aufgezeichnet. Die Tests können beliebig oft wiederholt werden, ohne dass der Tester in den Test erneut eingreifen muss. Bei der Aufnahme erfolgt eine automatische Integration der Elemente des GUI, der Aktionen des Anwenders und der zugehörigen Daten in eine Baumstruktur. Der entstandene Baum spiegelt die hierarchische Struktur des GUI der Anwendung wieder. Auch ist der Baum das zentrale Element der grafischen Oberfläche von qftestJUIs. „Er ermöglicht den schnellen Zugriff auf jedes Detail der aufgezeichneten Information und gibt eine gute Übersicht über die Daten und deren Zusammenhänge“ (Quelle: [23]).

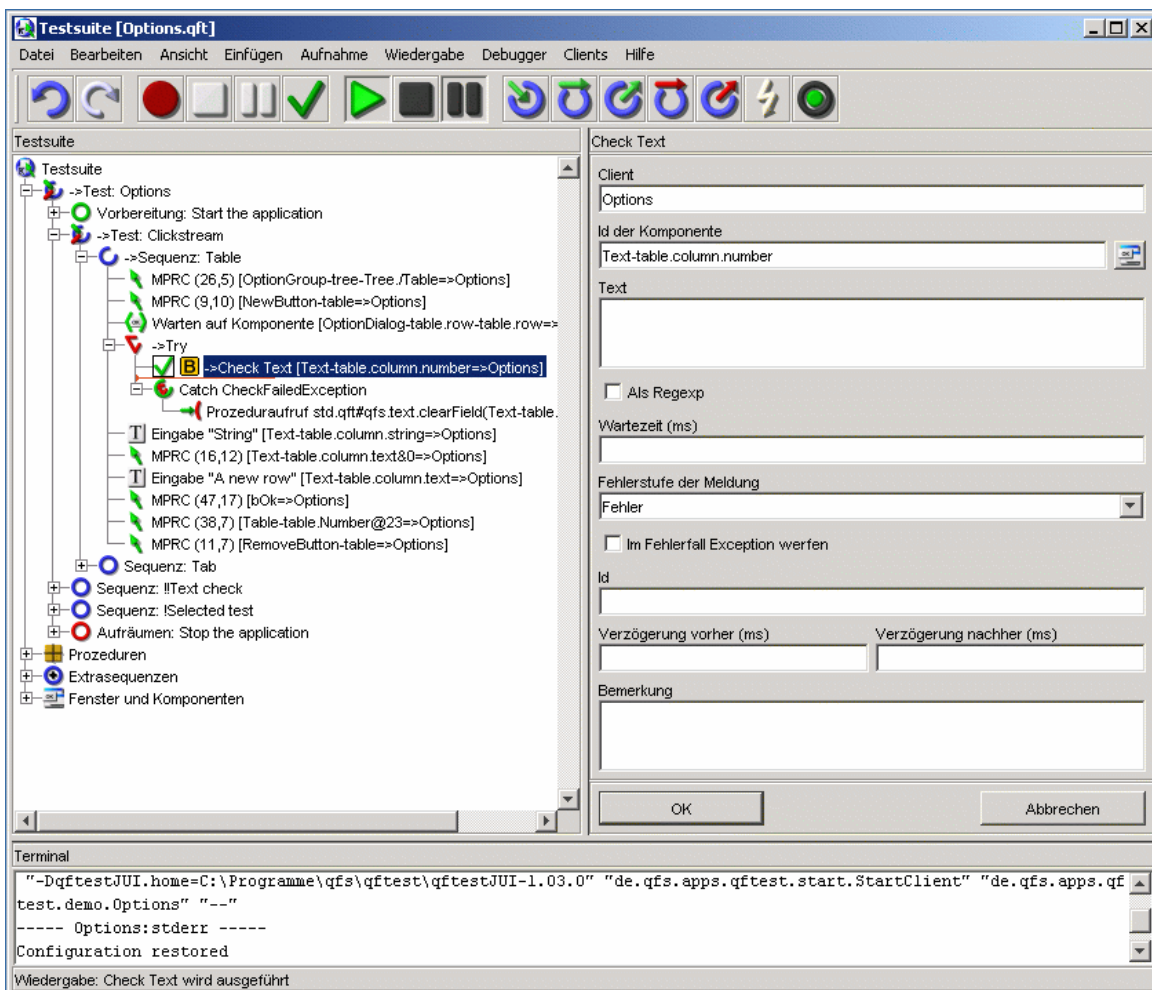


Abbildung 5.3: Hauptfenster qftestJUI (Quelle: [25])

