Konzeption und Realisierung eines datenbankgestützten Testwerkzeugs zur Überdeckungsanalyse von Java-Programmen

Diplomarbeit

Lehrgebiet Praktische Informatik III
Software Engineering
Fachbereich Informatik
FernUniversität Hagen

Eingereicht von Hans-Gerd Wefels

12. Februar 2003

Betreuer:
Prof. Dr. H.-W. Six
Prof. Dr. M. Winter

Abstract

In dieser Arbeit wird zunächst ein Überblick über die kommerziell und im Opensource-Bereich erhältlichen Testwerkzeuge zur Überdeckungsanalyse von Java-Programmen gegeben. Nach einem Vergleich der Eigenschaften dieser Werkzeuge erfolgt der Entwurf eines Instrumentierers für Java-Programme auf der Grundlage des von B. Bokowksi und A. Spiegel an der Technischen Universität Berlin entwickelten Opensource-Parser-Paketes Barat. Der im Rahmen der vorliegenden Arbeit entwickelte Instrumentierer ist als Datenbankanwendung implementiert, er ist leicht an jedes relationale Datenbankmanagementsystem anpassbar und trägt den Namen ‚DoiT' (Datenbank zur Dokumentation von instrumentierten Testläufen von Java-Programmen). Mit ‚DoiT' ist es möglich, viele Testläufe verschiedener Prüflinge verschiedener Versionen in einer oder mehreren Datenbanken dauerhaft zu dokumentieren. ‚DoiT' ist in der Lage Messungen zur Ermittlung verschiedener Überdeckungsmetriken an Java-Programmen, bis hin zur minimalen Mehrfachüberdeckung, vorzunehmen. Darüber hinaus ist eine Visualisierungskomponente enthalten, mit deren Hilfe Daten aus durchgeführten Testläufen aus der Datenbank ausgelesen und zusammen mit dem zugehörigen Programm-Quelltext angezeigt werden können.

Dank

Besonderen Dank schulde ich neben meinen Betreuern Prof. Dr. Winter und Prof. Dr. Six, Herrn Dr. Boris Bokowski/Bonn, der den Parser Barat auf meinen Wunsch erweitert und auch sonst jederzeit hilfreich und kritisch meine Ideen zur Anwendung von Barat begleitet hat. Ebenso danke ich Herrn Ian Darwin/Palgrave, Ontario für seine Hilfe bei einigen technischen Problemen mit Java.

Ganz besonders möchte ich meiner Ehefrau, Ulrike Wefels, für das Korrekturlesen der vorliegenden Arbeit und für die Geduld, mit der sie mein Informatik-Studium in den letzten Jahren ‚erduldet' und unterstützt hat, danken.

Inhaltsverzeichnis

1 Einleitung und Motivation des Testens von Software 1

1.1 Testparadigmen 1

1.2 Überdeckungstests und Anforderungen an entsprechende Werkzeuge 3

1.2.1 Anweisungsüberdeckung C_0 4

1.2.2 Zweigüberdeckung C_1 4

1.2.3 Bedingungsüberdeckung C_2 4

1.2.4 Minimale Mehrfachbedingungsüberdeckung 4

1.2.5 Objektorientierte Überdeckungsmessungen 5

1.3 Zielsetzung und Überblick 6

2 Spezielle Anforderungen beim Überdeckungstesten von Java Programmen 8

2.1 Problematische Spracheigenschaften 8

2.1.1 Variablendefinitionen 8

2.1.2 Implizite Ausführung von Funktionen und Operationen 8

2.1.3 Konditionaloperator 10

2.1.4 Blöcke ohne Methode 10

2.1.5 Methoden und Blöcke – unstrukturierter Kontrollfluss 11

2.1.6 Klasseninstantiierung und Konstruktoren 11

2.1.7 Anonyme Klassen 11

2.1.8 Vererbung und Instrumentierung 11

2.1.9 Dynamisches Laden von Klassen 12

2.2 Vereinfachende Spracheigenschaften 12

2.2.1 Try-Finally 12

2.2.2 Reflektivität bzw. Selbstinspektion 12

2.2.3 Initialisierer 12

2.2.4 Variablendefinition und Gültigkeitsbereich 13

3 Verfügbare Werkzeuge 14

3.1 Kommerziell verfügbare Testwerkzeuge 14

3.1.1 McCluskey-Tools 14

3.1.2 JCover von Man Machine Systems 15

3.1.3 JavaScope von SUN 15

3.1.4 Clover von Cortex 16
3.1.5 Borlands OptimizeIt Code Coverage 16
3.1.6 Code Coverage von Koalog 17
3.1.7 Testers Edge GJ-Cover 17
3.1.8 Bewertung der kommerziellen Produkte 18
3.2 Testwerkzeuge und Parser aus dem Opensource-Bereich 19
3.2.1 JIE von E. Tromer 19
3.2.2 Barat von B. Bokowski und A. Spiegel 20
3.2.3 JUnit und Hansel 20
3.2.4 Bewertung der Opensource Produkte 20
4 Anforderungsermittlung 22
4.1 Instrumentierung von Klassen 23
4.2 Instrumentierung von Dateien 24
4.3 Instrumentierung von Paketen 25
4.4 Durchführung eines instrumentierten Testlaufes 26
4.5 Evaluierung von Testläufen instrumentierter Klassen 27
4.6 Evaluierung von Testläufen instrumentierter Dateien 28
4.7 Evaluierung von Testläufen instrumentierter Pakete 28
4.8 De-Instrumentieren 30
4.9 Anforderungen an die Dokumentation der Testläufe und an die Struktur der Datenbank 30
5 Konzeption und Grobentwurf 31
5.1 Architektur von Barat 31
5.2 Visitor Konzept 32
5.3 Architektur von ‚DoiT' 34
5.3.1 Die grafische Benutzungsschnittstelle ‚DoitGraph' 36
5.3.2 Die Klassen ‚barat_test' und ‚barat_mess' 36
5.3.3 Die Datenbankschnittstelle ‚Test_db' 36
6 Feinentwurf und Implementation 38
6.1 Datenbankschema zur Ablage der Messergebnisse 38
6.2 Struktur von Barat-ASTs 41
6.3 Instrumentierende und evaluierende Barat-Visitoren und entsprechende Rahmenprogramme 43
6.3.1 Besonderheiten und Probleme des Entwurfs 43

6.3.2 Entwurf des Instrumentierers ‚barat_test' und des ‚InstrumentingVisitors' 47
6.3.3 Entwurf des Evaluierers ‚barat_mess' und des ‚MeasuringVisitors' 51
6.4 Entwurf der Benutzungsschnittstelle 53
6.4.1 Instrumentieren 55
6.4.2 Evaluieren 55
6.4.3 Datenbankoptionen und Datenbankverwaltung 55
6.4.4 Weitere Funktionen der Benutzungsschnittstelle 56
6.5 Entwurf der Datenbankschnittstelle 56
6.5.1 Öffnen und Schließen der Datenbank 56
6.5.2 Dokumentation der Instrumentierungspunkte 58
6.5.3 Messwerterfassung 58
6.5.4 Lokale Messwertausgabe 58
6.5.5 Globale Messwertausgabe 59
6.5.6 Informationen für ‚DoitGraph' 59
7 Beispiele 60
7.1 Allgemeine Vorgehensweise beim Instrumentieren und bei der Überdeckungsanalyse mit ‚DoiT' 60
7.2 Messungen an der ‚DoiT'-GUI 62
7.3 Messungen an und mit JUnit 63
8 Zusammenfassung und Ausblick 68
9 Glossar 71
10 Literaturverzeichnis und eingesetzte sonstige Hilfsmittel 74
11 Internet-Links 76
12 Anhang 77
12.1 Installation 77
12.2 Hinweise zur Weiterentwicklung von ‚DoiT' 78
12.2.1 Entwicklungsumgebung 78
12.2.2 Java-Grammatik 78
12.2.3 Erweiterung von ‚DoiT' 78
12.2.4 Weitere Datenbankmanagementsysteme 79
12.3 Benutzerhandbuch 79
12.4 Einsatzmöglichkeiten der Batch-Version 89
12.5 Datenlexikon der ‚DoiT'-Datenbank 92

Verzeichnis der Abbildungen

Abbildung 1: Beispiel für implizite Funktionsaufrufe 9
Abbildung 2: Beispiel für Zuweisung in einem Bedingungstest 9
Abbildung 3: Zur Short-Circuit-Evaluation 9
Abbildung 4: Workaround für Fehler, die bei der Short-Circuit-Evaluation instrumentierter Ausdrücke auftreten können 10
Abbildung 5: Beispiel für die Verwendung eines Konditionaloperators 10
Abbildung 6: Beispielhaftes Kollaborationsdiagramm zum Visitor-Konzept (nach [BoSpie1998]) 33
Abbildung 7: Beispiel für einen Barat Visitor 34
Abbildung 8: Grobarchitektur von ‚DoiT' 35
Abbildung 9: E/R Diagramm der ‚DoiT' Datenbank 40
Abbildung 10: Ausschnitt aus einem Barat-AST, dargestellt als UML Klassendiagramm 42
Abbildung 11: Klasse ‚InstrumentingVisitor' 47
Abbildung 12: Algorithmus zum Zerlegen zusammengesetzter logischer Ausdrücke 49
Abbildung 13: Klasse ‚MeasuringVisitor' 51
Abbildung 14: Klassendiagramm des Pakets ‚doit' 52
Abbildung 15: Klassendiagramm der graphischen Benutzungsoberfläche 54
Abbildung 16: Klasse ‚Test_db' 57
Abbildung 17: Ablaufplan für die Instrumentierung mit ‚DoiT' 62
Abbildung 18: Beispiel für den Objekt-Kontext einer Methode 67
Abbildung 19: Auswahl der Quelldateien 81
Abbildung 20: Paketweises Instrumentieren 82
Abbildung 21: Zur Auswahl beim datei-/klassenweisen Instrumentieren 83
Abbildung 22: Beispielausgabe der ‚DoiT'-Konsole 84
Abbildung 23: Auswahl der zu evaluierenden Testläufe 85
Abbildung 24: Beispiel für Test-Evaluierung mit ‚barat_mess' 86
Abbildung 25: Einstellung der ‚DoiT' Datei-Pfade 87
Abbildung 26: Dialog zur Datenbank-Anbindung 88
Abbildung 27: Dialog zur Datenbank-Verwaltung 89
Abbildung 28: Struktur der Datei ‚DB_props.ini' 90

1 Einleitung und Motivation des Testens von Software

„*Software ist entweder so einfach, dass sie offensichtlich keine Fehler enthält oder so komplex, dass sie keine offensichtlichen Fehler enthält.*“ [IntHoare]

„*Testen kann die Anwesenheit von Fehlern zeigen, aber nicht deren Abwesenheit.*“ [DaDiHo1972]

Diese bekannten Zitate machen deutlich, wie wichtig das systematische Testen von Software ist. Man hofft, durch Testen so viele Fehler wie möglich aufzudecken. Dazu existieren verschiedene Ansätze, die in der folgenden Einführung kurz angerissen werden sollen.

Die möglichen Auswirkungen fehlerhafter Programme motivieren die Notwendigkeit rigoroser Tests hinreichend. Programme steuern Fertigungsprozesse und Kernkraftwerke, tätigen Finanztransaktionen und greifen regelnd in Verkehrsabläufe und die Funktion von Verkehrsmitteln ein, um nur ein paar Beispiele aufzuzählen. Aber auch aus betriebswirtschaftlicher Sicht kommt dem Testen von Software große Bedeutung zu, denn ein ausgetestetes Produkt verursacht weniger Wartungsaufwand und geringere Garantieleistungen, und letztendlich verkauft es sich wegen seines zuverlässigeren Rufs bei den Anwendern auch besser. Aus dem zweiten Zitat oben leitet sich sofort das Ziel des Testens her: Ziel des Testens ist es Fehler zu finden und nicht die Aussage, dass die Software „wahrscheinlich so gut wie keine Fehler enthält“ [PagSix1994]. Definitionen zu den Begriffen „Fehler“, „Testen“ und einigen anderen, im folgenden häufig benutzten Termini wurden ebenfalls [PagSix1994] entnommen und im Glossar am Ende dieser Arbeit zusammengefasst.

1.1 Testparadigmen

[PagSix1994] unterscheiden mehrere Testparadigmen. Zunächst werden statische und dynamische Testverfahren gegeneinander abgegrenzt. Die statischen Verfahren kommen gänzlich ohne Ausführung des Prüflings aus und werden durch ‚Lesen’ der Software durch Review-Teams und Durchspielen von Testszenarien durchgeführt. Im Gegensatz dazu setzen dynamische Verfahren die Ausführbarkeit des Prüflings voraus. Dabei soll das Programm nach [Lig1990]

- mit konkreten Testdaten ausgeführt werden
- und in der realen Einsatzumgebung getestet werden (in sicherheitssensitiven Umgebungen in einem entsprechenden Simulator).

Eine weitere Prämisse nach [Lig1990] von dyn. Verfahren ist:

- Es handelt sich um ein Stichprobenverfahren, das die Korrektheit (im Sinne der Programmverifikation) nicht beweisen kann.

Anhand der Systematik zur Auswahl der Testfälle werden strukturorientierte, funktionsorientierte und diversifizierende dynamische Tests unterschieden. Bei strukturorientierten Tests können die Testfälle kontroll- oder datenflussbezogen abgeleitet werden, funktionsorientierte Tests benutzen die Spezifikation und diversifizierende Verfahren vergleichen Resultate unterschiedlicher Programmversionen.

In dieser Arbeit wird der Schwerpunkt auf die dynamischen Verfahren und hier insbesondere auf die kontrollflussbezogenen Testverfahren gelegt. Kontrollflussbezogene Verfahren sind ‚White-Box-Verfahren', wobei das Programm in erster Linie gegen sich selbst und nicht gegen eine Spezifikation getestet wird. Die Verfahren zielen darauf ab, eine bestimmte Menge von Programmausführungspfaden beim Testen zu durchlaufen. Man spricht daher auch von Überdeckungstests. Allgemein haben Überdeckungstests den Sinn, nicht ausgeführte Programmteile und Funktionalitäten aufzuspüren, denn bei den Tests nicht ausgeführter Code ist ungetesteter Code!

Dabei werden folgende Varianten und Kriterien unterschieden:

- Die Anweisungsüberdeckung (C_0-Test) ist das einfachste Kriterium.
- Die Zweigüberdeckung (C_1-Test) ist strenger und fordert das mindestens einmalige Durchlaufen aller Selektionen und Schleifen.
- Die Bedingungsüberdeckung (C_2-Test) bezieht die Bedingungen in Schleifen und Auswahlkonstrukten mit ein und fordert eine mindestens einmalige Bewertung der beiden Wahrheitswerte der elementaren Glieder dieser Ausdrücke.
- Die Mehrfachbedingungsüberdeckung (C_3-Test) fordert die Überprüfung aller möglichen Permutationen der Wahrheitswerte der elementaren Bedingungen; dies führt aber zu einem exponentiellen Wachstum der erforderlichen Testfälle.
- Die minimale Mehrfachbedingungsüberdeckung fordert daher - wie der C_2-Test - nur die mindestens einmalige Bewertung der beiden Wahrheitswerte der elementaren Glieder zusammengesetzter Ausdrücke; zusätzlich aber die Bewertung aller auftretenden zusammengesetzten Teilausdrücke. Die minimale Mehrfachbedingungsüberdeckung ist ein schwächeres Kriterium als der C_3-Test und stellt einen guten Kompromiss zwischen C_2- und C_3-Test dar [PagSix1994.]

- Die vollständige Pfadüberdeckung (C_4-Test) fordert die Analyse aller möglichen Pfade des Programms. Durch das Vorkommen von Schleifen in einem Programm wird die Anzahl der Testfälle schnell nicht mehr handhabbar. Ein etwas schwächerer Test ist der ‚Boundary Interior Test' bei dem nur diejenigen Testfälle untersucht werden, die einen Schleifenrumpf genau einmal ausführen und zusätzlich die Testfälle, die genau zu einer weiteren Schleifenwiederholung führen.

[SneWin2002] beschreiben neben diesen klassischen Überdeckungsmaßen für prozedurale Software einige weitere Überdeckungsmaße für objektorientierte Software:

- Überdeckung der angebotenen Operationen einer Klasse (Methodenüberdeckung)
- Überdeckung der durch eine Klasse aufgerufenen Operationen
- Überdeckung von Operationsparametern und exportierten Attributen einer Klasse
- Überdeckung der ausgelösten (behandelten) Ausnahmen
- Polymorphe Methodenüberdeckung
- Überdeckung der Attributänderungen

In der Praxis werden oft in erster Linie funktionsorientierte Testverfahren verwendet. Bei der Testausführung wird dann gemessen, wie weit der Programmcode allein durch die Ausführung der funktionsorientierten Tests überdeckt wurde. Für die so nicht überdeckten Teile des Programmcodes werden dann noch strukturorientierte Testfälle erstellt und ausgeführt.

Nach Erstellung eines entsprechenden Testdatensatzes für ein Programm bzw. einen Programmteil bedarf es also eines Werkzeugs, das in der Lage ist, die Wirkung des Testdatensatzes auf den Ablauf des Prüflings zu verifizieren. Ein Werkzeug zur Überdeckungsanalyse instrumentiert einen Programmquelltext durch ‚Parsen' des Quellcodes und Einfügen von ‚Trace'-Code. Wird der so instrumentierte Quelltext mit den Trace-Statements ausgeführt, wird eine Ausgabedatei mit einer Liste der abgearbeiteten Statements und den ermittelten Ergebnissen von Bewertungen und/oder Berechnungen erstellt. Diese Datensammlung kann von einem geeigneten Analysewerkzeug zu einem Überdeckungsreport verarbeitet werden [Bin2000].

1.2 Überdeckungstests und Anforderungen an entsprechende Werkzeuge

In diesem Abschnitt sollen die Überdeckungsmaße im einzelnen betrachtet werden und daraus erste Anforderungen an entsprechende Messinstrumente abgeleitet werden.

1.2.1 Anweisungsüberdeckung C_0

Offensichtlich kann jede fehlerhafte Anweisung eines Programms zu einem Programmfehler führen. Eine aus dieser Beobachtung folgende Strategie des Testens schreibt eine Testmenge vor, durch die jede Anweisung mindestens einmal ausgeführt wird. Dies führt auf den Begriff der Anweisungsüberdeckung. Das zu benutzende Analyse-Werkzeug muss lediglich ausgeführte Anweisungen zählen und (bspw. anhand der Zeilennummer) markieren. Am Ende eines Testlaufes wird die Anzahl der markierten Zeilen zur Gesamtzahl der Anweisungen ins Verhältnis gesetzt. Letztlich bedeutet dies aber die Instrumentierung jeder Zeile, wodurch der Prüfling zur Laufzeit der instrumentierten Version ein deutlich verlangsamtes Verhalten zeigen kann.