Das Programm `qftestJUI` und weitere Informationen sind unter <http://www.qfs.de/de/qftestJUI/index.html> zu finden.

5.6 Exacum

Mit Exacum können unkompliziert und effektiv Java-Klassen und Module getestet werden. „Es muß lediglich ein Testprojekt erzeugt und mit den Verzeichnispfaden der zu testenden Anwendung initialisiert werden“ (Quelle: [11]). Die Analysierung der Klassen erfolgt in Exacum automatisch. Die Methoden der Klassen können unmittelbar in einem einfachen Testschritt ausgeführt werden. Mit Exacum können Objekte konstruiert, Methoden mit beliebigen Parametern ausgeführt und die Ergebnisse sofort ausgewertet werden.

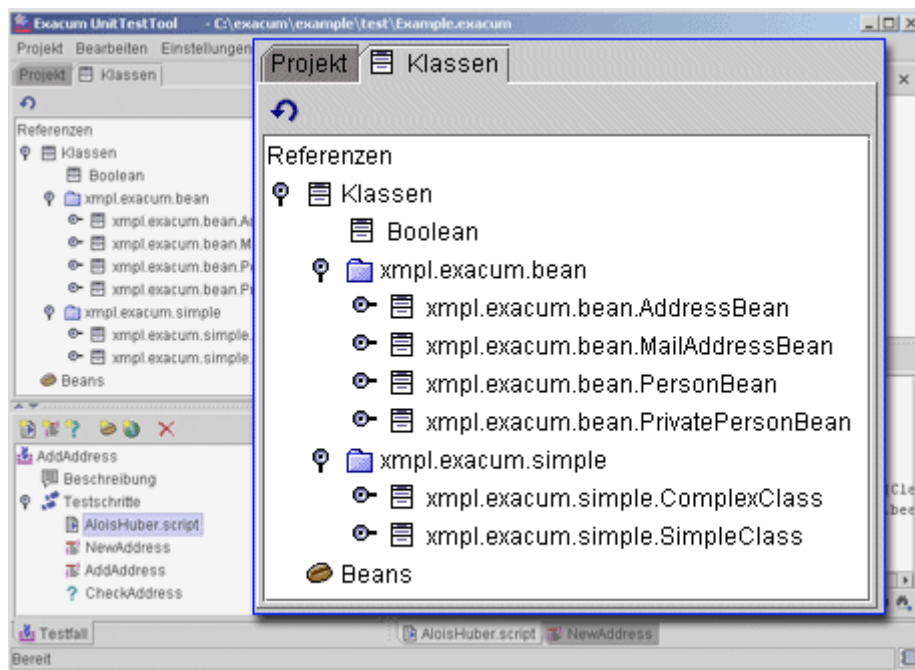


Abbildung 5.4: Hauptfenster Exacum (Quelle: [11])

Der Testablauf sieht nach [4] wie folgt aus:

- 1. Schritt: Ausgangssituation erstellen
- 2. Schritt: Methodenaufruf von Java Objekten
- 3. Schritt: Prüfung des Ergebnisses in Form einer Bedingung.

Im 1. Schritt wird ein neuer Testfall erstellt. Zum Herstellen der Ausgangssituation wird ein Testschritt vom Typ *Skript* erstellt. Das *Skript* enthält „Mengen initialisierter und miteinander verknüpfter Objekte als Ausgangssituation für einen Test“ (Quelle: [22]), die im Objektmanager konstruiert werden. Jeder Testlauf hat einen eigenen Objektmanager. Damit wird eine Beeinflussung von Testfällen und deren Testläufen ausgeschlossen.

Im 2. Testschritt können Java-Methoden an Java-Objekten aufgerufen werden. Hierbei wird einfach die zu testende Methode aus der zu testenden Klasse ausgewählt. Der zu erwartende Datentyp wird in einer Parametertabelle automatisch gesetzt; der Tester kann einen Testwert vom selben Datentyp eingeben.

Im letzten Schritt wird eine Bedingung definiert, die erfüllt sein muss. Dabei kann zum Beispiel geprüft werden, ob der in der Parametertabelle angegebene Testwert einem bestimmten definierten Wert entspricht.

Sind diese 3 Schritte erfolgreich durchlaufen worden, ist ein Testfall definiert, welcher sowohl in der Entwicklungsphase als auch in der Testphase verwendet werden kann. Solange ein Testlauf noch nicht abgeschlossen ist, können immer wieder Testläufe durchgeführt und auch erweitert werden. Wird ein Testlauf abgeschlossen, erfolgt eine Ablage im Dateisystem, ein erneutes Durchführen von Testläufen ist nicht möglich.

Weiterführende Information zu Exacum sind unter <http://www.ist-dresden.de/products/Exacum/index.html> zu finden.

6 Testframework für Java

6.1 Was ist Java?

Java wurde von der Firma Sun entwickelt und erstmals am 23. Mai 1995 als neue, objektorientierte, einfache, plattformunabhängige und sichere Programmiersprache vorgestellt.

Die Programmiersprache lehnt sich sehr stark an die C-übliche Syntax an. Im Gegensatz zu C wird bei Java der Quellcode jedoch nicht direkt in ausführbaren Maschinencode übersetzt, sondern zunächst in einen Bytecode, welcher maschinenunabhängig ist. Dieser Bytecode kann anschließend von einer maschinenabhängigen Java Virtual Machine (JVM) interpretiert und ausgeführt werden. Damit ist ein in Java geschriebenes Programm auf allen Systemen ausführbar, für welche eine JVM zur Verfügung steht.

Neben der Plattformunabhängigkeit bietet Java einige weitere wichtige Vorteile. So wird durch ein weit reichendes Sicherheitskonzept gewährleistet, dass Java Programme nur auf bestimmte, autorisierte Systemressourcen, wie z.B. Dateien oder Sockets, zugreifen können. Weiterhin wird der Zugriff auf Arrays durch die JVM überwacht. Damit ist es nahezu unmöglich, Pufferüberläufe zu erzeugen, die das System von außen angreifbar machen können. Ebenfalls ein Vorteil von Java besteht in dem automatischen Speichermanagement. Durch den Einsatz eines Garbage Collectors werden nicht mehr benötigte Speicherbereiche von diesem selbständig erkannt und wieder freigegeben.

Als Nachteil bei Java ist die Performance anzusehen. Die Interpretation des Bytecodes und die Übersetzung in Maschinencode bei der Ausführung von Programmen wirken sich negativ auf das Laufzeitverhalten aus. Weiterhin kann die Performance durch den Einsatz des Garbage Collectors beeinträchtigt werden. Dieser läuft in einem eigenen Hintergrundthread und kann während der Speicherbereinigung den Ablauf des eigentlichen Java-Programms verzögern.

6.2 Besonderheiten beim Testen

Wie schon in Kapitel 3.3 beschrieben, gibt es in Java Zugriffsmodifizierer, die Zugriffsrechte der Klassen und Methoden regeln. Werden für das Testen zusätzliche Testklassen verwendet, können diese in einem anderen Java-Package als die zu testenden Klassen definiert sein. In diesem Fall können die Testklassen nur auf *public* deklarierte Klassen und Methoden zugreifen. Ein Test der als *protected* bzw. *private* deklarierten Klassen und Methoden ist somit nicht möglich.