1.2.2 Zweigüberdeckung C_1

Da beim Testen der Anweisungsüberdeckung bedingte Anweisungen, Selektions-Anweisungen und Schleifen nicht in allen möglichen Entscheidungszweigen durchlaufen werden, führt dies auf den Begriff der nächst stärkeren Strategie - der Zweigüberdeckung. Dabei soll die Testmenge so gewählt werden, dass alle Ausführungspfade innerhalb eines Programms mindestens einmal durchlaufen werden. Das zu benutzende Werkzeug muss in diesem Fall einen Parser enthalten, der (verzweigende) Anweisungen erkennt und mit entsprechenden Instrumentierungen an syntaktisch korrekten Stellen versehen kann.

1.2.3 Bedingungsüberdeckung C_2

Die Bedingungsüberdeckung ist eine weitere Verschärfung der Zweigüberdeckung. Zusätzlich wird nun gefordert, dass zusammengesetzte Bedingungsausdrücke in nicht weiter zerlegbare ‚atomare' Ausdrücke gespalten werden. Die Testmenge soll so gewählt werden, dass jeder dieser atomaren Ausdrücke jeweils einmal ‚wahr' und einmal ‚falsch' ausgewertet wird. In diesem Fall muss das benutzte Werkzeug neben dem Parsing des Quellprogramms auch die Zerlegung und Isolierung von booleschen Ausdrücken unterstützen.

1.2.4 Minimale Mehrfachbedingungsüberdeckung

Vor allem bei sicherheitskritischen Softwareanwendungen ist die Prüfung der Verarbeitungslogik sehr wichtig. Empirische Untersuchungen zeigen, dass in diesem Anwendungsbereich die komplizierte Verarbeitungslogik in der Regel entsprechend kompliziert aufgebaute Entscheidungen in der Software verursacht. Die minimale Mehrfachbedingungsüberdeckung fordert,

dass bei zusammengesetzten logischen Ausdrücken nicht nur alle atomaren Entscheidungen einmal zu ‚true' und einmal zu ‚false' ausgewertet werden, sondern auch jeder auftretende zusammengesetzte Teilausdruck – ebenfalls jeweils ‚true' und ‚false' - durch einen Test berührt werden [ChiMil1994][Lig2001]. Für den Instrumentierer bedeutet diese Forderung – genau wie schon bei der Bedingungsüberdeckung, dass der eingesetzte Parsers solche zusammengesetzten Ausdrücke in elementare Teilausdrücke zerlegen zu kann. Diese Zerlegung muss jedoch so erfolgen, dass auch die beschriebenen Teilausdrücke isoliert und ausgewertet werden können. Bei der Auswertung dieser Ausdrücke zur Laufzeit muss darauf geachtet werden, dass der Zustand des Prüflings durch die Auswertungen nicht verändert wird.

1.2.5 Objektorientierte Überdeckungsmessungen

Speziell bei objektorientierten Sprachen, die das Vererbungsprinzip implementieren, kommt objektorientierten Überdeckungsmaßen eine gegenüber den klassischen Überdeckungsmetriken zunehmende Bedeutung zu. Dies ist insbesondere dann wichtig, wenn Methoden einer Oberklasse von Methodenimplementationen in den davon abgeleiteten Unterklassen (teilweise) überschrieben werden (Polymorphie). Werden nur für die Unterklassen Überdeckungstests durchgeführt, dann werden die beim Test benutzten Funktionalitäten der Oberklasse nicht mit in die Überdeckungsanalyse einbezogen.

Instanzvariablen, Parameter und Rückgabewert einer Operation sowie die Pseudovariable ‚this' in Java sind Referenzen auf Objekte. Dabei können diese Objekte nicht nur die angegebenen Typen referenzieren sondern auch alle Untertypen (=abgeleitete Objekte). Berücksichtigt man außerdem die durch Generalisierungsbeziehungen induzierte ‚Zerfaserung' der Klassendefinitionen, wird die Aussagefähigkeit statischer Überdeckungsanalysen erheblich eingeschränkt [WinSix2001].

Allgemein sollte ein geeignetes Werkzeug zur Überdeckungsanalyse für objektorientierte Sprachen mindestens in der Lage sein, die folgenden Kriterien zu beurteilen:

- Ermittlung der Statementüberdeckung
- Ermittlung der Methodenüberdeckung
- Ermittlung der Anzahl behandelter Ausnahmen
- Ermittlung der Objektkontexte polymorpher Methoden

Diese Forderungen ergeben sich aus der Diskussion von Anforderungen an Überdeckungsmodelle in [Bin2000]. Die letzte Anforderung bedingt auch eine etwas andere

Instrumentierungsstrategie als bei prozeduralen Sprachen, um einen maximalen Informationsgewinn beim Ablauf eines instrumentierten Tests zu erzielen, denn:

„*Unless superclasses of the class under test have been instrumented, coverage analysis tells us little, if anything, about which inherited features have or have not been exercised.*“ [Bin2000]

1.3 Zielsetzung und Überblick

Zielsetzung der Arbeit ist, den Einsatz struktureller Testverfahren für in Java geschriebene Programme zu unterstützen. Hierzu sind zunächst diesbezügliche Eigenschaften der Programmiersprache Java zu ermitteln, insbesondere jene, die einerseits die Ermittlung entsprechender Messwerte erschweren oder andererseits erleichtern. Darauf aufbauend wird ein Überblick über das Angebot und den Funktionsumfang von kommerziellen und als Forschungsprototypen erhältlichen Werkzeugen zur Überdeckungsmessung von Java-Programmen gegeben. Aufgrund der sich schon im Vorfeld der Arbeit abzeichnenden Nicht-Verfügbarkeit entsprechender Werkzeuge, wird der zentrale Punkt der Arbeit die Konzeption und Realisation eines ‚maßgeschneiderten' Testwerkzeuges zur Überdeckungsanalyse sein - entweder durch Anpassung existierender Werkzeuge oder durch komplette Neuentwicklung.

Das zu entwickelnde Testwerkzeug soll dabei zunächst die Standard-Überdeckungsmetriken (C_0, C_1, C_2) durch geeignete Instrumentierung eines Quelltextes ermitteln können. Vor allem soll es sich um ein datenbankgestütztes Werkzeug handeln, dass seine Instrumentierungs- und Überdeckungsmesswerte permanent und jederzeit recherchierbar in einer relationalen Datenbank hält. Zusätzlich zu den genannten Standardmetriken sollen auch mehr an objektorientierten Sprachen ausgerichtete Überdeckungs-Metriken wie Klassen- Methoden- und Ausnahmeüberdeckung messbar sein. Darüber hinaus wünschenswert, aber schwierig zu realisieren, ist die Messung der minimalen Mehrfachbedingungsüberdeckung. Zusätzlich zur ‚normalen', klassenorientierten Methodenüberdeckung sollte es möglich sein, zu jedem Aufruf einer Methode den jeweiligen Objektkontext zu dokumentieren und so ein Maß für die polymorphe Methodenüberdeckung zu realisieren.

Hier nun der Überblick über die weiteren Kapitel:

In Kapitel 2 gehe ich auf spezielle Anforderungen beim Testen von Java Programmen ein. Insbesondere werden diejenigen Eigenschaften der Programmiersprache Java näher erläutert, die bei der Instrumentierung und Durchführung von Überdeckungsanalysen besonders hilfreich, besonders problematisch oder besonders zu berücksichtigen sind.

Kapitel 3 gibt einen Überblick über die kommerziell und im Opensource-Bereich verfügbaren Werkzeuge zur Überdeckungsanalyse von Java-Programmen. Es wird jeweils eine kurze Zusammenfassung des Funktionsumfanges und der Besonderheiten gegeben. Für die kommerziell vertriebenen Produkte wurde jeweils der Anschaffungspreis ermittelt.

Kapitel 4 identifiziert und beschreibt die ‚Use-Cases' für das zu entwickelnde Werkzeug.

Kapitel 5 beschreibt den Grobentwurf des Instrumentierers/Evaluierers ‚DoiT' der im Rahmen dieser Arbeit entwickelt wird. Es wird ein Überblick über die Architektur von ‚DoiT' gegeben, der verwendete Parser Barat und dessen Visitor-Konzept beschrieben und eine grobe Beschreibung der Datenbankschnittstelle von ‚DoiT' gegeben.

Kapitel 6 gibt einen kurzen Überblick über die Struktur von Barats Abstract-Syntax-Trees und befasst sich mit dem Fein-Entwurf von ‚DoiT' und der zugehörigen Datenbank. Ein weiterer Schwerpunkt ist die Beschreibung der Implementationsdetails der ‚DoiT' Datenbankschnittstelle und der ‚DoiT' Barat-Visitoren. Dabei werden Lösungsvorschläge für die im Kapitel 2 identifizierten Probleme diskutiert und der Einbau dieser Lösungen in ‚DoiT' beschrieben.

Kapitel 7 gibt einige Beispiele für die Anwendung von ‚DoiT' anhand von Testfällen des Unit-Testers ‚JUnit'.

Im Anhang finden sich neben dem Datenlexikon der ‚DoiT'-Datenbank, Hinweise zur Installation und zur Weiterentwicklung des Programms sowie das Benutzerhandbuch zu ‚DoiT'.

2 Spezielle Anforderungen beim Überdeckungstesten von Java Programmen

In diesem Abschnitt sollen im Hinblick auf das Instrumentieren von Java-Quellcodes einige positive und negative Eigenschaften der Sprache Java dargestellt werden.

2.1 Problematische Spracheigenschaften

Die Sprache Java zeichnet sich als Verwandte der Sprache C durch eine hohe syntaktische Flexibilität bzw. durch große stilistische Freiheiten aus. Fast genau wie in C lassen sich relativ tief geschachtelte und kryptische Anweisungen aufschreiben, die nur dadurch etwas verständlicher werden, dass auf Zeiger und Zeigerarithmetik beim Entwurf von Java verzichtet wurde. Im folgenden sollen insbesondere solche Spracheigenschaften aufgezeigt werden, die den Entwurf eines Instrumentierers erschweren.

2.1.1 Variablendefinitionen

In Java können nahezu überall Variablen definiert werden, die abhängig vom Definitionsort einen beschränkten Gültigkeitsbereich haben. Dieser Gültigkeitsbereich entspricht meist dem umschließenden Block. Auf die Spitze getrieben wird diese Definitionsfreiheit z.B. bei **for**-Schleifen, wo die Laufvariable und nach Wunsch auch einige andere Variablen erst im Steuerblock des **for**-Statements definiert sein können. Eine Instrumentierung die vor dem Schleifenkopf eine Bedingung im Steuerblock dokumentieren möchte, greift sozusagen ins Leere, da an dieser Stelle die Variable noch gar nicht definiert ist.

2.1.2 Implizite Ausführung von Funktionen und Operationen

Ähnlich wie in C können in Java innerhalb von logischen Vergleichen Zuweisungen vorgenommen werden oder - den Wert von Variablen verändernde - Funktionen bzw. Methoden aufgerufen werden. Beispiel aus einem File-Filter:

```
ArrayList exts;
java.io.File f;
Iterator it = exts.iterator();
while (it.hasNext()){
      if (f.getName().endsWith((String)it.next()))
      return true;
}
```

Abbildung 1: Beispiel für implizite Funktionsaufrufe

Durch die Liste ‚exts', die genau wie das File ‚f' extern gefüllt worden ist, wird offensichtlich ein Iterator geschoben. Der Listenzeiger wird jedes Mal, wenn das Suffix des Dateinamens mit dem Listenelement verglichen wird, um eine Position weitergeschoben.

Häufig in Datei-Leseroutinen anzutreffen sind auch Schleifensteuerungen wie z.B.

```
while ( (line = inBuf.readLine() ) != null )
```

Abbildung 2: Beispiel für Zuweisung in einem Bedingungstest

Wenn nun ein Instrumentierer versucht, den Wahrheitswert der Bedingung durch nochmalige Auswertung der Bedingung zur Laufzeit zu dokumentieren, führt dies zu einem Seiteneffekt. Bei beiden Beispielen wird sofort klar, dass Bedingungen nur jeweils einmal ausgewertet werden dürfen, wenn die Semantik des Programms nicht durch die Instrumentierung verändert werden soll.

Ähnliche Effekte erzielt man mit den beliebten unären Operatoren wie ‚++' oder ‚--', die ebenfalls gern im Zusammenhang mit Schleifensteuerungen verwendet werden.

Eine andere, in diesem Zusammenhang unangenehme, Eigenschaft des Java Laufzeitsystems ist die Auswertung logischer Ausdrücke. Man betrachte beispielsweise den Ausdruck:

```
if (expected != null && expected.equals(actual))
```

Abbildung 3: Zur Short-Circuit-Evaluation

Es handelt sich dabei um eine Beispielzeile aus dem Unit-Test-Rahmenwerk ‚JUnit' [IntJun]. Falls ‚expected == null' wahr ist, führt diese Zeile nicht zu einer NullPointerException, da das Laufzeitsystem die Auswertung des Ausdrucks nach (expected != null) beendet. Für die JVM steht dann bereits fest, dass der Ausdruck wegen der UND-Verknüpfung nicht mehr WAHR werden kann. Wie in in C oder Pascal spricht man bei dieser Art der Bewertung von ‚short-circuit-evaluation' oder Unvollständiger Evaluation [Lig2001]. Allgemein folgt aus dieser Beobachtung, dass es offenbar Spracheigenschaften gibt, die nicht aus der Grammatik oder der Semantikbeschreibung folgen, sondern von den Eigenschaften des Laufzeitsystems abhängen.

Im Interesse einer ‚sauberen' Programmierung sollte deshalb auf Formulierungen wie die gerade beschriebene verzichtet werden und statt dessen besser folgende Konstruktion benutzt werden:

```
if (expected != null )
      if ( expected.equals(actual))
      …
```

Abbildung 4: Workaround für Fehler, die bei der Short-Circuit-Evaluation instrumentierter Ausdrücke auftreten können

Instrumentiert man nämlich die erste Version zum Zwecke einer Analyse der Bedingungsüberdeckung, führt diese Formulierung unweigerlich zur beschriebenen ‚NullPointerException', es sei denn, man berücksichtigt bei der Zerlegung und Bewertung die Eigenschaften des Laufzeitsystems.

2.1.3 Konditionaloperator

„*Der Konditionaloperator ist ein etwas obskurer ternärer Operator, der von C geerbt wurde.*“, schreibt [Flanag2000]. Dabei handelt es sich bekanntermaßen um eine Operatorversion der **if-else** Anweisung. Fatalerweise ist der Gebrauch dieses Operators fast überall in einem Java-Programm gestattet. Aus der Sicht eines Programmierers lassen sich damit elegante Lösungen für manches Problem finden. Wenn es aber darum geht, solche Konstrukte und ihre Bedingungen zu instrumentieren, steht man vor scheinbar nicht zu lösenden Problemen. Beispiel Variablenzuweisung:

```
int max = (x > y) ? x : y;
```

oder

```
sql = "SELECT " +
      (dbProduct.equals("oracle")) ? "NVL" : "IFNULL" +
      "(MAX(Inst_nr),0) AS mn " +
      "FROM PACKAGES WHERE P_NAME = '" + pr + "'" ;
```

Abbildung 5: Beispiel für die Verwendung eines Konditionaloperators

Die Bedingungen sind aus solchen Zuweisungen nur sehr schwer zu extrahieren und noch schwerer ist es, sie an dieser Stelle seiteneffektfrei zu instrumentieren.

2.1.4 Blöcke ohne Methode

Ein wenig benutztes aber hilfreiches Java-Konstrukt sind die statischen und dynamischen Klassen- bzw. Instanz-Initialisierer. Diese haben allerdings die bei der Instrumentierung ungünstige

Eigenschaft, keine umgebende Methode und keinen Namen zu haben. Außerdem ist ihre Anzahl und ihr Ort innerhalb einer Klasse relativ freigestellt (‚*überall da, wo eine Feld- oder Methodendefinition erlaubt ist*' [Flanag2000]).

2.1.5 Methoden und Blöcke – unstrukturierter Kontrollfluss

Bei einem ereignisgesteuerten Programmablauf ist es naturgemäß schwierig das Betreten und Verlassen einer Methode durch den Kontrollfluss richtig zu dokumentieren. Durch Ereignisse aus einer graphischen Benutzungsoberfläche, durch Synchronisationsereignisse zwischen laufenden Threads oder durch Ausnahmebehandlungen kann der Kontrollfluss ‚springen' und eine Methode - oder allgemeiner einen Code-Block - an nicht vorhersehbarer Stelle verlassen. Ebenso problematisch sind mehrere **return**s in Methoden oder diverse **break**s in Schleifensteuerungen.

2.1.6 Klasseninstantiierung und Konstruktoren

Eine der wichtigsten Aufgaben eines Java-Instrumentierers ist es, die Instantiierung von Klassen zu dokumentieren. Da aber 0..n Konstruktoren einer Klasse existieren können, gestaltet sich das Instrumentieren der Instantiierung schwierig. Bei Klassen, die den Konstruktor ihrer Superklasse benutzen, muss eventuell ein Konstruktor eingefügt werden, der dann die entsprechende Instrumentierung enthält.

2.1.7 Anonyme Klassen

Anonyme Klassen haben per definitionem keine Konstruktoren und können syntaktisch auf verschiedenste Art und Weise instantiiert werden. Eine Instrumentierung einer solchen Klasse ist wegen der fehlenden allgemeingültigen Signatur problematisch.

2.1.8 Vererbung und Instrumentierung

Bei abgeleiteten Klassen, die unter Umständen nur wenige Methoden ihrer Superklasse überschreiben, ergibt sich das Problem des Instrumentierungsumfangs. Bei Programmen die zum größten Teil aus abgeleiteten Klassen bestehen, wird durch eine Instrumentierung dieser abgeleiteten Klassen nur ein kleiner Teil des letzten Endes zur Ausführung kommenden Codes instrumentiert. Diese Schwierigkeit ist natürlich kein Spezifikum der Sprache Java, sondern tritt bei allen Sprachen auf, die das Vererbungsprinzip implementieren.

2.1.9 Dynamisches Laden von Klassen

In Java ist es bekanntlich möglich, Klassen, die zur Entwicklungszeit noch nicht einmal namentlich bekannt sein müssen, dynamisch zu laden und ihre Methoden und Felder ebenso dynamisch mit Hilfe der Selbstreflektivität von Java (s. 2.2.2) aufzurufen. Eine andere Möglichkeit des dynamischen Ladens besteht darin, neue Java-Prozesse (also neue JVMs) mit der zu ladenden Klasse zu starten. In beiden Fällen ist das Überleben bzw. die Sichtbarkeit eines neben dem Prüfling existierenden Testwerkzeuges nicht automatisch garantiert.

2.2 Vereinfachende Spracheigenschaften

Viele der unter 2.1 beschriebenen Probleme haben wegen der ‚guten' Spracheigenschaften von Java elegante Lösungen. Diese Lösungen und einige weitere Spracheigenschaften, die beim Instrumentieren von Java-Code hilfreich sind, sollen in diesem Abschnitt beschrieben werden.

2.2.1 Try-Finally

Durch dieses Konstrukt kann man jeden Block so ausrüsten, dass der Kontrollfluss an einer vorherbestimmten Stelle diesen Block wieder verlassen muss. Dabei kann es sich um einen Anweisungsblock, einen Klassen- oder Instanz-Initialisierer oder einen ganzen Methodenblock handeln. Dabei ist es völlig egal, ob innerhalb des Blocks **return**-Anweisungen oder **break**s auftreten. Wenn dieser Block mit **try** { ... } **finally** { .. } geklammert ist, stellt man sicher, dass **finally** { .. } als letztes ausgeführt wird. Dort kann dann auch eine entsprechende Anweisung des Instrumentierers untergebracht werden [IntMccl].

2.2.2 Reflektivität bzw. Selbstinspektion

Durch die in Java eingebaute Klasse ‚java.lang.reflect' und auch mittels ‚java.lang.object' kann der Code einer Klasse sich selbst oder den anderer Klassen untersuchen. Auf diese Art und Weise lassen sich Klasseneigenschaften, wie z.B. der Name des Objektes, die Namen der zur Verfügung stehenden Felder oder die zur Verfügung stehenden Methoden zur Laufzeit ermitteln.

2.2.3 Initialisierer

Einer der Gründe, warum Klassen und Instanz-Initialisierer in Java eingebaut wurden, ist nach [Flanag2000] die Initialisierung von anonymen Klassen. Der Initialisierer wird jeweils bei der Instantiierung der Klasse ausgeführt. Da eine anonyme Klasse jeweils mehrere Initialisierer

haben kann, lassen sich Anweisungen zur Klassen-Instrumentierung elegant in einem Initialisierer-Block unterbringen.

2.2.4 Variablendefinition und Gültigkeitsbereich

Wie in 1.2.4 bereits verdeutlicht, benötigt man für eine seiteneffektfreie Instrumentierung von Bedingungsausdrücken Hilfsvariable. Da Variablen bei Java blockbezogen sind, können diese Hilfsvariablen an geeigneter Stelle eingefügt werden und sind dennoch in ihrer Gültigkeit auf den zu instrumentierenden Block beschränkt. Falls dies nicht möglich ist, kann zwanglos ein äußerer Block, der die Variablendefinition und den zu instrumentierenden Block umschließt, eingefügt werden, um die Hilfsvariablen nach außen ‚unsichtbar' zu machen.

3 Verfügbare Werkzeuge

In diesem Abschnitt soll ein möglichst breiter Überblick über die zur Überdeckungsanalyse von Java-Programmen verfügbaren Werkzeuge sowohl kommerzieller Art, als auch aus dem Opensource-Bereich gegeben werden. Die Anzahl der bei den Recherchen gefundenen kommerziellen Werkzeuge überwiegt dabei erstaunlicherweise die der verfügbaren Opensource-Tools.

3.1 Kommerziell verfügbare Testwerkzeuge

3.1.1 McCluskey-Tools

Die McCluskey-Tools bestehen aus je einem Programm zum Parsen und zum Instrumentieren von Java-Quelltexten. Bei beiden Werkzeugen handelt es sich um Class-Dateien, die von eigenen Rahmenprogrammen importiert bzw. benutzt werden können. Sie erlauben individuell festlegbare Instrumentierungs-Statements und sind daher äußerst flexibel einsetzbar. Wegen der frei definierbaren Instrumentierungs-Statements ist eine Datenablage in einem nahezu beliebigen relationalen Datenbanksystem leicht zu implementieren. Da es sich um ein teilweise kommerziell vertriebenes Produkt handelt, werden keine Quelltexte mitgeliefert. Die im Java Byte-Code vorliegenden Dateien sind allerdings unverschlüsselt, so dass sie mit ‚geeigneten' Werkzeugen dekompilierbar sind. Der Parser baut einen ‚Abstract Syntax Tree' auf, der mittels angebotener Methoden analysiert und auch wieder ausgegeben werden kann. Jeder Knoten des AST implementiert eine Methode, die es erlaubt, bei der Ausgabe des ASTs ‚vor' und ‚hinter' dem jeweiligen Knoten frei wählbaren Text auszugeben - also auch Instrumentierungsanweisungen. Zum Testzeitpunkt unterstützte der Parser allerdings nur die Java 1.2 Grammatik.

Fazit: Bis auf die veraltete Java-Syntax-Unterstützung ein sehr geeigneter Parser um einen datenbankgestützten Instrumentierer zu bauen. Der erste Prototyp des im Rahmen dieser Arbeit entwickelten Instrumentierers ‚DoiT' basierte auf den McCluskey-Tools.

Neuerdings sind die McCluskey-Tools zumindest als Bytecode frei verfügbar; die entsprechenden Sourcen sind laut [IntMccl] ‚erhältlich'. Die McCluskey-Tools sind damit eher dem Grenzbereich zwischen kommerziell und ‚public domain' zuzuordnen.

3.1.2 JCover von Man Machine Systems

Bei diesem Programmpaket des indischen Softwarehauses 'Man Machine Systems' handelt es sich um ein datenbankgestütztes Testwerkzeug. Wegen des kommerziellen Charakters dieses Paketes sind ebenfalls keine Quelltexte erhältlich – es sei denn im Rahmen einer entsprechenden zusätzlichen Nutzungslizenz.

Als Besonderheit misst JCover die verschiedenen Kennzahlen für Überdeckungsmetriken mit einer ‚Coverage Gathering Agent' genannten Software die zwischen instrumentierter Anwendung und dem Interpreter des Java Bytecodes, der JVM (Java Virtual Machine) liegt. Daraus resultiert, dass JCover sowohl Java-Quellcode als auch Java-Bytecode instrumentieren kann. Unter einer ansprechenden und ergonomischen Benutzungsoberfläche werden klassische Überdeckungsmetriken angeboten, um einen Quelltext zu instrumentieren: Statement-, Zweig-, Methoden- und Klassenüberdeckung. Für jede Instrumentierung wird eine relationale Datenbank im Microsoft-Access-Format bzw. Microsoft-Jet-Format angelegt, in der die Testergebnisse Aufnahme finden. Das Schema dieser Datenbank ist offengelegt und von sehr einfacher und nur teilweise normalisierter Form. JCover enthält ein Analyse Werkzeug um Messwerte von Überdeckungstests mit dem Quelltext zu hinterlegen. Die Art der Präsentation hat das Design des im Rahmen dieser Arbeit erstellten Instrumentierers ‚DoiT' und insbesondere seiner GUI beeinflusst.

Der Listenpreis von JCover beträgt, Stand November 2002, USD 895,- [IntJC].

3.1.3 JavaScope von SUN

In den 1990er Jahren wurde von SUN das Werkzeug ‚JavaScope' angeboten mit dessen Hilfe Java Quelltexte instrumentiert werden und diverse Überdeckungsmaße gemessen werden konnten. Dazu gehörten

- Methoden/Konstruktoren-Überdeckung
- Block-Statement-Überdeckung
- Zweigüberdeckung
- Ausnahmeüberdeckung
- Bedingungsüberdeckung

Leider ist dieses Programmpaket nur noch auf dem ‚second hand'-Markt erhältlich, da Sun die Unterstützung/Fortentwicklung des Produktes ‚aus strategischen Gründen' 1999 aufgegeben hat. Die letzte von SUN verkaufte Version war 1.1.5. und damit werden neuere Java Grammatiken (>= 1.2) von JavaScope nicht unterstützt.

JavaScope schreibt seine Resultate in ‚flache' Dateien des jeweiligen Host-Dateisystems. Obschon in der Terminologie des Programms von einer ‚Database' die Rede ist, handelt es sich nicht um eine ‚richtige' Datenbank-Applikation die gegen ein RDBMS eingesetzt werden kann. Jedoch ist dies nur ein relativ kleiner Makel, der bei einer stetigen Weiterentwicklung sicher längst behoben worden wäre. Deshalb lautet das Fazit: Kein anderes der am Markt verfügbaren Werkzeuge die ich bei meinen Recherchen gefunden habe, bietet ein vergleichbares Leistungsspektrum.

3.1.4 Clover von Cortex

Dieses Werkzeug wurde kürzlich (August 2002) von der Fa. Cortex auf den Markt gebracht [IntCort]. Es handelt sich im Wesentlichen um Klassen zur Instrumentierung von Java-Sourcecode. Clover arbeitet eng mit ‚Ant' zusammen und wird über Ant-Steuerdateien im XML-Format parametriert. Clover ist ursprünglich ein von Cortex für die interne Nutzung bei der Softwareerstellung entwickeltes Werkzeug. Clover erlaubt einfache Methoden-, Statement- und Zweigüberdeckungsmessungen und unterstützt den JDK 1.4 mit dem neuen Schlüsselwort ‚assert'. Eine Unterstützung von Datenbankformaten fehlt.

Eine Einzellizenz für private Nutzung kostet USD 125,-, eine Teamlizenz für mehrere Entwickler bei kommerzieller Nutzung liegt bei USD 1125,- [IntCort].

3.1.5 Borlands OptimizeIt Code Coverage

Die Fa. Borland bietet unter dem Namen ‚OptimizeIt' ein Paket aus drei Testwerkzeugen für Java-Programme, bestehend aus einem Profiler (Messen von Performance Kennzahlen), einem Thread Debugger zum Aufspüren von Synchronisationsproblemen und einem Überdeckungsanalysator.

Der OptimzeIt ‚Code Coverage' erlaubt es dem Entwickler bzw. dem Tester, während eines Testlaufes die Entwicklung von diversen Überdeckungskennzahlen mit fortschreitendem Test quasi ‚online' zu verfolgen. Gemessen werden die Klassen-, Methoden- und Statementüberdeckung. Eine Prüfung der Bedingungsüberdeckung fehlt ebenso wie die Archivierung der gemessenen Daten in einer Datenbank. Dafür bietet OptimizeIt Integration mit dem verbreiteten Entwicklungswerkzeug ‚JBuilder' von Borland, sodass Testläufe direkt aus dem JBuilder heraus gestartet werden können. Offenbar installiert OptimizeIt eine Zwischenschicht zwischen Java VM und Object-Code und arbeitet mit instrumentiertem Bytecode, denn zum Testen eines Pro-

gramms ist weder der Quelltext erforderlich (obwohl zur Visualisierung hilfreich), noch ist eine Neuübersetzung des (instrumentierten) Quelltextes nötig.

Erhältlich ist offenbar immer nur das ganze Werkzeugpaket, zum Preis von USD 1599,- [IntOpti].

3.1.6 Code Coverage von Koalog

Dieses Programm arbeitet ebenfalls direkt auf Java-Bytecode. Es kommt völlig ohne Instrumentierung und Neukompilierung des Prüflings aus und ist damit ein echter ‚in-situ' Analysator, da zwischen Überdeckungstest- und Einsatz-Version eines zu prüfenden Programms kein Unterschied besteht. Ein weiterer sich daraus ergebender Vorteil ist die nur geringe Verzögerung in der Ausführung des Prüflings durch die Überdeckungsanalyse. Der Preis dafür ist, dass als Überdeckungsmaße nur die Statement- und Klassen-/Methodenüberdeckung gemessen werden können. Das Bewerten logischer Ausdrücke und die Erfassung ausgelöster Ausnahmebehandlungen sind nicht enthalten. Ebenso fehlt Code Coverage eine Datenbankanbindung. Es setzt dafür zum Ablegen der Daten ‚Log4J' ein, ein Opensource-Produkt, das Klassen zur Erzeugung und zum Auswerten von Log-Dateien zur Verfügung stellt. Code Coverage stellt übersichtliche Reports der Überdeckungsanalyse zusammen, die in mehreren gängigen Formaten ausgegeben werden können (XML, HTML, LaTex, CSV, ASCII).

Code Coverage läuft auf allen Betriebssystemen, auf denen eine JVM zur Verfügung steht und kostet (November 2002) € 500,- [IntKoa].

3.1.7 Testers Edge GJ-Cover

Dieses Testwerkzeug von Testers Edge nutzt ausschließlich die Java Virtual Machine um Überdeckungsmesswerte zu gewinnen. Darum wird für die Überdeckungsanalyse mit GJ-Cover kein Source Code, sondern ausschließlich Byte-Code benötigt. Unterstützt werden neben Zeilen/Statement-Überdeckung, die Zweigüberdeckung und die Methoden(aufruf)überdeckung. Da alle Messungen an der JVM vorgenommen werden und der Code offenbar nur wenig durch Instrumentierungsanweisungen mit entsprechendem I/O verlangsamt wird, ist GJ-Cover sehr schnell. Zur Integration mit automatischen Testumgebungen und Buildern wie JUnit bzw. Ant existiert neben der GUI-Version auch eine Kommandozeilenversion von GJ-Cover. Die GUI ist allerdings eher spartanisch. Sie beschränkt sich im Wesentlichen auf Dialogfelder, die die nötigen Informationen für den Start der Kommandozeilen-Version von GJ-Cover vom Benutzer abfragen. Es ist auch nicht möglich, Messwerte für alle drei angebotenen Metriken in einem

Testlauf zu erhalten, denn es wird nach dem entweder-oder-Prinzip verfahren. Besonderen Wert haben die Entwickler auf die Reporting-Werkzeuge von GJ-Cover gelegt. Es lassen sich sehr ansprechende, frei konfigurierbare Reports in verschiedenen Formaten (u.a. HTML) vom Ergebnis der Testläufe erstellen. Die zur Verfügung stehenden Werkzeuge zum Instrumentieren und De-Instrumentieren arbeiten bevorzugt mit JAR-Dateien und können diese in einem Zug instrumentieren oder de-instrumentieren. Ein Werkzeug zur Instrumentierung von einzelnen Class Files existiert ebenfalls.

Die Datenhaltung erfolgt in Dateien des Dateisystems des Hostrechners - ein RDBMS Anschluss ist nicht vorgesehen. GJ-Cover unterstützt Java bis zur aktuellen Version 1.4 (‚assert') und viele gebräuchliche JVMs.

Derzeitiger (November 2002) Preis für eine Einzellizenz: USD 495,- [IntGJ].

3.1.8 Bewertung der kommerziellen Produkte

Ein zusammenfassendes Urteil müsste sicher zugunsten von JavaScope gefällt werden - wenn es dieses hervorragende Werkzeug noch zu kaufen gäbe. Im verbleibenden Feld muss bei nüchterner Bewertung des Preis-/Leistungsverhältnisses sicher das Produkt JCover von Man Machine Systems auf den ersten Platz der Wertung gesetzt werden, denn es bietet neben den Standardmetriken auch die Bewertung der Methoden- und Klassenüberdeckung an. Das Programm macht einen ausgereiften Eindruck und ist in der Ergonomie JavaScope ebenbürtig. Hinzu kommt die – wenn auch einfache - Datenbankschnittstelle und die Möglichkeit Bytecode zu instrumentieren.

Für den Nutzer, der sich selbst eine maßgeschneiderte Überdeckungstest-Umgebung bauen möchte, sind natürlich die McCluskey-Tools die erste Wahl. Hier gibt es zwar keinerlei Ergonomie in Gestalt grafischer Benutzungsoberflächen, sondern ‚nur' einige Klassendateien zum Parsen und Instrumentieren; dafür aber eine sehr hohe Flexibilität bis hin zur selbst implementierbaren Datenbankschnittstelle. Das alles wohlgemerkt ohne den Sourcecode der McCluskey-Tools zu verändern! McCluskey benutzt derzeit offenbar die Java-Werkzeuge zum Parsen und Instrumentieren, um den Absatz von ‚schwergewichtigeren' – und damit auch teureren – Tools für das Testen von C++-Programmen zu fördern. Ob es allerdings in näherer Zukunft eine Version geben wird, die auch die Java 1.4 Grammatik unterstützt, war auf Anfrage nicht zu erfahren.

3.2 Testwerkzeuge und Parser aus dem Opensource-Bereich

In diesem Abschnitt werden einige Testwerkzeuge und Parser aus dem Forschungs- bzw. Opensource-Bereich vorgestellt. Im Bereich der Parser ist der Marktüberblick naturgemäß nicht vollständig. Die vorgestellten Programme sind nur wegen Ihrer Nähe zum Instrumentieren und zur Überdeckungsanalyse ausgewählt worden. Generell fällt auf, dass neben einem relativ breiten Angebot an kommerziellen Programmen zur Überdeckungsanalyse von Java-Programmen nur eine sehr kleine Auswahl an Opensource-Programmen für diesen Zweck zur Verfügung steht. Diese Aussage muss natürlich auf das eingesetzte Recherche-Werkzeug ‚Internet-Suchmaschinen' beschränkt werden.

3.2.1 JIE von E. Tromer

Die ‚Java Instrumentation Engine' (JIE) von Eran Tromer stellt ähnlich wie die McCluskey-Tools eine offene Schnittstelle zur Verfügung, mit deren Hilfe individuell festlegbare Instrumentierungs-Statements in einen Prüfling eingefügt werden können. Die JIE kann Klassen-, Methoden- und Statementüberdeckungen messen. Dies geschieht durch geeignete Instrumentierung des Quelltextes, mit deren Hilfe Methodeneintritt und –austritt sowie Klasseneintritt und –austritt ermittelt werden können. Konfiguriert werden die Instrumentierungsläufe über XML-Dateien, in die man die einzufügenden Instrumentierungsanweisungen frei eintragen kann. Die JIE baut mit Hilfe eines Parsers einen AST auf, der mittels des Visitorkonzepts (siehe nächster Abschnitt) wieder als Quelltext ausgegeben werden kann. Überhaupt ähnelt die Parser-Architektur der JIE sehr dem im folgenden beschriebenen Parser Barat. Bei der Ausgabe des ASTs werden die instrumentierenden Anweisungen eingefügt. Hauptunterschied ist, dass in der JIE durch den Einbau der XML-Konfigurations-Schnittstelle bereits die Werkzeuge zum Instrumentieren von Sourcecode enthalten sind, während für Barat spezielle Visitor-Klassen zum Instrumentieren von AST-Knoten geschrieben werden müssen. Die JIE verfügt lediglich über ein Kommadozeilen-Interface und enthält nur beschränkte Auswertungsmöglichkeiten für instrumentierte Quelltexte bzw. deren Testläufe. Auch eine Datenbankunterstützung fehlt, könnte aber wg. der Parametrierung über einfach aufgebaute XML-Dateien leicht eingefügt werden, ohne die JIE selbst zu verändern. Da die Quelltexte der JIE zugänglich sind, wäre die nachträgliche Implementation einer Datenbankschnittstelle innerhalb des JIE-Paketes natürlich ebenfalls möglich.