Weiterhin sollten Java-Programme trotz des Einsatzes des Garbage Collectors auf Memory-leaks getestet werden, die zum Beispiel durch die Verwendung von Listnern auftreten können. Ein Listener ist ein Objekt, welches in einem anderen Objekt über eine entsprechende Methode registriert wird, um von diesem Ereignis zu empfangen. Wurde solch ein Listener Objekt registriert, so sollte die Registrierung auch wieder aufgehoben werden, sobald das Listener Objekt nicht mehr verwendet wird. Geschieht das nicht, so kann der Garbage Collector das Listener Objekt nicht bereinigen, da es immer noch von einem anderen Objekt refe-

renziert wird. Somit wird der eigentlich nicht mehr benötigte Speicher nicht wieder freigegeben und bei häufigem Auftreten könnte dies letztlich Programmabstürze zur Folge haben.

Besonders zu beachten ist beim Testen von Java Programmen auch die Verwendung von expliziten Castings. Zwar gibt es in Java eine strenge Typisierung, jedoch können mit Hilfe eines Cast-Operators auch Typumwandlungen vorgenommen werden. Dieser Operator wird durch einen in Klammern geschriebenen Datentyp definiert, wie:

```
Object o = "ein Text";  
// korrektes Casting als String  
String s = (String) o;
```

Diese expliziten Castings stellen in Java eine potentielle Fehlerquelle dar, da sie nicht vom Compiler überprüft werden. So würden sich folgende Anweisungen zwar übersetzen lassen, zur Laufzeit jedoch einen Fehler verursachen:

```
Object o = "ein Text";  
// löst eine ClassCastException aus, da die Variable o  
// nicht vom Typ Integer ist  
Integer i = (Integer) o;
```

Zu berücksichtigen ist beim Einsatz von Java auch die Verwendung der richtigen JVM in der richtigen Version. Insbesondere muss bei Client/Server Anwendungen darauf geachtet werden, dass sowohl die Client- als auch die Serveranwendung für die richtige Ziel JVM übersetzt worden sind. Werden verschiedene Versionen genutzt, können Kompatibilitätsprobleme auftreten und die Applikationen sind nicht zusammen lauffähig.

Da es in Java möglich ist, Bibliotheken zu verwenden, muss auch hier auf die korrekte Version dieser geachtet werden. Wird eine Bibliothek durch eine neuere ersetzt, kann nicht davon ausgegangen werden, dass alle verwendeten Methoden mit den Neuen übereinstimmen.

7 Konzept für ein integriertes Java-Testtool

Im folgenden Kapitel wird ein Konzept für ein integriertes Werkzeug entwickelt, welches für den Test von Java-Applikationen verwendet werden kann.

7.1 Anforderungen

Das integrierte Testtool soll den gesamten in Kapitel 4 dargelegten Testablauf abdecken. Dabei werden 3 Testphasen unterschieden:

- der Unit-/Integrationstest
- der Systemtest und
- der Abnahmetest.

Das Tool bietet eine graphische, intuitiv bedienbare Benutzeroberfläche. Zentrales Element ist dabei eine Projektverwaltung. Hier können Testprojekte angelegt, bearbeitet und auch gelöscht werden. Bestandteile eines Testprojekts sind die einzelnen Testphasen. Ist ein Projekt angelegt bzw. geöffnet, kann ausgewählt werden, für welche Testphase ein neuer Test angelegt werden soll oder ob ein bereits Vorhandener bearbeitet wird. Für die Tests sowie für die Ergebnisse der einzelnen Tests ist eine Versionsverwaltung unumgänglich. Diese versieht die Tests und die Testergebnisse automatisch mit einer Versionsnummer beim Speichern. Die Darstellung der Testergebnisse erfolgt in einer Tabelle.

Die Speicherung der angelegten Tests und Testergebnisse erfolgt in einer Datenbank oder in einem XML-Dokument. Arbeiten mehrere Entwickler an den Tests, so empfiehlt sich die Verwendung einer Client/Server Architektur, um einen gleichzeitigen Multiuserbetrieb zu gewährleisten.

Für die Testdokumentation kann ein HTML-Report für jedes gespeicherte Testergebnis erstellt werden. Auch ein Vergleich von zwei ausgewählten Testergebnissen soll möglich sein.