Nachteil der JIE ist, dass sie offenbar nicht mehr weiterentwickelt wird. Das letzte im Internet verfügbare Release unterstützt JDKs bis Version 1.2 [IntJIE].

3.2.2 Barat von B. Bokowski und A. Spiegel

Barat ist ein Parser für Java Programme, der ursprünglich an der Freien Universität Berlin von Boris Bokowski und André Spiegel entwickelt wurde [BoSpie1998]. Barat wurde von seinen Verfassern dem Opensource-Projekt zur Verfügung gestellt und kann über [IntBarat] bezogen werden. Barat ist ein reiner Parser, der einen Abstract Syntax Tree (AST) des geparsten Programms erzeugt und das Visitor-Konzept implementiert, um diesen AST zu erzeugen, näher zu untersuchen und danach wieder auszugeben. Als Grundlage für ein Testwerkzeug ist Barat genau wie die JIE gut geeignet, da sich aus dem erzeugten AST wieder ein Quelltext ausgeben lässt, der beliebige Modifikationen enthalten kann. Weitere - und für diese Arbeit entscheidende - Vorteile von Barat, sind die Opensource-Lizenz und vor allem die Möglichkeit, die logischen Bedingungen von Schleifen und Verzweigungen beim Parsen zu isolieren, zu analysieren und zur Laufzeit des Prüflings auszuwerten. Eine weitere positive Eigenschaft von Barat ist, dass sich ASTs nicht nur in Richtung einer grammatikalischen Produktion, sondern auch rückwärtsgewandt durchsuchen lassen (bspw. ‚finde die Methode meines Anweisungsblocks' oder ‚finde die umgebende Klasse' oder ‚die enthaltende Datei'). Dies ist einer der wesentlichen Unterschiede zu den McCluskey-Tools und zur JIE. Dort ist der AST stärker gerichtet, und ‚getContainer' Funktionen für AST-Knoten sind nicht eingebaut. Barat wird nach wie vor von den Autoren gepflegt und liegt derzeit in der Version 1.6 vor, die auch die neueste Java-Grammatik (1.4) verarbeitet.

3.2.3 JUnit und Hansel

Das bekannte JUnit Test-Framework [IntJun] zum Schreiben von Testklassen wird durch Hansel um Klassen erweitert, die eine Überdeckungsanalyse zulassen. Hansel (getestet wurde die Version 0.96) unterstützt neben dem Pfadüberdeckungstest die Messung der Bedingungsüberdeckung. Das Programm gibt beim Abarbeiten des Unit-Tests aus, welche Zeilen/Bedingungen vom aktuellen Test nicht berührt werden. Eine Visualisierung erfolgt zur Zeit nicht, ebenso werden keine globalen Messwerte für die obigen Metriken ausgegeben. Vorteil von Hansel ist, dass es sich sehr gut in Unit-Tests integrieren lässt und damit gut automatisierungsfähig ist. Eine Datenbankunterstützung ist derzeit nicht vorgesehen [IntHan].

3.2.4 Bewertung der Opensource Produkte

Zusammenfassend ist über die ermittelten Opensource-Werkzeuge zu sagen, dass sie wegen des offengelegten Sourcecodes und der flexiblen Schnittstellen als Grundlage für ein Instrumentierungs-/Evaluierungswerkzeug zur Überdeckungsanalyse besser geeignet sind, als die kommer-

ziellen Produkte. Gegenüber diesen ‚out-of-the-box' Produkten haben die Opensource-Programme aber den Nachteil, dass der Anwender sich zunächst intensiver mit den Werkzeugen beschäftigen muss und eventuell für seine speziellen Anforderungen Instrumentierer und Evaluierer selbst schreiben muss.

4 Anforderungsermittlung

Aus dem Vergleich der am Markt, sowohl im Opensource, als auch im kommerziellen Bereich verfügbaren Werkzeuge zur Durchführung von Überdeckungstest ergibt sich, dass die Unterstützung relationaler Datenbanksysteme – obwohl naheliegend – bisher höchstens ansatzweise implementiert wurde. Einige Eigenschaften sind dagegen mehreren der verglichenen Werkzeuge gemein: Trennung von GUI und Batchversion, Messung von C_0, C_1 und z.T. C_2 Überdeckungsmetriken, sowie Klassen und Methodenüberdeckung. Eine Messung der Ausnahmeüberdeckung findet sich dagegen nur in einem Fall. Keines dieser Werkzeuge beherrscht allerdings alle Metriken. Am flexibelsten einsetzbar sind die Parser/Instrumentierer, die aus Quelltexten ASTs erzeugen und diese dann instrumentiert ausgeben können. Hier ist man als Anwender frei bei der Definition der zu instrumentierenden Programmeinheiten und des Ausgabe- und Dokumentationsmediums. Nachteilig ist dabei natürlich, dass der Aufwand bis zum ersten erfolgreichen instrumentierten Testlauf ungleich höher ist, als bei den monolithischen (kommerziellen) Werkzeugen. Nicht zuletzt ist beim Vergleich Opensource/kommerziell auch der finanzielle Aufwand zu berücksichtigen, wobei zumindest im betrachteten Bereich die frei verfügbaren Werkzeuge den käuflichen in Qualität und Bestandssicherheit ebenbürtig sind. Wie gesehen, wird auch die Entwicklung kommerzieller Werkzeuge manchmal ‚aus strategischen Gründen' – siehe 3.3 – eingestellt.

Von dem im Rahmen dieser Arbeit zu entwickelnden Instrumentierer/Evaluierer, werden sowohl im GUI-, als auch im Batch-Modus folgende Eigenschaften erwartet:

Der Instrumentierer arbeitet ‚in situ'. Das heißt, die instrumentierte Datei befindet sich nach der Verarbeitung an der Stelle der Originaldatei. Die Originaldatei wird über einen einstellbaren Sicherungspfad im Dateisystem des Hostsystems abgelegt.

Bei der Implementierung wird darauf geachtet, dass plattformunabhängig implementiert wird.

Folgende Metriken werden implementiert: C_0, C_1, C_2, minimale Mehrfachbedingungsüberdeckung, Klassen- Methoden- und Ausnahmeüberdeckung. Eine Erfassung des jeweiligen Objektkontextes von Aufrufen polymorpher Methoden wird ebenfalls realisiert.

Die Bedienung der GUI-Komponente orientiert sich an den Gepflogenheiten und dem Stil des jeweiligen Hostsystems.

Im folgenden werden die Anwendungsfälle für den Instrumentierer bzw. Evaluierer beschrieben.

4.1 Instrumentierung von Klassen

Bei der klassenorientierten Instrumentierung sollen entweder einzelne oder in einer Aufzählung enthaltene Klassen instrumentiert werden. Dabei gehören die instrumentierten Klassen zu ein und demselben Paket. Klassen von verschiedenen Paketen müssen in jeweils separaten Instrumentierungsläufen verarbeitet werden. In UML ausgedrückt bedeutet dies:

use case ausgewählte Klassen instrumentieren

actors Bediener

precondition

Der Quelltext der gesamten Datei(en) in der die ausgewählte(n) Klasse(n) enthalten ist (sind), entspricht syntaktisch der Java-Grammatik >= 1.2. Der Quelltext ist noch nicht instrumentiert.

mainflow

Jede Quelldatei, in der eine ausgewählte Klasse enthalten ist, wird mittels Barat-Parser zerlegt, und ein AST wird aufgebaut. Für die zu instrumentierende(n) Klasse(n) werden die Instrumentierungspunkte mit ihren Zeilen-Nummern in die Datenbank eingetragen. Der AST wird mit den Instrumentierungsanweisungen mittels Barat auf die Standard-Ausgabe des Host-Systems geschrieben. Die nicht ausgewählten Klassen einer Datei werden unverändert ausgegeben.

exceptional flow Komandozeilensyntax falsch

Sind unzureichende oder syntaktisch unrichtige Angaben beim Start von ‚barat_test' gemacht worden, wird die erlaubte Syntax als Kurzhilfe ausgegeben.

exceptional flow Klasse nicht vorhanden

Falls ein nicht vorhandener Klassenname übergeben wurde, kann die Verarbeitung nur mit dem Hinweis auf die fehlende Klasse abgebrochen werden.

exceptional flow Datei nicht vorhanden

Falls ein Steuerdatei-Name fehlerhaft übergeben wurde, wird ein Hinweis auf die fehlende Datei ausgegeben und das Programm abgebrochen.

postcondition

Die Original-Dateien in denen die ausgewählten Klassen enthalten sind, sind im Sicherungsverzeichnis abgelegt. Die ausgewählten Klassen sind instrumentiert und stehen unter ihrem ursprünglichen Pfad zur weiteren Verfügung. Die Instrumentierungspunkte sind in der ‚DoiT'-Datenbank dokumentiert.

end ausgewählte Klassen instrumentieren

4.2 Instrumentierung von Dateien

Bei der dateiorientierten Instrumentierung werden entweder einzelne Kompilierungseinheiten oder eine Aufzählung mehrerer Kompilierungseinheiten verarbeitet. Auch hier gilt, dass alle Dateien Teil des gleichen Paketes sind. Dateien verschiedener Kompilierungseinheiten müssen in getrennten Instrumentierungsläufen verarbeitet werden. In UML:

use case ausgewählte Dateien instrumentieren

actors Bediener

precondition

Die Quelldatei(en) entspricht (entsprechen) syntaktisch der Java-Grammatik >= 1.2. Die Quelldatei(en) ist (sind) noch nicht instrumentiert.

mainflow

Jede Quelldatei wird mittels Barat-Parser zerlegt, und ein AST wird aufgebaut. Für die zu instrumentierende Datei werden jeweils die Instrumentierungspunkte mit ihren Zeilen-Nummern in die Datenbank eingetragen. Der AST wird mit den Instrumentierungsanweisungen mittels Barat auf die Standard-Ausgabe des Host-Systems geschrieben.

exceptional flow Komandozeilensyntax falsch

Sind unzureichende oder syntaktisch unrichtige Angaben beim Start von ‚barat_test' gemacht worden, wird die erlaubte Syntax als Kurzhilfe ausgegeben.

exceptional flow Quelldatei nicht vorhanden.

Falls ein nicht vorhandener Quelldateiname übergeben wurde, kann die Verarbeitung nur mit dem Hinweis auf die fehlende Datei abgebrochen werden

exceptional flow Datei nicht vorhanden

Falls ein Steuerdatei-Name fehlerhaft übergeben wurde, wird ein Hinweis auf die fehlende Datei ausgegeben und das Programm abgebrochen.

postcondition

Die Original-Dateien sind ins Sicherungsverzeichnis kopiert worden. Die ausgewählten Dateien sind instrumentiert und stehen unter ihrem ursprünglichen Pfad zur weiteren Verarbeitung zur Verfügung. Die Instrumentierungspunkte in den ausgewählten Dateien sind in der ‚DoiT'-Datenbank dokumentiert.

end ausgewählte Dateien instrumentieren

4.3 Instrumentierung von Paketen

Bei der paketorientierten Instrumentierung werden ganze Pakete einschließlich Sub-Pakete instrumentiert. Entsprechende Formulierung in UML:

use case ganzes Paket instrumentieren

actors Bediener

precondition

Das Quellpaket entspricht zur Gänze syntaktisch der Java-Grammatik >= 1.2. Die einzelnen Quelldatei(en) ist (sind) noch nicht instrumentiert.

mainflow

Alle zum Paket gehörenden Dateien inklusive der Sub-Pakete werden ermittelt. Jede Quelldatei wird mittels Barat-Parser zerlegt, und ein AST wird aufgebaut. Für die zu instrumentierende Datei werden jeweils die Instrumentierungspunkte mit ihren Zeilen-Nummern in die Datenbank eingetragen. Der AST wird mit den Instrumentierungsanweisungen mittels Barat auf die Standard-Ausgabe des Host-Systems geschrieben.

exceptional flow Komandozeilensyntax falsch

Sind unzureichende oder syntaktisch unrichtige Angaben beim Start von ‚barat_test' gemacht worden, wird die erlaubte Syntax als Kurzhilfe ausgegeben.

exceptional flow Paket nicht vorhanden

Falls ein nicht vorhandener Paketname übergeben wurde, kann die Verarbeitung nur mit dem Hinweis auf das fehlende Paket bzw. den fehlerhaften Classpath abgebrochen werden.

postcondition

Die Original-Dateien des ganzen Paketes sind ins Sicherungsverzeichnis kopiert worden. Die enthaltenen Dateien sind instrumentiert und stehen unter ihrem ursprünglichen Pfad zur weiteren Verarbeitung zur Verfügung. Die Instrumentierungspunkte in den Dateien des Paketes sind in der ‚DoiT'-Datenbank dokumentiert.

end ganzes Paket instrumentieren

4.4 Durchführung eines instrumentierten Testlaufes

Dieser Use-Case liegt z.T. außerhalb von ‚DoiT', da zunächst das instrumentierte Programm rekompiliert werden muss. Beim Ablauf des instrumentierten Programms sind aber wesentliche Teile von ‚DoiT' (siehe 5.3.3) beteiligt. Die Anforderungen an diese Programmteile sollen durch einen entsprechenden Use-Case beschrieben werden.

use case Testlauf durchführen

actors Bediener

precondition

Der Quelltext der zu testenden Programmeinheiten ist gemäß 4.1-4.3 instrumentiert worden und mit den für den Prüfling erforderlichen Parametern mit ‚javac' o.ä. kompiliert worden

mainflow

Ausführung des neuübersetzten Prüflings. Öffnen der Datenbank und Einfügen von Datensätzen zur Aufnahme der Messwerte dieses Testlaufes - alternativ auch Akkumulierung von Messwerten auf einem früheren Testlauf. Die Instrumentierungsanweisungen schreiben für jeden Instrumentierungspunkt Kennwerte in die Datenbank. Dies sind: Besuchshäufigkeit, Besuchsreihenfolge, Ergebnis bewerteter (Teil-)Ausdrücke, ausgelöste Ausnahmebehandlungen, Instantiierung von Klassen, Betreten und Verlassen von Methoden, Abarbeitung von Schleifen. Beim Betreten einer nicht statischen Methode wird der Objektkontext ermittelt und für jeden Kontext die Besuchshäufigkeit separat ermittelt.

exceptional flow Absturz des instrumentierten Programms

Ausnahmebehandlung der instrumentierten Programmeinheit.

postcondition

Die Ergebnisse des instrumentierten Testlaufes sind in der ‚DoiT' Datenbank abgelegt worden. Die Ergebnisse sind nach Maßgabe des Benutzers unter einer neuen Testlauf-Nummer abgelegt oder unter der jeweils letzten Testlauf-Nummer akkumuliert worden.

end Testlauf durchführen

4.5 Evaluierung von Testläufen instrumentierter Klassen

Bei der klassenorientierten Evaluierung sollen entweder einzelne oder in einer Aufzählung enthaltene Klassen, die bereits mit mindestens einem instrumentierten Testlauf in der ‚DoiT' Datenbank abgelegt sind, evaluiert werden. Das bedeutet, der ursprüngliche Quelltext wird erneut geparst und mit den an den Instrumentierungspunkten ermittelten Werten und Besuchshäufigkeiten als syntaktisch korrekter Java-Quelltext ausgegeben. Außerdem werden Werte für die Überdeckungsmaße vor dem Beginn des Quelltextes als Kommentar ausgegeben.

Als UML Use-Case ausgedrückt bedeutet dies:

use case Klasse evaluieren

actors Bediener

precondition

Der Quelltext der ausgewählten Klasse ist im Sicherungspfad enthalten und entspricht der instrumentierten Version. Messwerte zu mindestens einem Testlauf sind in der Datenbank vorhanden.

mainflow

Die Quelldatei, in der eine ausgewählte Klasse enthalten ist, wird mittels Barat-Parser aus dem Sicherungspfad zerlegt, und ein AST wird aufgebaut. Für die zu evaluierende Klasse werden die Instrumentierungspunkte mit ihren Zeilen-Nummern aus der Datenbank ausgelesen. Der AST wird mit den Messwerten als Java-Kommentar mittels Barat auf die Standard-Ausgabe des Host-Systems geschrieben. Eventuell vorhandene, nicht ausgewählte Klassen der Datei werden unverändert ausgegeben.

exceptional flow Komandozeilensyntax falsch

Sind unzureichende oder syntaktisch unrichtige Angaben beim Start von ‚barat_mess' gemacht worden, wird die erlaubte Syntax als Kurzhilfe ausgegeben.

exceptional flow Klasse nicht vorhanden

Falls ein in der Datenbank nicht vorhandener Klassenname übergeben wurde, wird die Verarbeitung abgebrochen.

exceptional flow gesicherte Quell-Datei nicht vorhanden

Falls im Sicherungspfad die Quelldatei nicht existiert, wird ein Hinweis auf die fehlende Datei ausgegeben und das Programm abgebrochen.

exceptional flow gesicherte Quell-Datei entspricht nicht der instrumentierten Version

Falls die gesicherte Datei nach der Instrumentierung geändert wurde, wird die Evaluierung mit einem Hinweis auf den unplanmäßig beendeten Evaluierer beendet.

postcondition

Eine Ausgabe des jeweiligen Quelltextes, unterlegt mit den Ergebnissen des instrumentierten Testlaufes, wurde erzeugt.

end Klassen evaluieren

4.6 Evaluierung von Testläufen instrumentierter Dateien

Die dateiorientierte Evaluierung verläuft analog zur klassenorientierten. Im Unterschied zur klassenorientierten Evaluierung wird jedoch der gesamte Quelltext mit den in Kommentarzeichen gesetzten Messergebnissen versehen.

4.7 Evaluierung von Testläufen instrumentierter Pakete

Zur Evaluierung ganzer Pakete wird zunächst geprüft, welche Dateien dieses Paketes bereits mit Testläufen in der Datenbank dokumentiert sind. Diejenigen Dateien, denen Werte zugeordnet werden können, werden dann nacheinander im Sicherungspfad gesucht, geparst und mit den in Java-Kommentar-Zeichen gesetzten Messwerten auf die Standardausgabe geschrieben. In UML ausgedrückt:

use case Pakete evaluieren

actors Bediener

precondition

Zu dem Paket muss wenigstens das Ergebnis eines Testlaufes in der Datenbank dokumentiert sein, und der Bediener muss die richtigen lfd. Nr. der Instrumentierung und des Testlaufes übergeben haben. Zu jeder der enthaltenen Dateien muss der Quelltext der genannten Datei im Sicherungspfad abgelegt sein und der instrumentierten Version entsprechen.

mainflow

Jede der im Paket enthaltenen Quelldateien, die die Vorbedingung erfüllen, wird mittels Barat-Parser aus dem Sicherungspfad zerlegt, und ein AST wird aufgebaut. Für jede zu evaluierende Datei werden die Instrumentierungspunkte mit ihren Zeilen-Nummern aus der Datenbank ausgelesen. Der AST wird mit den Messwerten als Java-Kommentar mittels Barat auf die Standard-Ausgabe des Host-Systems geschrieben.

exceptional flow Komandozeilensyntax falsch

Sind unzureichende oder syntaktisch unrichtige Angaben beim Start von ‚barat_mess' gemacht worden, wird die erlaubte Syntax als Kurzhilfe ausgegeben.

exceptional flow Klasse nicht vorhanden

Falls ein in der Datenbank nicht vorhandener Paketname übergeben wurde, wird die Verarbeitung abgebrochen.

exceptional flow gesicherte Quell-Datei nicht vorhanden

Falls im Sicherungspfad eine Quelldatei nicht existiert, wird ein Hinweis auf die fehlende Datei ausgegeben und die Datei übersprungen.

exceptional flow gesicherte Quell-Datei entspricht nicht der instrumentierten Version

Falls die gesicherte Datei nach der Instrumentierung geändert wurde, wird die Evaluierung mit einem Hinweis auf den unplanmäßig beendeten Evaluierer beendet.

postcondition

Eine Ausgabe des jeweiligen Quelltextes aller enthaltenen Dateien, unterlegt mit den Ergebnissen des instrumentierten Testlaufes, wurde erzeugt.

end Pakete evaluieren

4.8 De-Instrumentieren

Nach Abschluss der Überdeckungsanalyse an einem Prüfling muss der Prozess der Instrumentierung natürlich auch wieder rückgängig gemacht werden. Dies wird dadurch erreicht, dass die zuvor in den Sicherungspfad kopierten Original-Quelldateien wieder an ihren Ursprungsort zurückkopiert werden. Die instrumentierte Version wird dabei überschrieben.

4.9 Anforderungen an die Dokumentation der Testläufe und an die Struktur der Datenbank

Die Ergebnisse aller Instrumentierungen und Testläufe stehen standardmäßig gemeinsam in einer Datenbank. Alternativ sind aber mehrere getrennte Datenbanken möglich, da ‚DoiT' die schnelle Erstellung einer neuen (leeren) Datenbank zur Aufnahme ausgewählter Tests unterstützt. Abgelegt werden dort die Namen der instrumentierten Pakete (bei Klassen die nicht zu einem bestimmten Paket gehören, wird ‚default' genommen), der Dateien, der Klassen und der Methoden. Festgehalten wird außerdem eine fortlaufende Nummer der Instrumentierung dieses Paketes, das zugehörige Datum, sowie jeweils optional eine Versionsbezeichnung und eine freie Bemerkung. Zu jedem Instrumentierungspunkt wird die Art des instrumentierten Statements und dessen Position (Zeilennummer) im Quelltext festgehalten. Bei der Durchführung eines Testlaufes gibt es zwei Möglichkeiten: Entweder es wird ein neuer Testlauf mit einer neuen Testnummer eingefügt, oder ein früherer Testlauf wird mit geänderten Parametern fortgesetzt. Im ersten Falle werden neue Datensätze für jeden Instrumentierungspunkt dieses neuen Testlaufes erzeugt. Dabei werden für die Ermittlung der Bedingungsüberdeckung und der Minimalen-Mehrfachbedingungsüberdeckung jeweils ein Eintrag für ‚wahr' und einer für ‚falsch' für jeden Bedingungsausdruck erzeugt, damit später anhand der besuchten Einträge eine Bedingungsüberdeckung quantifiziert werden kann. Neben der Besuchshäufigkeit wird auch die Besuchsreihenfolge aufgezeichnet.

An das Datenbanksystem werden keine besonderen Anforderungen gestellt. Es muss lediglich über eine SQL- bzw. JDBC-Schnittstelle verfügen. ‚DoiT' kann momentan gegen ‚MySQL' und gegen ‚Oracle' eingesetzt werden.

5 Konzeption und Grobentwurf

In diesem Kapitel wird auf der Grundlage des Parsers Barat die Architektur und der Grobentwurf für einen Java-Instrumentierer erstellt.

5.1 Architektur von Barat

Barat ist ein Werkzeug zur statischen Analyse von Java-Programmen und läuft mit dem Java JDK ab Version 1.2. Barat parst sowohl Java Quelltexte als auch Java-Class-Dateien (Byte-Code) und erstellt einen Abstract Syntax Tree (AST) des untersuchten Programms. Dabei ist den Knoten dieses Baumes jeweils eine Produktion der Java-Grammatik (JavaJJ) zugeordnet. Klassen die nur im Byte-Code vorliegen, werden allerdings nicht dekompiliert, sondern nur bis zur Ebene der Methodenköpfe analysiert. Aus diesem Grunde wurde auf die Möglichkeit des Instrumentierens von Bytecode verzichtet. Dies hätte außerdem auch bedeutet, auf Bytecodeebene Instrumentierungsanweisungen, die Updates in einer Datenbank vornehmen, einzubauen.

Barat selbst ist nicht in reinem Java implementiert, sondern benutzt „Poor Man's Genericity"; ein Konzept das von Bokowski und Dahm [BokDah1998] entwickelt wurde und parametrisierte Klassen und Typen unterstützt. Dementsprechend kann der – gemäß dem Opensource Gedanken mitgelieferte – Barat-Sourcecode auch nicht mit herkömmlichen Java-Compilern übersetzt werden. Stattdessen benötigt man zur Weiterentwicklung bzw. Neukompilierung die Werkzeuge ‚JavaCC', um aus einer geänderten Grammatik einen neuen Barat-Parser zu generieren und den Generic Java Compiler ‚GJ' [IntGJ] um die Barat-Quellen zu übersetzen. Zur einwandfreien Funktion von Barat wird außerdem die vom M. Dahm entwickelte Byte Code Engineering Library ‚bcel.jar' [IntBcel] benötigt.

Barats öffentliches Interface besteht aus den drei Paketen Barat, ‚barat.reflect' und ‚barat.collections'. Das vierte Paket ‚barat.parser' enthält die Implementation und ist normalerweise vor den Benutzern verborgen [BoSpie1998]. Für jeden der verschiedenen AST-Knoten-Typen gibt es in der Klasse ‚barat.reflect' ein öffentliches Interface und in ‚barat.parser' eine entsprechende Klasse, die dieses Interface implementiert. Der erzeugte AST kann nicht verändert werden. Deshalb sind die von den Interfaces bereitgestellten Methoden alle Nur-Lese-Methoden. Bei der Verwendung von Barat als Instrumentierer und damit als Veränderer des Quelltextes, muss diese Veränderung deshalb bei der Ausgabe des AST während eines depth-first Durchlaufes (=Durchlauf in ‚natürlicher' Reihenfolge) durch den AST durchgeführt werden. Barats Autoren garantieren für die semantische Äquivalenz zwischen diesem aus dem AST re-generierten Quellcode und dem Originalprogramm. Die

Formatierung des Quelltextes, die Reihenfolge von Deklarationen und die Qualifizierung von externen Objekten kann allerdings vom Original abweichen. Kommentare im Quelltext gehen in jedem Fall verloren [BoSpie1998].

Im Package Barat gibt es diverse Visitoren (siehe Abschnitt 5.2), die es gestatten, den AST nach verschiedenen Besuchsstrategien zu durchlaufen, sowie das Interface ‚Node' als Vaterklasse aller Knoten-Typen. Jeder Knotentyp repräsentiert dabei eine Produktion in der Java Grammatik. Außerdem ist hier der Einstiegspunkt zum ganzen Barat-System angeordnet, nämlich die Klasse ‚barat.Barat', die Methoden bereitstellt um Java Programmkonstrukte zu analysieren. Anwender-Visitoren können entweder von bereits vorhandenen Barat-Visitoren abgeleitet, oder durch Implementierung des Barat-Interfaces ‚Visitor' erstellt werden.

In der Klasse ‚barat.collections' sind Mengen-Klassen für viele AST-Knoten-Typen enthalten. Außerdem werden entsprechende Iteratoren zur Verfügung gestellt um solche Sammlungen gleicher AST-Knoten zu durchlaufen (bspw. Mengen der Felder einer Klasse, der Konstruktoren einer Klasse oder Menge der Parameter einer Methode). Im Kapitel 6.2 findet sich eine graphische Darstellung zur Veranschaulichung eines Barat AST.

5.2 Visitor Konzept

Nach einem Konzept, das 1995 von E. Gamma et.al. beschrieben worden ist und auf einen Begriff zurückgeht, den M. Linton 1993 geprägt hat [Gamma1995], wurde in Barat eine besondere Möglichkeit des Traversierens eines AST implementiert. Dabei handelt es sich um ein Call-Back-Verfahren. Jeder Knoten in dem von Barat aufgebauten AST besitzt eine ‚accept'-Methode mit deren Hilfe ein Instanzname einer ‚Visitor' genannten Klasse im Knoten hinterlegt werden kann (Abbildung 6). Diese Klasse muss entweder von einem Barat-Visitor abgeleitet sein oder das Interface ‚Visitor' implementieren. Auf diese Art und Weise kann z.B. für eine bestimmte Art einen AST zu durchlaufen, der dazu nötige Code extern hinterlegt werden. Das Konzept hat allgemein den Vorteil, dass bei der Definition einer neuen Operation die Klassen, auf denen ein Visitor arbeitet, nicht geändert werden müssen, da die gesamte neue Funktionalität gebündelt im Visitor liegt. Jeder Knotentyp des ASTs wiederum erwartet eine bestimmte Besuchsmethode im Visitor, die er zurückzurufen versucht. Im Falle von Barat heißen diese Methoden visit*NodeType*(this), d.h der Knoten ruft seine Visitormethode mit sich selbst als Argument auf. Beispiel für einen Knoten, der eine if-Anweisung verkörpert: visitIf(this). Die einzige Konvention, die die Visitorklasse einhalten muss, ist also das Vorhandensein der Visit-Methoden und der ‚passenden' Parameter dieser Methoden. Dies kann einfach durch das Im-

plementieren des Visitor-Interfaces gewährleistet werden oder - noch besser - durch Ableitung vom ‚DefaultVisitor', der jeweils jede Methode des Interfaces leer implementiert. Die benötigten Visitoren können - wie beschrieben - leicht von bereits vorhandenen Barat-Visitoren abgeleitet werden, wobei einfach die Visit-Methoden überschrieben bzw. ausprogrammiert werden, die näher untersucht bzw. instrumentiert werden sollen. Die ‚DoiT'-Visitoren sind beispielsweise vom Barat ‚OutputVisitor' abgeleitet.

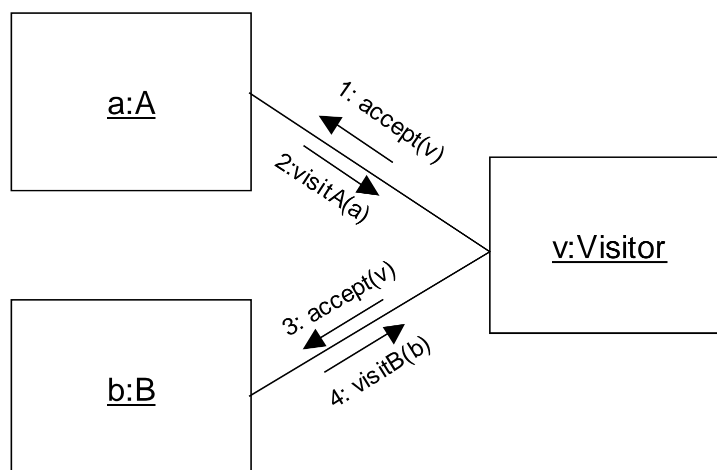

Abbildung 6: Beispielhaftes Kollaborationsdiagramm zum Visitor-Konzept (nach [BoSpie1998])

Zur weiteren Verdeutlichung des Visitor-Konzeptes sei hier ebenfalls in Anlehnung an [BoSpie1998] ein kurzes Java-Beispiel für einen Visitor gegeben, der die Häufigkeit des Auftretens von ‚System.out' Aufrufen bzw. Referenzen in einem Java-Programm zählt:

```
	public class SystemOutVisitor extends barat.DescendingVisitor {
	public int result = 0;
	public void visitStaticFieldAccess (StaticFieldAccess o) {
		Field f = o.getField();
		if (f.qualifiedName().equals ("java.lang.System.out"))
			result++;
		super.visitStaticFieldAccess (o);
	}
}
```

Um diesen Visitor auf die Klasse ‚example.MyClasss' anzuwenden, muss folgender Code ausgeführt werden:

```
barat.reflect.Class c = barat.Barat.getClass ("example.MyClass");
MyVisitor v = new SystemOutVisitor();
c.accept (v);
System.out.println  ("System.out  wurde  von  example.MyClass:  "  +
v.result +
" referenziert.");
```

Abbildung 7: Beispiel für einen Barat Visitor

5.3 Architektur von ‚DoiT'

Wie bereits einige Male erwähnt, hat das im Rahmen dieser Arbeit entwickelte Testwerkzeug den Namen ‚DoiT' erhalten. Dieses Akronym steht für „Datenbank zur **Do**kumentation von **i**nstrumentierten **T**estläufen von Java-Programmen".

Die folgende Grafik erläutert die Grob-Architektur von ‚DoiT' und die Anbindung des Datenbanksystems:

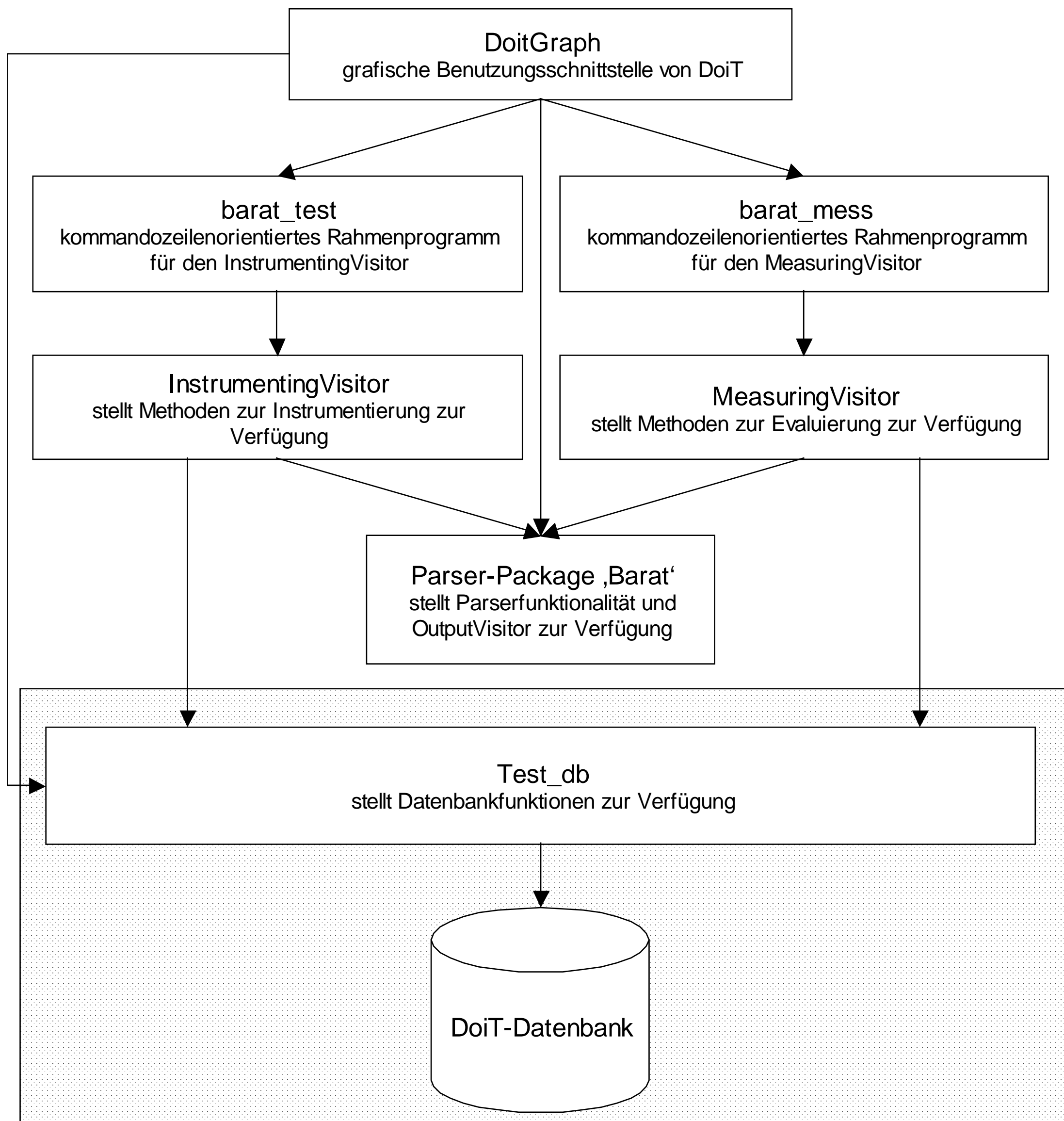

Abbildung 8: Grobarchitektur von ‚DoiT'

Die Besonderheit von ‚DoiT' liegt neben der Vielzahl der messbaren Überdeckungsmetriken in der Art der Ablage der ermittelten Programmstrukturen, der zur Laufzeit ermittelten Überdeckungs-Kennzahlen und der Ergebnisse der Bedingungsbewertungen. Herkömmliche Werkzeuge (siehe Kapitel 3) benutzen mit einer Ausnahme ‚flache' Dateien des jeweiligen Hostsystems, um Überdeckungsanalysen zu dokumentieren. Bei einer größeren Anzahl von Testläufen und von zu testenden Projekten erfordert die Verwaltung und die Dokumentation der Überdeckungstest-Ergebnisdateien schon etwas Überblick. Eine wesentliche Vereinfachung der Dokumentation ist durch den Einsatz eines Datenbankmanagementsystems möglich. Dabei können alle Testläufe eines Projektes oder eines Moduls frei von Redundanzen in einer Datenbank

dokumentiert werden. Die Datenbankschnittstelle ist dabei so konzipiert, dass eine Erweiterung von ‚DoiT' auf andere Datenbankmanagementsysteme leicht möglich ist.

5.3.1 Die grafische Benutzungsschnittstelle ‚DoitGraph'

Von der graphischen Benutzungsoberfläche ‚DoitGraph' können die beiden ‚DoiT' Kernprogramme ‚barat_test' und ‚barat_mess' parametriert und gestartet werden. ‚DoitGraph' dient dabei ‚nur' als Aufrufschale und erlaubt ein benutzerfreundliches Parametrieren der beiden Kernprogramme. Durch eine direkte Benutzungsbeziehung zur Klasse ‚Test_db', welche die Datenbank kapselt, kann ‚DoitGraph' direkt Datenbankinhalte in einer JTree-Struktur hierarchisch darstellen. Außerdem können diverse Datenbankverwaltungsfunktionen (Tabellen anlegen oder löschen, Inhalte im- oder exportieren, Inhalte löschen, JDBC-Verbindung des Datenbanksystems parametrieren) ausgeführt werden.