7.2 Unit-/Integrationstest

Für den Unittest werden separate Testklassen verwendet. Bei der Verwendung dieser Testklassen wird zunächst eine Testkonfiguration erstellt, die aus einer Menge von Objekten mit definierten Zuständen besteht und damit das zu testende System widerspiegelt. Für den Test werden anschließend Methoden in den Objekten aufgerufen, wobei das korrekte Systemverhalten geprüft wird. Implizit erfolgt hierbei auch ein Integrationstest, da bei den Methodenaufrufen unter Umständen auch auf andere Komponenten zugegriffen wird. Deshalb können der Unittest und der Integrationstest als eine Teststufe zusammengefasst werden.

Als Basis für diese Testphase kann das in Kapitel 5 näher beschriebene OpenSource Tool JUnit eingebunden werden. Die Codierung der Testklassen erfolgt wie in Kapitel 5.1 beschrieben. Die Einbindung der Testergebnisse in das integrierte Testtool ist wie folgt möglich:

```
public void doTest(Test toTest) {
    //erzeugt ein neues TestResult
    TestResult result = new TestResult();
    //durchlaufen des Tests und speichern der
    //Ergebnisse im TestResult
    toTest.run(result);

    //Verarbeitung der Errors
    processErrors(result);

    //Verarbeitung der Failures
    processFailures(result);
}
```

In der Methode `processErrors(result)` werden die Errors weiterverarbeitet. Die Ausgabe der Fehler könnte in einer graphischen Oberfläche oder einfach in einer Tabelle erfolgen. Analog dazu arbeitet die Methode `processFailures(result)`, die für die Weiterverarbeitung der Failures zuständig ist. Der Testname, die verwendeten Testklassen und die gewonnenen Testergebnisse, ggf. die Fehlermeldungen und der Fehlerstatus, werden mit einer fortlaufenden Versionsnummer gespeichert.

Durch die Verwendung von Listnern in der Klasse `TestResult` könnte eine kontinuierliche Aktualisierung des Teststatus im Testtool während der Testausführung, zum Beispiel in Form eines Fortschrittbalkens, erreicht werden.

7.3 Systemtest

Wie schon in Kapitel 4.4 beschrieben, wird der Systemtest in den Umgebungstest, den Test der funktionalen Anforderungen sowie den Test der nichtfunktionalen Anforderungen unterteilt.

Beim Umgebungstest ist es sinnvoll, einen Überblick über alle benötigten Bibliotheken und ihre Versionen zu haben. Damit kann geprüft werden, welche Bibliotheken sich bereits auf dem Rechner befinden und welche noch zu installieren sind.

Alle für das System spezifischen Anforderungen werden übersichtlich in einer Tabelle dargestellt, die beliebig erweiterbar ist. In einer Spalte kann dabei angegeben werden, ob die Anforderungen erfüllt sind oder nicht. Als Beispiel für mögliche Anforderungen dient Tabelle 7.1:

Anforderung	Vorhanden	Bestanden
<i>Hardware</i>		
Pentium III 550 mHz	Pentium II 200 MHz	Nein
512 MB RAM	1024 MB RAM	Ja
<i>Software</i>		
Java Version 1.5	Java Version 1.5	Ja
Betriebssystem Windows	Betriebssystem Linux	Nein

Tabelle 7.1: Mögliche Kriterien für einen Umgebungstest

Der Funktionstest prüft, ob das System die funktionalen Anforderungen erfüllt. Hierbei werden die einzelnen Funktionen anhand eines Beispiels geprüft. Da es nicht möglich ist, alle Funktionen zu überprüfen, sollte das Beispiel so gewählt werden, dass möglichst viele Funktionalitäten abgedeckt sind. Für den Funktionstest ist eine Checkliste empfehlenswert, die die Eingabewerte, die Ausgabewerte und die zu erwartenden Werte aufnimmt. Diese Werte sollten bei jedem neuen Test zur Verfügung stehen, um abweichende Ausgabewerte sofort zu erkennen. Damit ist eine schnelle Korrektur der Fehler möglich.