‚DoitGraph' stützt sich außerdem auf einige Klassen, die GUI- und Datenbankadministrations-Funktionen zur Verfügung stellen ab. Diese unmittelbar zur GUI gehörenden Klassen wurden zusammen mit ‚DoitGraph' im Paket ‚doitgui' zusammengefasst.

5.3.2 Die Klassen ‚barat_test' und ‚barat_mess'

Die Programme ‚barat_test' und ‚barat_mess' können auf Grund dieser Architektur auch aus Batch-Umgebungen heraus – ohne GUI – betrieben werden. ‚Barat_test' und ‚barat_mess' stellen ihrerseits einen parametrierenden und aufrufenden Rahmen für die darunterliegenden Visitoren ‚InstrumentingVisitor' und ‚MeasuringVisitor' dar. Diese Visitoren sind vom Barat-OutputVisitor abgeleitete Klassen, die einen Barat-AST – und damit den kompletten Sourcecode des Prüflings - vollständig durchlaufen und dabei die gewünschten Instrumentierungen/Evaluierungen des Quelltextes des Prüflings vornehmen.

5.3.3 Die Datenbankschnittstelle ‚Test_db'

Die Visitoren stützen sich ihrerseits auf die Klasse ‚Test_db' ab, die den Zugriff auf die ‚DoiT' Datenbank während der Instrumentierung und Evaluierung bereitstellt und die Zugriffsdetails (also die Datenstruktur und insbesondere die Abfragesprache) vor dem Rest der Anwendung verbirgt. Der in der Grafik mit einem gerasterten Hintergrund versehene Bereich ist derjenige, der zur Laufzeit des Prüflings zur Verfügung stehen muss. Beim Design von ‚DoiT' wurde darauf geachtet, dass zur Laufzeit des Prüflings möglichst wenig ‚Overhead' nötig ist. So kann während der Ausführung des instrumentierten Prüflings gänzlich auf den stark bremsenden Par-

ser verzichtet werden. Zudem werden fast alle Inserts in die Datenbanktabellen schon beim Instrumentieren erledigt, so dass zur Laufzeit nur noch Updates in - im Wesentlichen - eine Datenbank-Tabelle geschrieben werden.

Die einzige Ausnahme von dieser Regel stellt die Tabelle mit den Objektkontexten der Methodenaufrufe dar, denn die Kontexte können erst zur Laufzeit ermittelt werden und daher können auch erst dann die entsprechenden Datensätze erzeugt werden. Da die Visitoren nebst ihren Rahmenprogrammen und der Datenbankschnittstelle ‚Test_db' unabhängig von ‚DoitGraph' verwendet werden können, wurden diese im Package ‚doit' zusammengefasst, so dass die Applikation ‚DoiT' insgesamt aus den beiden Packages ‚doit' und ‚doitgui' besteht.

6 Feinentwurf und Implementation

In diesem Kapitel werden die Feinheiten des Entwurfes und der Implementationsdetails beschrieben. Insbesondere die Datenbankschnittstelle und der Einbau des Barat-AST sollen dabei im Vordergrund stehen.

6.1 Datenbankschema zur Ablage der Messergebnisse

Das Datenbank-Schema modelliert in erster Linie die statische Struktur von Java Programmen, also der 1..n Abhängigkeit entsprechend PACKAGES → FILES → CLASSES → METHODS → BLOCKS. Dementsprechend gibt es Entitäten für Pakete (PACKAGES), Dateien (FILES), Klassen (CLASSES), Methoden (METHODS) und Statements/Blocks (INSTR). Die verschiedenen Testläufe eines oder mehrerer Programme werden mit Hilfe der Entität ‚TESTRUNS' dokumentiert. In der Entität ‚TEST_DATA' schließlich werden die Werte, die während des Testlaufes eines Prüflings ermittelt wurden, abgelegt. Außerdem gibt es noch eine Tabelle ‚ATOM_EXPR' in der die einzelnen Teilausdrücke von zusammengesetzten logischen Ausdrücken abgelegt werden können, falls eine Ermittlung der Bedingungsüberdeckung gewünscht ist. Um zur Laufzeit eines Prüflings nur noch Updates in der Datenbank durchzuführen und die Datenbankschnittstelle ‚schlank' zu halten, wird beim Instrumentieren bereits für jeden Instrumentierungspunkt ein Datensatz in ‚INSTR' bzw. ‚ATOM_EXPR' erzeugt.

Beim Start des Prüflings wird dann zunächst für jeden Instrumentierungspunkt ein Datensatz in ‚TEST_DATA' erzeugt, sodass zur Laufzeit des Prüflings nur noch Updates durchgeführt werden müssen. Beim Einfügen der Datensätze wird von der ‚DoiT'-Datenbank dafür gesorgt, dass durch Aufbau eines Index die genannte Tabelle bezüglich ihres zusammengesetzten Schlüssels invertiert ist. Die Updates zur Laufzeit erfolgen dann entlang des beim Start erzeugten Index mit entsprechend ‚günstigem' Laufzeitverhalten. Nach der MySQL-Dokumentation sind die Indizes B-Tree-artig aufgebaut, sodass ein Zugriff auf einen bestimmten Datensatz in $O(\log_{(m+1)}n)$ Zeit möglich ist (für einen B-Baum der Ordnung m mit n Schlüsseln) [IntMySQL], [Güt1992].

Dokumentiert werden insbesondere die Häufigkeit der Besuche für einen gegebenen Instrumentierungspunkt, die Reihenfolge der besuchten Instrumentierungspunkte und die eventuell ausgewerteten Ausdrücke (Bedingungen, Schleifenkontrollausdrücke und Case Selektoren). Die Sequenznummer der Besuchsreihenfolge wird wie folgt gezählt: Für die Reihenfolge der besuchten Blöcke wird eine ordinale Sequenznummer mitgeführt, die jeweils beim ersten Besuch des Blocks geschrieben wird. Zum Beispiel wird bei einem Schleifendurchlauf nur der Wert der

Sequenz-Nummer am Beginn der Schleife vermerkt. Der Wert der Sequenz Nr. am Ende der Schleife beträgt Sequenz Nr. + Anzahl der Besuche. Mit diesem Wert wird die Sequenznummer des nächsten, nach dem Schleifenende besuchten, Instrumentierungspunktes gesetzt.

Einzige Ausnahme von dieser Regel ist die Erfassung des Objektkontextes für nicht statische Methoden. Zum Zeitpunkt der Instrumentierung ist noch nicht klar, in wie viel verschiedenen Kontexten und wie häufig eine Methode aufgerufen wird. Daher können auch nicht im Voraus Datensätze für diese verschiedenen Aufruf-Kontexte reserviert werden. Deshalb wird jeweils erst zur Laufzeit beim ‚Betreten' einer Methode die Tabelle ‚KONTEXT' mit den erforderlichen Angaben (aktueller Objekt-Kontext, Besuchshäufigkeit, Sequenznummer, etc.) beschrieben. Dabei wird geprüft ob es schon einen Eintrag für die aktuelle Methode in diesem Kontext gibt. Falls ja, wird nur ein Update vorgenommen; falls nein, wird eine neuer Datensatz erzeugt.

Das Instrumentieren verläuft aus Sicht der Datenbank immer paketorientiert. Das heißt, jeder neue Instrumentierungslauf in einem bestimmten Paket wird von eins beginnend fortlaufend nummeriert. Daraus folgt, dass im Extremfall die instrumentierten Testläufe ganzer Pakete (wobei ev. Sub-Pakete mit eigener ‚Instr_nr' gezählt werden) unter einer bestimmten Instrumentierungs-Nummer dokumentiert werden oder dass es sich lediglich um eine einzige Klasse handelt.

Das Datenbankschema wurde mit Hilfe der Modellierungssoftware ‚ErWin' von Computer Associates entwickelt und besitzt folgendes ER-Diagramm:

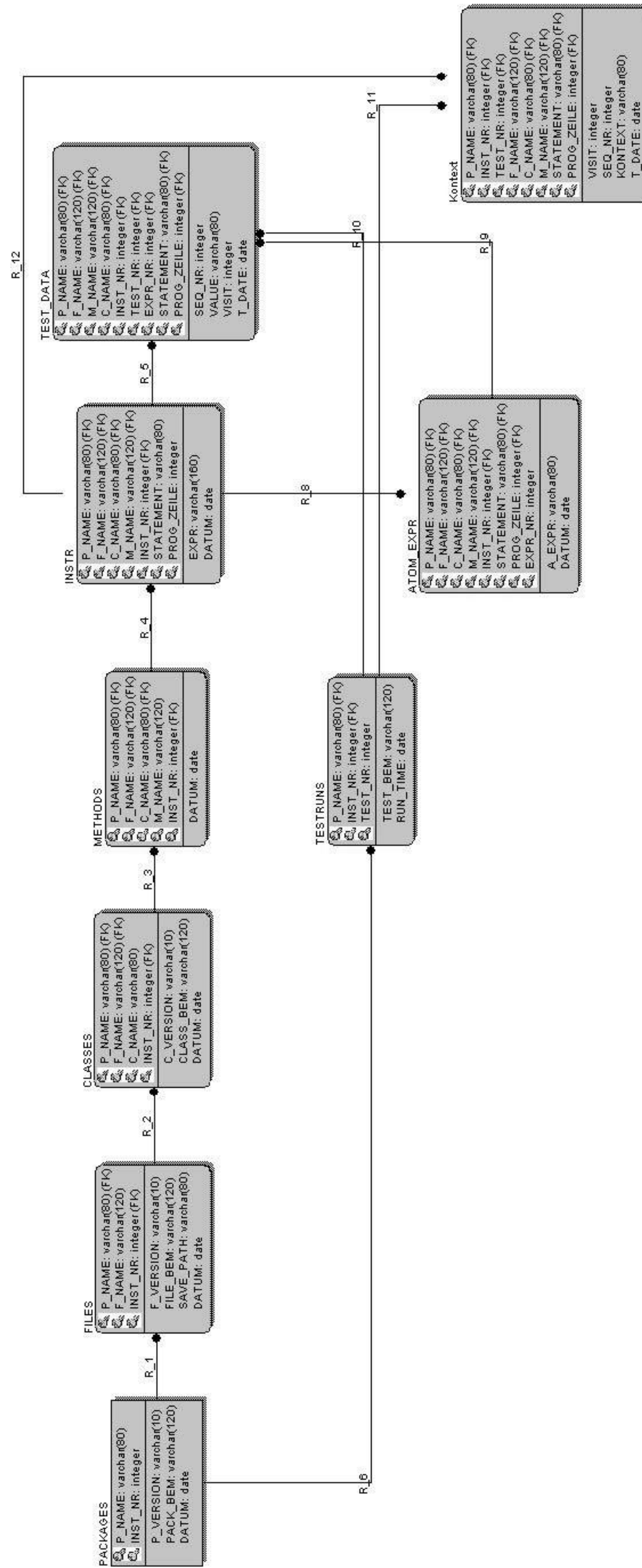

Abbildung 9: E/R Diagramm der ‚DoiT' Datenbank

Wichtigste Tabellen der Datenbank sind zweifellos ‚TEST_DATA', ‚ATOM_EXPR' und ‚INSTR'. Letztere enthält die Angaben zu Lage und Inhalt der instrumentierten Statements in den Prüflingen. Außerdem werden Datum und Uhrzeit der Instrumentierung und der Klartext eventueller zu bewertender Ausdrücke hier abgelegt. In der Tabelle ‚ATOM_EXPR' werden diejenigen logischen Teilausdrücke abgelegt, die durch Zerlegung der Ausdrücke in ‚INSTR' ermittelt worden sind. Diese Teilausdrücke sind für jedes der in Frage kommenden Bedingungen von 0..n im Attribut ‚expr_nr' nummeriert. Dabei werden die Ausdrücke, die Teil einer Zerlegung im Sinne der ‚minimalen Mehrfach-Bedingungsüberdeckung' sind, mit negativen ‚expr_nr' gekennzeichnet. Dies hat den Vorteil, dass sofort erkennbar ist, welche Ausdrücke zu welcher Metrik (Bedingungsüberdeckung oder minimale Mehrfachbedingungsüberdeckung) gehören, und dass über die SQL Absolut-Funktion/bzw. Größer-0-Test sofort alle Teilausdrücke für die eine oder die andere Metrik zur Verfügung stehen.

Die Tabelle ‚TEST_DATA' steht zu den beiden vorgenannten in einer 1..n Beziehung. Für jeden Testlauf können in der Tabelle ‚TEST_DATA' Einträge für jeden Instrumentierungspunkt erzeugt werden. Dabei wird für logische Ausdrücke, egal ob zusammengesetzt oder atomar, jeweils für beide möglichen Wahrheitswerte ein Datensatz erzeugt (Use-Case 4.4). So kann ermittelt werden, wie oft eine Bedingung zu ‚true' und zu ‚false' ausgewertet wurde. Auch hier findet sich das Attribut ‚expr_nr'. Bewertungsergebnisse mit ‚expr_nr == 0' gehören zu einer (zusammengesetzten) Bedingung aus ‚INSTR', solche mit ‚ABS(expr_nr) > 0' gehören zu zerlegten Teil-Ausdrücken aus ‚ATOM_EXPR'. Außerdem werden hier Datum und Uhrzeit des letzten Tests der zugehörigen Anweisung, die Besuchshäufigkeit und die Sequenz-Nummer zur Ermittlung der Besuchsreihenfolge dokumentiert.

6.2 Struktur von Barat-ASTs

Folgende Abbildung soll die Struktur eines Barat-ASTs verdeutlichen:

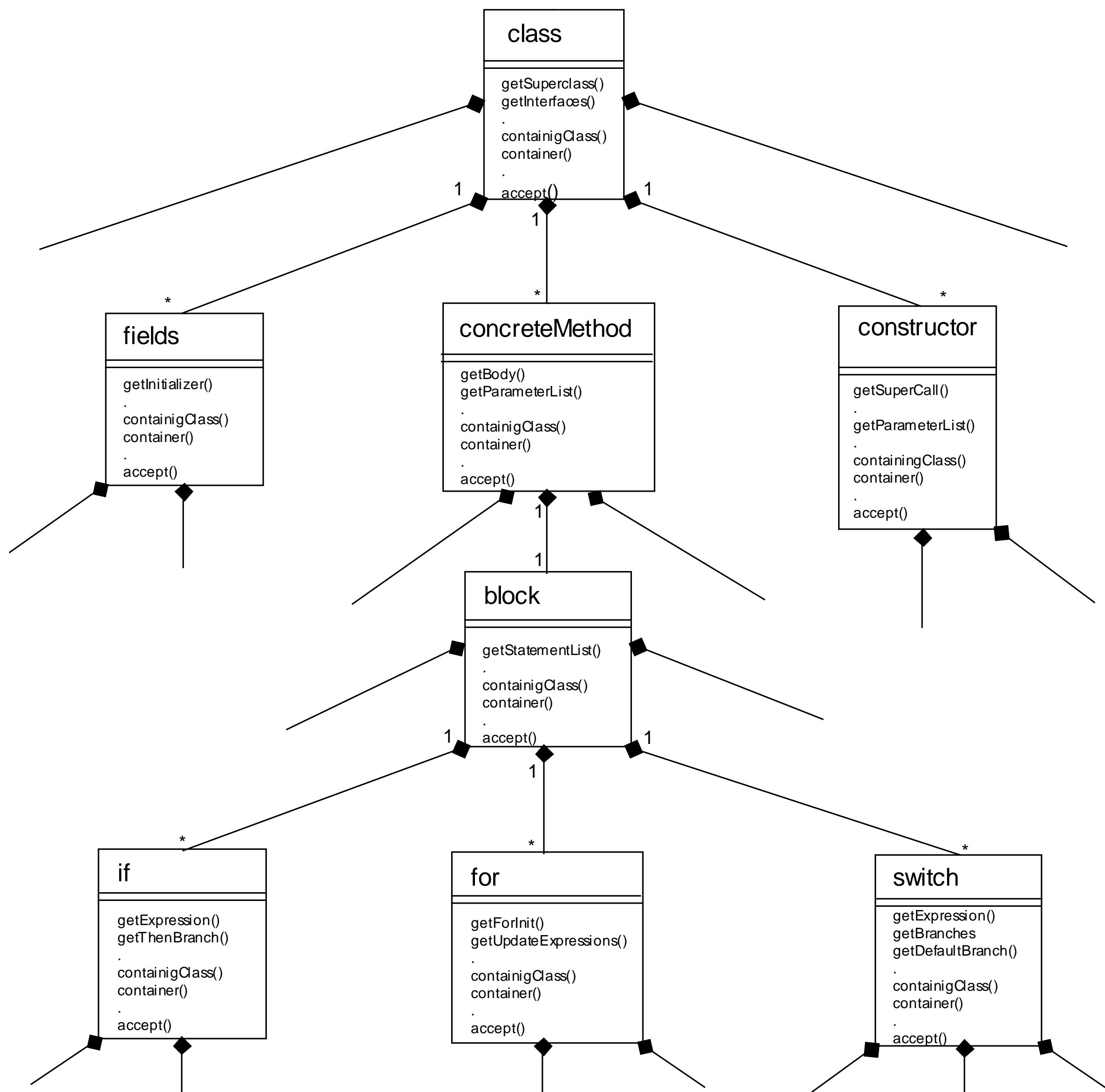

Abbildung 10: Ausschnitt aus einem Barat-AST, dargestellt als UML Klassendiagramm

Da es sich nur um einen Ausschnitt aus einem Barat-AST handelt, sollen die freien Kompositions-Kanten deutlich machen, dass der Knotengrad im AST abhängig von der Java-Grammatik variiert. Alle Knoten des Barat AST sind Ableitungen von ‚barat.Node'. Diese Klasse stellt die allen AST-Knoten gemeinsame Funktionalität zur Verfügung. Dies ist hier durch den Eintrag der Methoden ‚containingClass()', ‚container()' und ‚accept()' in die AST-Knoten angedeutet. Einige der für den Knoten spezifischen Methoden sind jeweils zu Beginn der Methodenliste genannt. Wichtigste Methode zum Traversieren des AST ist ‚accept()', mit der ein Instanzname eines Visitors an den jeweiligen Knoten übergeben werden kann. Die Baumstruktur wird von Barat nur dann komplett durchlaufen, wenn ein entsprechend konstruierender Visitor mit einem

Wurzelknoten gestartet wird (z.B. mit einer Java-Datei oder einer Klasse). Der Ausdruck ‚Traversieren' charakterisiert das Durchlaufen eines Barat-AST deshalb nur unzureichend, weil er suggeriert, dass es sich um eine gewöhnliche, durch geeignete Verbindungen aggregierte Datenstruktur handelt. Barat parst aber ‚on-demand', d.h. Java-Sourcecode wird in dem Moment weiter zerlegt, wo die entsprechende Methode eines AST-Knotens aufgerufen wird. Beispiel: ‚getConcreteMethods()'-Aufruf an einen Class-Node führt zum Parsen der entsprechenden Klasse und zur Rückgabe der entsprechenden ‚ConcreteMethod'-Nodes. Den Methoden-Knoten können dann mittels ‚accept()' wiederum Verweise auf den Visitor übergeben werden, der dann mit der Untersuchung der Methoden-Knoten (z.B. mittels ‚getBody()', ‚getParameterList()', etc. fortfahren kann. Dabei haben die Knoten keine feste Verbindung untereinander. Einzig das durch den Visitor vorgegebene Besuchsmuster bzw. Erzeugungsmuster der Knoten besitzt eine Baumstruktur, denn alle Knoten dieses Baumes werden erst auf Grund des Austausches von Botschaften zwischen AST-Knoten und Visitor erzeugt.

6.3 Instrumentierende und evaluierende Barat-Visitoren und entsprechende Rahmenprogramme

Wie bereits im Abschnitt 5 beschrieben, wurde für das Design von ‚DoiT' insgesamt folgende Entwurfsentscheidung getroffen: Instrumentierer/Evaluierer und Bedieneroberfläche sollten streng getrennt werden. Das bietet den Vorteil, dass der Instrumentierer sich vollständig über eine Steuerdatei parametrieren lässt und so auch aus unbeaufsichtigten Batch-Abläufen ohne GUI einsetzbar ist. Die Struktur der Steuerdatei ist sehr einfach und wird im Anhang im Benutzungshandbuch beschrieben.

Aufgrund der Java Syntax kommt es an einigen Stellen eines Programms zu Problemen beim Einfügen von instrumentierenden Anweisungen. Im folgenden sollen einige dieser ‚Problemzonen' dargestellt werden, soweit sie im Verlauf des Entwurfes und der Implementation von ‚DoiT' relevant geworden sind.

6.3.1 Besonderheiten und Probleme des Entwurfs

Die folgende Sammlung beschreibt einige der beim Entwerfen und Implementieren von ‚DoiT' bzw. der Barat-Visitoren gefundenen Probleme und die entsprechenden Lösungsansätze dazu:

- Instrumentierung von Methoden

Im Gegensatz zum ‚Betreten' einer Methode, steht der Ort bzw. die Anweisung, an dem der Kontrollfluss eines Programms den Code einer Methode verlässt, im Vorhinein nicht fest. Ein

Instrumentierer, der Beginn und Ende der Verarbeitung einer Methode aufzeichnen will, muss also in der Lage sein, dies entsprechend zu berücksichtigen. Hier wurde eine in den McCluskey-Tools beschriebene Lösung aufgegriffen. Jeder Methodenblock wird in einen **try-finally**-Block verpackt, und die Anweisung, die das Methoden-Ende dokumentiert, wird in den **finally**-Block eingesetzt. Damit ist sichergestellt, dass die Anweisung, die das Methodenende dokumentieren soll, auch zuletzt ausgeführt wird. Beim Betreten der Methode wird außerdem, wie unter 6.1 beschrieben, der Objekt-Kontext einer nicht statischen Methode dokumentiert bzw. die Häufigkeit des Auftretens gezählt.

- Instantiierung von Klassen

Ein Überdeckungswerkzeug, das die Instantiierungshäufigkeit von Klassen messen will, muss in der Lage sein, Instantiierungen in jedem Falle zu erkennen und richtig zu dokumentieren. Dies ist z.B. bei Klassen, die nur über einen Default Konstruktor (also gar keinen eigenen) oder über mehrere Konstruktoren verfügen, gar nicht so einfach. Noch schwieriger ist das Instantiierungsereignis einer anonymen Klasse zu erkennen, da diese per definitionem gar keinen Konstruktor besitzt und obendrein in syntaktisch sehr verschiedenen Konstruktionen instantiiert werden kann.

Hier wird zur Lösung eine der positiven Eigenschaften von Barat genutzt. Gibt es nur einen Default-Konstruktor, fügt Barat einen parameterlosen Konstruktor automatisch ein. Dieser wird von ‚DoiT' mit einer geeigneten Instrumentierungsanweisung versehen um die Instantiierung der Klasse zu dokumentieren. Im Falle der anonymen Klasse wird ein eher selten benutztes Java-Konstrukt ausgenutzt, nämlich das des dynamischen Initialisierers. Ein Initialisierer einer Klasse ist ein Block, der jeweils beim Instantiieren einer Klasse ausgeführt wird und auch I/O Anweisungen enthalten darf. Dieser eingefügte Initialisierer enthält die Instrumentierungsanweisung für die Instantiierung der anonymen Klasse.

Eine weitere Schwierigkeit ergibt sich bei instrumentierten Prüflingen, die instrumentierte Klassen dynamisch nachladen oder in einer neuen JVM-Umgebung ausführen, wie dies z.B. bei JUnit (2.1.9) der Fall ist. Deshalb wird von den Instrumentierungsanweisungen in einem Konstruktor zunächst getestet, ob die Datenbank bzw. die entsprechende Java-Klasse, die die Datenbankschnittstelle zur Verfügung stellt, noch im Sichtbarkeitsbereich liegt. Falls dies nicht der Fall ist, wird die Datenbank erneut geöffnet.

- Instrumentierung von Schleifen

Sollen Schleifen mitsamt ihrer Schleifen-Bedingungen instrumentiert werden, sodass die Schleifenbedingung jeweils ausgewertet werden kann, stellt sich z.B. bei einer **for**-Schleife das Problem, dass die Laufvariable der Schleife erst im Block der Schleifensteuerung definiert werden kann; also unmittelbar vor dem **for**-Statement noch gar nicht bekannt ist. Eine Auswertung der Schleifenbedingung vor dem ersten Schleifendurchlauf scheint dadurch nicht möglich zu sein, es sei denn, durch Isolation der Definition bzw. der Initialisierung der Laufvariablen und Veränderung des Codes, sodass die Variable bereits vor dem Schleifen-Block deklariert wird. Da ein solches Vorgehen wohl nicht mehr als Instrumentierung bezeichnet werden kann, wurde auf eine entsprechende Implementierung verzichtet. Stattdessen erfolgt die Instrumentierung ausschließlich im Gültigkeitsbereich der Laufvariable, also innerhalb des Schleifenblocks.

- Nicht geklammerte Blöcke

Beim Instrumentieren implizit geklammerter Blöcke, wie z.B. einem einzeiligen Schleifenrumpf oder einzeiligen then- oder **else**-Zweigen in **if**-Statements, führt das bloße Hinzufügen von instrumentierenden Anweisungen zu Fehlern oder semantischen Verzerrungen. Ein Instrumentierer der dies berücksichtigt, setzt also solche ungeklammerten Blöcke immer in Blockklammern. Genauso wird dies von ‚DoiT' gemacht. Da an den beschriebenen Stellen im instrumentierten Code immer mehr als eine Anweisung steht, wird generell geklammert.

- Instrumentierung von Bedingungen

Wie in 2.1.2 dargestellt, ist die Evaluierung von Bedingungs-Anweisungen, die Funktionsaufrufe enthalten, problematisch, da das Auswerten einer Bedingung zum Zwecke der Dokumentation eines Wahrheitswertes den Zustand des Prüflings ändern kann. Dazu gibt es generell zwei Lösungsansätze: Entweder man verzichtet beim Programmieren auf Bedingungsanweisungen und Schleifensteuerungen, die neben einer Zustandsabfrage auch den Zustand des Programms verändern, oder man wertet beim Instrumentieren Bedingungen konsequent nur einmal aus und fügt zum Zwecke der Dokumentation des auftretenden Wahrheitswertes für jede im Prüfling auftretende Bedingung eine Variablendefinition ein. Mit Hilfe dieser Variablen wird der Wahrheitswert der Bedingung frei von Seiteneffekten in der Testdatenbank gesichert.

Bei der Durchführung einer Zerlegung zusammengesetzter boolescher Ausdrücke (also bei der Messung der C_2- und der minimalen Mehrfachbedingungs-Überdeckung) fügt ‚DoiT' ebenfalls

für jeden auftretenden Teilausdruck eine Hilfsvariable ein. Dabei werden zwei Fälle unterschieden: Entweder werden die Instrumentierungsanweisungen mit den atomaren und den ggf. ermittelten zusammengesetzten Teilausdrücken **vor** der zu testenden Bedingung eingefügt oder **hinter** der entsprechenden Bedingung. Bei kopfgesteuerten Schleifen z.B. werden dabei die instrumentierten Bedingungen hinter der Steuerbedingung eingesetzt (siehe 2.1.1), bei fußgesteuerten und bei **if**-Anweisungen davor. Im letzten Fall führt dies zu einem Austausch des Originalausdruckes aus dem Prüfling gegen einen isomorphen und aus Hilfsvariablen zusammengesetzten Ausdruck. Hier sind sicher die Grenzen einer Instrumentierung erreicht. Dabei stellt sich die fast schon philosophische Frage, wie viele und welche Änderungen kann/darf ein Instrumentierer vornehmen, ohne dass man von einem völlig neuen Programm (u.U. mit völlig neuen Fehlern!) sprechen muss?

Es sollte aber unabhängig von diesen Überlegungen darauf geachtet werden, dass zusammengesetzte boolesche Ausdrücke keine ‚Unsauberkeiten' wie z.B. (a < i++ && b < j++) enthalten. Auch das mehrfache Auftreten des gleichen Funktionsaufrufes in einer Bedingung wird von ‚DoiT' nicht erkannt und sollte deshalb vermieden werden

Um das instrumentierte Programm vor neuen Seiteneffekten oder gar Kompilierungsfehlern wg. mehrfach auftretender Definitionen zu schützen, werden die Blöcke mit den instrumentierten Bedingungen nochmals mittels {..} zu Blöcken zusammengefasst, um den Geltungsbereich der eingefügten Hilfsvariablen einzuschränken. Die Variablen werden aus dem Präfix ‚var' und einer fortlaufenden Nr. 0..n automatisch generiert. Sollte dies in der Praxis zu Interferenzen mit Programmvariablen führen, kann man einfach die Folge der Zahlen 0..n durch eine Folge, die mit ‚System.currentTimeMillis()' (Millisekunden seit 1.1.1970) beginnt, ersetzen. Es ist dabei extrem unwahrscheinlich, dass von einem Programm gerade dieser Variablenname benutzt wird. Aus Gründen der Lesbarkeit des instrumentierten Codes wurde zunächst von der Implementierung dieser Idee abgesehen.

Zur Zeit führt ‚DoiT' immer eine ‚Vollständige Evaluation' von zusammengesetzten Ausdrücken durch (siehe 2.1.2), da noch nicht geklärt ist, wie der Instrumentierer zur Laufzeit ermitteln kann, auf welche Weise Ausdrücke ausgewertet werden; vollständig oder unvollständig. Gegen eine unvollständige Evaluierung spricht derzeit, dass die instrumentierenden Anweisungen, die die atomaren Bedingungen dokumentieren sollen, in eine Konditional-Anweisung ‚verpackt' werden müssten, sodass die verschiedenen instrumentierenden Anweisungen abhängig von bestimmten Zuständen des Prüflings ausgeführt würden oder nicht. Dies ist zur Zeit mit den gewählten Mitteln nicht für alle möglichen Fälle handhabbar.

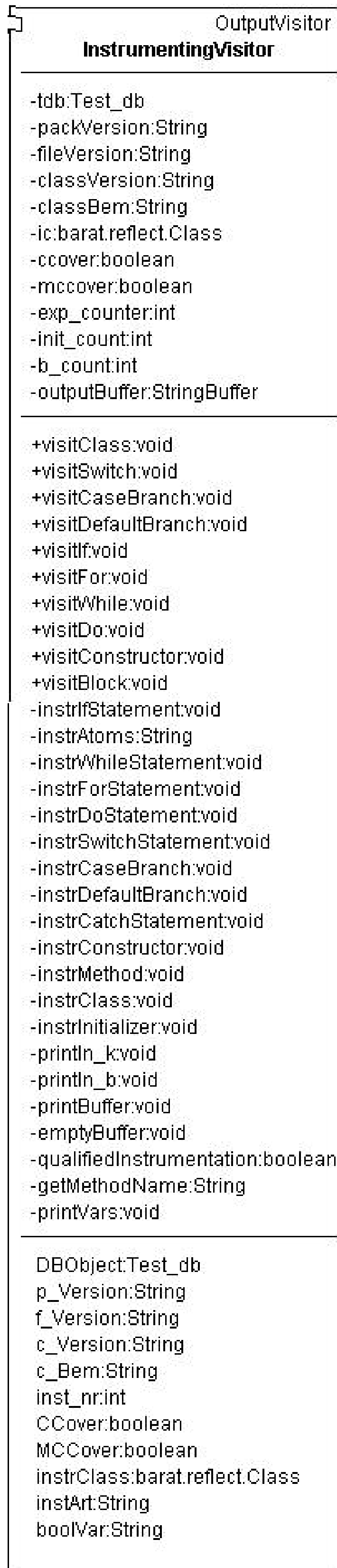

Abbildung 11:
Klasse 'InstrumentingVisitor'

Auf die Instrumentierung von Konditionaloperatoren im Prüfling und insbesondere der hier auftretenden Bedingungen, musste aus den unter 2.1.3 dargestellten Gründen verzichtet werden. Es konnte bisher kein Ansatz gefunden werden, der mit allen möglichen Ausprägungen des ‚?' Operators einwandfrei umgehen kann. Insbesondere der Konflikt zwischen dem möglichen Gültigkeitsbereich der zu instrumentierenden Bedingung bzw. der Variablen aus denen sie besteht und einem ‚erlaubten' Ort für die passende Instrumentierungsanweisung machen eine Lösung schwierig.

6.3.2 Entwurf des Instrumentierers ‚barat_test' und des ‚InstrumentingVisitors'

Der Instrumentierer ‚barat_test' stellt ein Rahmenprogramm für den ‚InstrumentingVisitor' dar, der von ‚barat_test' gestartet wird. Der ‚InstrumentingVisitor' ist eine Ableitung von ‚Barat-OutputVisitor' und überschreibt eine Reihe von Methoden ‚visitXXX', wobei XXX für einen Knoten-Typ im AST bzw. für eine Java-Struktur, die instrumentiert werden soll, steht. Die Namen der eigentlichen instrumentierenden Methoden beginnen mit dem Präfix ‚instr' und werden von den überschriebenen ‚visit' Methoden aufgerufen. Folgende Konstrukte werden instrumentiert:

Klassen (benannte und anonyme), Methoden (Betreten und Verlassen, sowie der Objekt-Kontext), Konstruktoren (Instantiierung), Schleifen (**for**, **while**, **do**-**while**), **if**-then-**else** Konstruktionen, Ausnahmen und Selektionsanweisungen.

Da alle Konstrukte instrumentiert werden, die den Kontrollfluss beeinflussen, wird auch eine Sequenznummer mitgeführt, die die Reihenfolge der ausgeführten Anweisungen/Blöcke reflektiert.

Die Implementation des ‚InstrumentingVisitors' stützt sich direkt auf Barat und auf die Klassen ‚doit.Test_db' und ‚doit.Utility' ab. Die letztgenannte Klasse enthält einige immer

wieder von den Visitoren benötigte Container-Methoden und String-Operationen. ‚InstrumentingVisitor' implementiert alle der im vorherigen Abschnitt beschriebenen Besonderheiten. Hervorzuheben ist dabei der algorithmisch anspruchsvollste Teil:

Die Zerlegung logischer Ausdrücke in atomare Elemente in der Methode ‚instrAtoms(..)'. Dabei wird intensiv von den Eigenschaften des Barat-AST-Knotens ‚binaryOperation' Gebrauch gemacht. Bei der rekursiven Methode ‚instrAtoms(..)' wird im Wesentlichen wie folgt vorgegangen:

```
Ausdruck := zusammengesetzter logischer Ausdruck;
Tiefe := 0;
Ausdrucksnummer := 0;
Präfix := wahr oder falsch;

Algorithmus instrAtoms ( Ausdruck, Tiefe, Präfix )
If Ausdruck ist ein unärer Ausdruck und Tiefe ist größer 0 Then     // Ausdruck ist nicht der
                                                                    // Ursprungsausdruck und
        Ausdrucksnummer++;                                          // ist auch nicht weiter zerlegbar
        Trage Ausdruck zusammen mit der Ausdrucksnummer in DB ein;
        Generiere neue Hilfsvariable;
        If Präfix Then
                Return "Hilfsvariable";
        Else
                Return "Hilfsvariable = (" + Ausdruck + ")";
        End If;
Else
        // Behandlung eventueller Zuweisungen an boolesche Variable innerhalb des Ausdrucks
        If Ausdruck ist Zuweisung und RHS ist boolescher Ausdruck Then
                If MMC gewünscht ist und Tiefe > 0 Then
                        Ausdrucksnummer++;
                        Trage Teilausdruck mit Ausdrucksnummer * (-1) in die DB ein;
                End If;
                If Präfix Then
                        Rückgabewert := „LValue = " + instrAtoms(RHS, Tiefe + 1, Präfix);
                Else
                        Generiere neue Hilfsvariable;
                        Rückgabewert := "HilfsVariable = ( " + „LValue = (" + instrAtoms(RHS,
                                        Tiefe + 1, Präfix ) + "))";
                End If;
        End if
        // Behandlung von Operationen mit zwei Operatoren ('binaryOperation')
        If Ausdruck ist binaryOperation Then
                If linker Operand ist ein boolescher Ausdruck Then  // Ausdruck ist also
                                                                    // zusammengesetzt
                                                                    // und möglicherweise weiter
                                                                    // zerlegbar
                        If MMC gewünscht ist und Tiefe > 0 Then
                                Ausdrucksnummer++;
                                Trage Teilausdruck mit Ausdrucksnummer * (-1) in die DB ein;
                        End If;
                        If Präfix Then
                                Rückgabewert := instrAtoms(linkerOperand, Tiefe + 1, Präfix) +
                                Operator;
                        Else
                                Generiere neue Hilfsvariable;
                                Rückgabewert := "HilfsVariable = ( " + instrAtoms(linkerOperand,
                                                Tiefe + 1, Präfix ) + ")" + Operator;
                        End If;
                Else                                                // Ausdruck ist atomar
                        Ausdrucksnummer++;
                        Trage atomaren Ausdruck zusammen mit der Ausdrucksnummer in die DB ein;
                        Generiere neue Hilfsvariable;
                        If Präfix Then
                                Rückgabewert = "Hilfsvariable";
                        Else
                                Rückgabewert = "(Hilfsvariable = (" + Ausdruck + "))";
                        End If
```

→

```
                    If rechter Operand ist ein boolescher Ausdruck Then // Ausdruck ist also
                                                                       // möglicherweise weiter zerlegbar
                              If Präfix Then
                                        Rückgabewert =  Rückgabewert +
                                                        instrAtoms(rechterOperand, Tiefe + 1, Präfix);
                              Else
                                        Generiere neue Hilfsvariable;
                                        Rückgabewert =  Rückgabewert + "HilfsVariable = ( " +
                                                        instrAtoms(rechterOperand, Tiefe + 1, Präfix ) + ")";
                              End If;
                    Else                                               // Ausdruck ist atomar
                              Ausdrucksnummer++;
                              Trage atomaren Ausdruck zusammen mit der Ausdrucksnummer in die DB ein;
                              Generiere neue Hilfsvariable;
                              If Präfix Then
                                        Rückgabewert = "Hilfsvariable";
                              Else
                                        Rückgabewert =  "(Hilfsvariable = (" + Ausdruck + "))";
                              End If;
                    End If;
          Return Rückgabewert;
End If;
End Algorithmus instrAtoms;
```

Abbildung 12: Algorithmus zum Zerlegen zusammengesetzter logischer Ausdrücke

Ausgenutzt wird hierbei, dass Barat die Vorrang-Regelungen zwischen Operatoren und allfälligen Klammerungen in zusammengesetzten Ausdrücken beachtet. Auf diese Weise kann folgende Fallunterscheidung für jeden Ausdruck getroffen werden:

1) Ausdruck ist unär und ist deshalb nicht weiter zerlegbar (z.B. eine einzelne boolesche Variable).
2) Ausdruck ist eine Zuweisung, deren rechte Seite ein boolescher Ausdruck ist. Also Ausdrücke der Gestalt (var0 = (A AND B)) und nicht etwa ((line = inBuf.readLine()) != **null**)). Letzterer ist nämlich ein boolescher Ausdruck mit einer Zuweisung als Operand!
3) Ausdruck ist binär, aber nicht weiter zerlegbar, da beide Operanden nicht boolesch sind (z.B. (3 < 5)).
4) Ausdruck ist ein weiter zerlegbarer binärer Ausdruck, der mindestens einen zusammengesetzten Operator enthält; also z.B. (A AND B), wobei B = (C OR D) bedeute und alle Variablen im Beispiel als boolesch angenommen seien.