Als letztes sollten die nichtfunktionalen Anforderungen geprüft werden. Für den Performance-Test kann das JMX (Java Management eXtension) verwendet werden. Damit ist es unter anderem möglich, Informationen über die Speicherauslastung und über laufende Threads abzufragen, die wichtige Indikatoren in Bezug auf die Performance sind. Treten Unregelmäßigkeiten auf, so kann dies unter anderem auf die in Kapitel 6.2 beschriebenen Memoryleaks hinweisen. Die spezifizierten Anforderungen sowie die bei der Abfrage gewonnenen Ergebnisse können in einer Tabelle dargestellt und gespeichert werden. Der Tester hat weiterhin die Möglichkeit, die Anforderungen als bestanden oder nicht bestanden zu markieren. Weiterführende Informationen zu JMX sind unter <http://java.sun.com/products/JavaManagement/reference/docs/index.html> [Stand: 03.02.2005] verfügbar.

7.4 Abnahme-/Akzeptanztest

Für den Abnahmetest wird eine Checkliste in Tabellenform generiert. Eine Versionierung ist auch hier notwendig, da der Abnahmetest scheitern kann oder bei Softwareupdates erneut ausgeführt werden muss. Ist bereits eine Version einer Checkliste vorhanden, so kann die zuletzt angelegte Version als Vorlage für eine neue Version dienen; ist keine Version vorhanden, so kann eine vordefinierte Liste geladen werden. Diese Default-Checkliste kann sich an die Tabelle 4.6 aus Kapitel 4.4.3 anlehnen und bei Bedarf beliebig erweitert bzw. verkürzt werden:

Liefergegenstand	k.o.-Kriterium	Geliefert
<i>Anwendungssoftware</i>		
Benötigte Java-Version	Ja	Ja
Benötigte Bibliotheken	Ja	Ja
Ausführbarer Code	Ja	Ja
Setup	Ja	Ja
Quellcode	Nein	Ja
<i>Dokumentation</i>		
Benutzerhandbuch	Ja	Ja
Installationshandbuch	Ja	Ja
Konfigurationshandbuch	Ja	Ja
Betriebshandbuch	Ja	Ja
Testfälle des Lieferanten	Ja	Ja

Tabelle 7.2: Mögliche Checkliste für den Abnahmetest einer Java-Applikation

8 Zusammenfassung / Ausblick

Der Softwaretest ist einer der am häufigsten unterschätzten Punkte bei der Softwareentwicklung. Da das Testen von Software einen beträchtlichen Teil der Entwicklungszeit in Anspruch nimmt, wird es, trotz seiner enormen Wichtigkeit, häufig vernachlässigt. Diese Vernachlässigung macht sich meist durch höhere Fehlerquoten im Endprodukt bemerkbar.

Trotz sorgfältiger und umfangreich durchgeführter Tests kann eine fehlerfreie Software nicht garantiert werden. Die Tests sollten regelmäßig und schon von Beginn der Implementierung an erfolgen, um Spezifikationsfehler schneller zu erkennen und zu beheben. Auch das Aufdecken und Korrigieren von Implementierungsfehlern und falsch verstandenen Anforderungen ist in einem frühen Stadium der Entwicklung möglich. Je früher ein Fehler gefunden wird, umso geringer sind die Kosten für die Korrektur.

Die vorgestellten Methoden für die objektorientierten Sprachen vereinfachen die Suche nach Fehlern. Werden die Testphasen sorgfältig durchlaufen, ist eine umfangreiche Überprüfung gewährleistet, was im Ergebnis zu einer deutlichen Reduzierung der Fehler führt.

Das dargelegte Konzept für ein integriertes Testtool umfasst den gesamten Testablauf vom Unittest bis zum Abnahmetest. Tests können bereits während der Implementierung durchgeführt und dokumentiert werden. Aufgrund der Verwendung von frei verfügbaren Technologien kann das vorgeschlagene Konzept mit geringem Aufwand und geringen Kosten umgesetzt und beliebig weiterentwickelt werden.

Da die Komplexität der Software auch in Zukunft immer weiter zunehmen wird, muss der Testablauf kontinuierlich an die neuen Bedürfnisse angepasst werden, um Fehlerquellen aufzudecken und eine schnelle Korrektur zu gewährleisten.

Literaturverzeichnis

- [1] R.Binder, Testing object-oriented systems, Addison Wesley, 2000
- [2] R. Dumke, Software Engineering. Vieweg, 2001, 3. Auflage
- [3] <http://www4.in.tum.de/~pretschn/teaching/testsem02-ausarb/kortkamp-ootest.pdf>
[Stand: 08.01.2005]
- [4] <http://exacum.de/products/Exacum/guide/concept/guide/quick/script.html> [Stand: 18.01.2005]
- [5] http://www.inf.fu-berlin.de/inst/ag-se/teaching/V-SWT-2003/65_2LE_18tw.pdf
[Stand: 08.01.2005]
- [6] <http://mitglied.lycos.de/alpia/berichte/testen.htm> [Stand: 08.01.2005]
- [7] <http://ivs.cs.uni-magdeburg.de/~dumke/EAD/Skript16.html> [Stand: 08.01.2005]
- [8] <http://www.junit.org/usersguide.html> [Stand: 08.01.2005]
- [9] <http://cruisecontrol.sourceforge.net/reporting/jsp/buildresultsjspscreenshot.gif> [Stand: 08.01.2005]
- [10] <http://www.vorgehensmodelle.de/giak/arbeitskreise/vorgehensmodelle/themenbereiche/qualitaetsmanagement.html> [Stand: 08.01.2005]
- [11] <http://www.ist-dresden.de/products/Exacum/guide/concept/guide/intro/developer.html>
[Stand: 05.07.2004]
- [12] <http://ebus.informatik.uni-leipzig.de/www/media/pups03/pups-V3-ppt.ppt> [Stand: 08.01.2005]
- [13] <http://ebus.informatik.uni-leipzig.de/www/media/lehre/st-test04/st-test04-ve11-pdf.pdf>
[Stand: 08.01.2005]
- [14] http://www-dssz.informatik.tu-cottbus.de/~wwwdssz/information/testen/05f_modaleKlassen_santen_4up.pdf [Stand: 08.09.2004]
- [15] <http://www.frankwestphal.de/UnitTestingmitJUnit.html> [Stand: 08.01.2005]
- [16] http://129.69.81.237/lernapplikation/info3/html/Info3_Kap5_01/tsld009.htm [Stand: 05.11.2004]
- [17] <http://www.oio.de/j2ee-testtools.htm> [Stand: 08.01.2005]
- [18] <http://akzeptanztest.adlexikon.de/Akzeptanztest.shtml> [Stand: 08.01.2005]
- [19] <http://www.software-kompetenz.de/?17364> [Stand: 08.01.2005]

- [20] <http://www.software-kompetenz.de/?10135> [Stand: 08.01.2005]
- [21] http://www.fbi.fh-darmstadt.de/~schuette/Vorlesungen/ProgrammierenII/Skript/03_KlassenUndObjekte/KlassenUndObjekte.pdf [Stand: 15.09.2005]
- [22] <http://exacum.de/products/Exacum/guide/concept/guide/script/language.html> [Stand: 18.01.2005]
- [23] <http://www.qfs.de/de/qftestJUI/proddesc.html> [Stand: 08.01.2005]
- [24] http://de.wikipedia.org/wiki/Polymorphie_%28Programmierung%29 [Stand: 08.01.2005]
- [25] <http://www.qfs.de/de/qftestJUI/index.html> [Stand: 08.01.2004]

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Diplomarbeit selbständig und nur mit erlaubten Hilfsmitteln angefertigt habe. Weiterhin erkläre ich, dass ich die Diplomarbeit keiner anderen Prüfungsbehörde vorgelegt habe.

Schönebeck, 16.02.2005

Yvonne Seitschek