Bei einem Ausdruck mit gleichrangigen Operatoren wird ein Ausdruck von links nach rechts zerlegt.

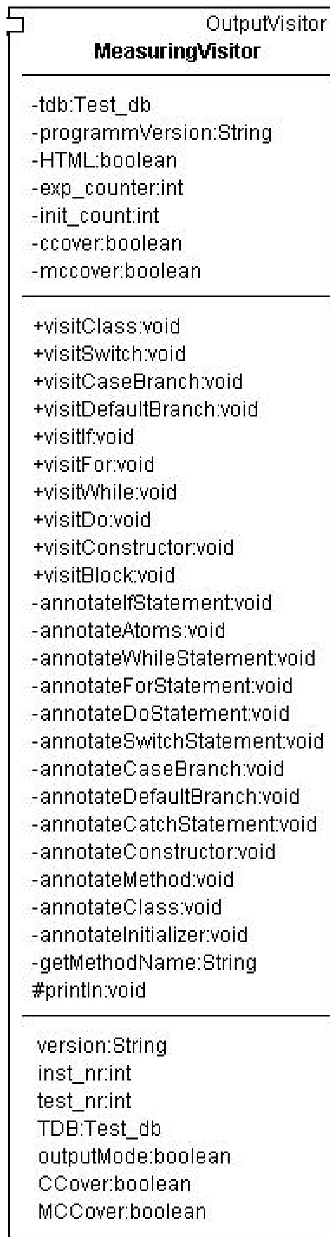

Abbildung 13: Klasse 'MeasuringVisitor'

6.3.3 Entwurf des Evaluierers ‚barat_mess' und des ‚MeasuringVisitors'

Zur Auswertung der mit ‚barat_test' instrumentierten Testläufe wurde das Evaluierer-Programm ‚barat_mess' entworfen. Auch hier ist ‚barat_mess' das Rahmenprogramm für den ‚MeasuringVisitor', der von ‚barat_mess' gestartet wird. ‚barat_mess' und der ‚MeasuringVisitor' lehnen sich eng an den Instrumentierer ‚barat_test' bzw. den ‚InstrumentingVisitor' an. Für jede ‚visit' Methode im ‚InstrumentingVisitor' gibt es eine Entsprechung im ‚MeasuringVisitor'. Der Unterschied besteht nun darin, dass statt der instrumentierenden eine auswertende Methode mit dem Präfix ‚annotate' aufgerufen wird. Die Namensgebung ‚annotate' wurde von den McCluskey-Tools übernommen, auf deren Grundlage die Entwicklung von ‚DoiT' begonnen wurde. Die ‚annotate'-Methoden haben die Aufgabe, die zu einem bestimmten Instrumentierungspunkt gehörenden Messwerte mit Hilfe der Klasse ‚Test_db' aus der Datenbank zu selektieren und als Java-Kommentar in den Quelltext des Originalprogramms einzufügen. Wenn ein entsprechender Parameter gesetzt ist, erfolgt diese Ausgabe im HTML-Format, um die Kommentare mit dem Ergebnis der Evaluierung und den durchlaufenen Quelltext farbig zu unterlegen.

Hier eine Übersicht über das Klassendiagramm des Pakets ‚doit' in dem sowohl ‚barat_test' als auch ‚barat_mess' zusammengefasst sind:

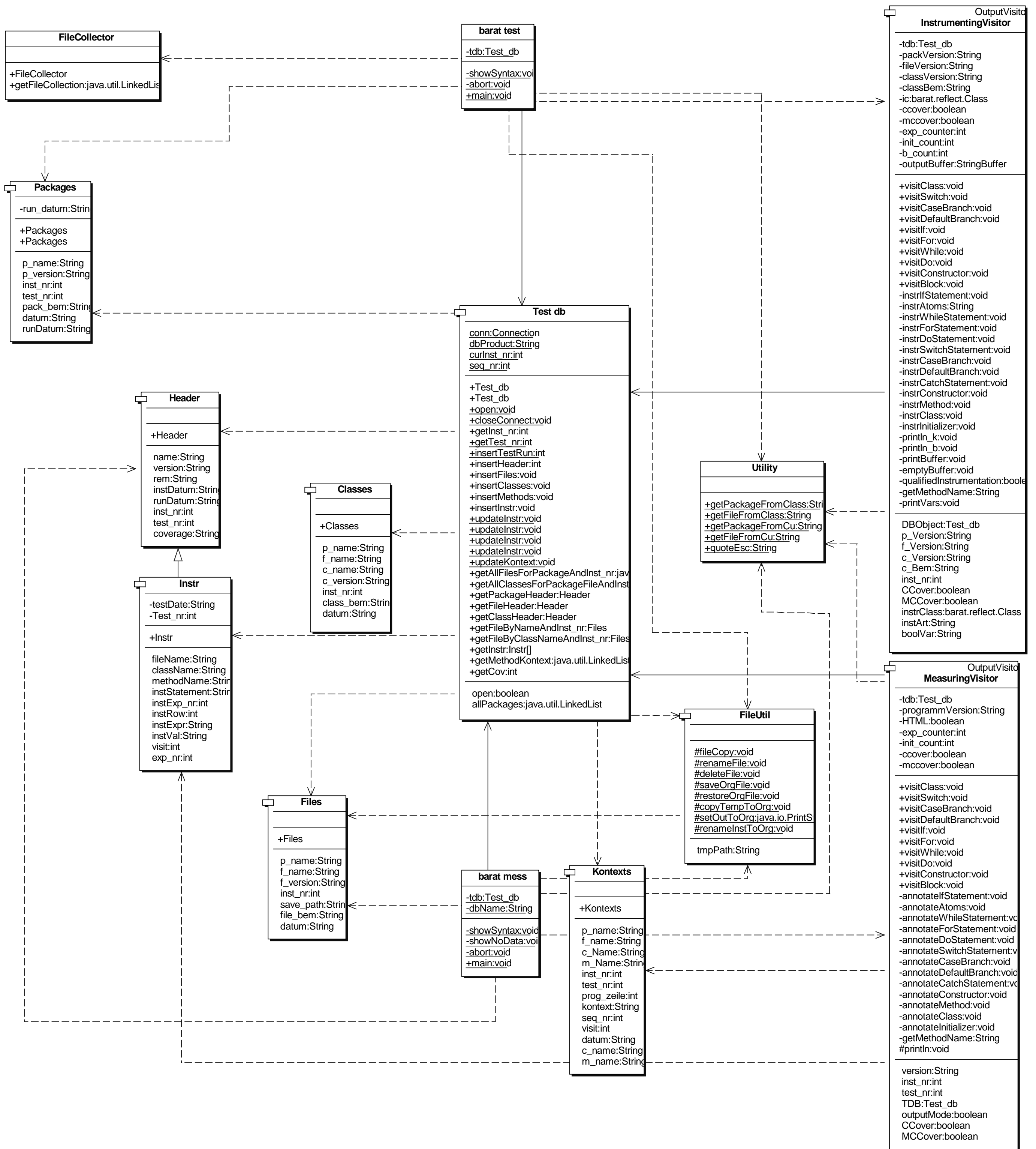

Abbildung 14: Klassendiagramm des Pakets ‚doit'

Die im vorherigen Abschnitt beschriebenen Funktionalitäten – insbesondere diejenigen zur Zerlegung logischer Ausdrücke – gelten für den ‚MeasuringVisitor' analog. Da im

‚MeasuringVisitor' keine isomorphen Ausdrücke gebildet werden müssen, fällt die Implementation von ‚annotateAtoms' etwas einfacher aus als diejenige von ‚instrAtoms'. Algorithmisch sind aber beide Methoden äquivalent.

6.4 Entwurf der Benutzungsschnittstelle

Um die beiden kommandozeilen-orientierten Programme ‚barat_mess' und ‚barat_test' mit einer ergonomischen, grafischen Benutzungsoberfläche auszustatten, wurde das Programm ‚DoitGraph' entwickelt. Kern der Anwendung ist die Java-Swing Klasse ‚JTree', mit deren Hilfe sich hierarchisch strukturierte Daten in einer Baum-Struktur ablegen lassen und sich (analog zum Windows-Explorer von Microsoft) anzeigen und selektieren lassen. Mit Hilfe dieser Struktur können die in 1:n-Beziehungen organisierten Daten der ‚DoiT'-Datenbank einfach und effizient dargestellt werden. Ebenso können die Paket-/Datei- und Klassen Hierarchien aus dem Dateisystem des Hostrechners zur Auswahl für den Benutzer aufbereitet werden.

Um die Event-Queue von Java nicht zu lange warten zu lassen, starten die meisten Funktionen aus der ‚DoitGraph' Benutzungsoberfläche als Java Threads. Die beiden Programme ‚barat_test' und ‚barat_mess' werden allerdings mit einer eigenen Kopie der JVM gestartet, da sich die Programme auf diese Weise ohne Veränderung auch ohne die GUI einsetzen lassen. Ein weiterer Vorteil ist, dass die Ausgaben, die die beiden Programme auf die Standardausgabe-Kanäle des Host-Systems schreiben, leichter abgefangen und auf die ‚DoiT'-Konsole geleitet werden können als dies bei Ausführung als Thread möglich gewesen wäre.

‚DoitGraph' wurde mit einer sich selbst auf das Host-System einstellenden ‚Look-and-Feel' Funktionalität ausgestattet, die in einer Microsoft-Windows-Umgebung den Stil von Windows-Controls, in einer X11-Umgebung den von Motif-Controls und in einer MacIntosh-Umgebung entsprechend gestylte Mac-Controls zeigt.

Die benutzten Java-Standarddialoge wurden soweit wie möglich an den deutschen Sprachgebrauch angepasst.

Es folgt das mit dem Designwerkzeug ‚Together' erstellte Klassendiagramm:

Abbildung 15: Klassendiagramm der graphischen Benutzungsoberfläche (Package ‚doitgui')

6.4.1 Instrumentieren

Beim Instrumentieren hat der Benutzer die Möglichkeit zunächst, entweder mit Hilfe eines Standard Dateidialogs gezielt eine oder mehrere Quelldatei(en) zu öffnen (siehe Use-Case 4.2), oder er gibt den Namen eines ganzen Paketes an, worauf alle Dateien des Paketes nacheinander geöffnet werden (siehe Use-Case 4.3). In der JTree Repräsentation werden dann neben Paket und Datei auch die enthaltenen Klassen angezeigt. Aus dem JTree können Pakete, Dateien oder Klassen (siehe Use-Case 4.1) ausgewählt werden, Versionsbezeichnungen und freie Bemerkungen eingetragen und die Instrumentierung gestartet werden. Falls mehr als eine Datei oder Klasse gewählt wird, erzeugt die GUI eine Steuerdatei mit der Aufzählung der Datei- bzw. Klassen-Namen, die dann an ‚barat_test' übergeben werden. Ganze Pakete sind nur einzeln wählbar.

6.4.2 Evaluieren

Zum Auswerten bereits durchgeführter Testläufe wird der Inhalt der Datenbank bzw. der Tabellen ‚PACKAGES', ‚FILES', und ‚CLASSES' in eine JTree Struktur eingelesen. Zusätzlich werden Instrumentierungsnummern und lfd. Nr. des Testlaufes angegeben. Hier kann jeweils nur ein ganzes Paket (Use-Case 4.7), eine Datei (Use-Case 4.6) oder eine Klasse (Use-Case 4.5) zur Auswertung gewählt werden. Mit der Selektion wird dann der Evaluierer ‚barat_mess' gestartet.

Im gleichen Dialog kann auch die De-Instrumentalisierung (Use-Case 4.8) ausgelöst werden. Dabei wird/werden, die bei der Instrumentalisierung in den Sicherungspfad kopierte(n) Quelldatei(en) wieder an den Ursprungsort zurückkopiert. Für Details: Siehe Benutzerhandbuch im Anhang.

6.4.3 Datenbankoptionen und Datenbankverwaltung

‚DoiT' ist eine Java-Datenbankapplikation. Die Verbindung zur Datenbank wird dementsprechend per JDBC hergestellt. Entwickelt wurde ‚DoiT' mit der Opensource Datenbank ‚MySQL' Version 3.23, implementiert und getestet ist aber auch eine ‚Oracle 8' Version. Zum möglichst einfachen Wechsel des Datenbank-Systems sind in ‚DoitGraph' Dialoge und Klassen zur Anbindung und Konfiguration des Datenbanksystems eingebaut. Die gesamten Datenbank-Routinen für ‚barat_mess' und ‚barat_test' wurden in der Klasse ‚Test_db' gekapselt. Die in ‚DoitGraph' enthaltenen Routinen zum Anlegen, Löschen, Leeren und Sichern der Datenbank wurden nicht ‚hartverdrahtet', sondern für diese Aufgaben wurden datenbankspezifische SQL-

Skripts angelegt, die von einer Leseroutine in ‚DoitGraph' eingelesen und abgearbeitet werden. Vorteil dabei ist unter anderem, dass die Skripts auch außerhalb von ‚DoiT' mit den Werkzeugen der Datenbank (am ‚SQL-Prompt') ausgeführt werden können.

6.4.4 Weitere Funktionen der Benutzungsschnittstelle

Weitere Dialoge und Klassen dienen dazu, die Applikation selbst zu konfigurieren; hier insbesondere den Installations- und Sicherungspfad. Unter letzterem werden die Originaldateien während der Instrumentierungs- und Testphase gesichert. Dies ermöglicht eine ‚in-situ' Instrumentierung des Prüflings. Mittels der Funktion ‚De-Instrumentalisierung' wird die instrumentierte Datei wieder mit der Originaldatei überschrieben.

‚DoitGraph' stellt auch eine knappe Online-Hilfe zur Verfügung und ‚merkt' sich über eine Initialisierungsdatei einige Eigenschaften, wie Fensterpositionen, Rahmengrößen und Selektionspfade. Wichtige Programmausgaben, insbesondere die Ausgaben des Instrumentierers/Evaluierers, werden auf einer System-Konsole mitprotokolliert, da die eigentlichen Werkzeuge ‚barat_test' und ‚barat_mess' wegen der angestrebten Batchtauglichkeit nur auf die Standard I/O Kanäle schreiben können.

6.5 Entwurf der Datenbankschnittstelle

Wie im Abschnitt 5.3.3 des Grobentwurfes beschrieben, wurde die Datenbankschnittstelle von ‚DoiT' in der Klasse ‚Test_db' verkapselt. Die Eigenschaften der Datenbank und der zugehörigen JDBC-Treiber-Klasse werden mittels der Java-Property-Datei ‚DB_PROPS.ini' an die Klasse ‚Test_db' übergeben. Dies erleichtert die Anpassung an andere DBMS, da nur wenig datenbankspezifischer Code direkt in ‚Test_db' abgelegt ist. Im Wesentlichen sind dies die SQL-Statements die Daten von oder in die Datenbank transportieren.

Von der Klasse ‚Test_db' werden sechs Funktionsbereiche abgedeckt:

6.5.1 Öffnen und Schließen der Datenbank

Von den dazu gehörenden Funktionen wird die Datenbank geöffnet und wieder geschlossen. Die Klasse ‚Test_db' kann sowohl dynamisch als Instanz von ‚Test_db' genutzt werden (z.B. während der Instrumentierung), als auch statisch ohne eine Instantiierung. Davon machen die zu prüfenden Klassen ausgiebigen Gebrauch. Dementsprechend sind z.T. parametrierte Konstruktoren vorhanden, die beim Öffnen der Datenbank bereits die Mastertabelle (‚PACKAGES') der Datenbank mit allgemeinen Angaben beschreiben können. Der Grund für diese un-

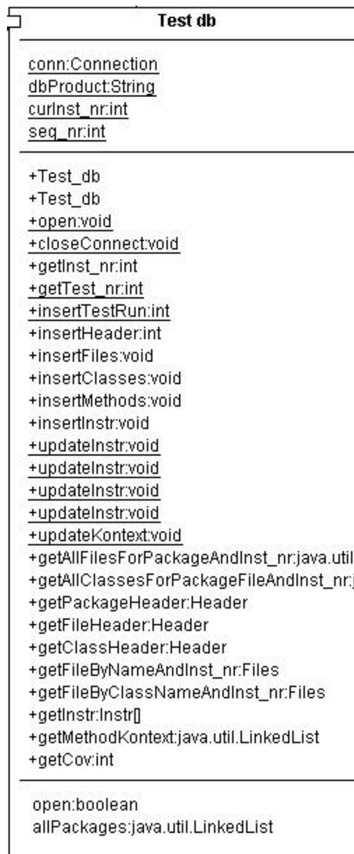

Abbildung 16: Klasse ‚Test_db'

terschiedliche Nutzung von ‚Test_db' liegt in der Art der Anforderung. Während der Instrumentierung kommt es darauf an, einen Barat-Visitor so auszurüsten, dass er mit einer Datenbank kommunizieren kann. Dies geschieht am einfachsten in dem man ihm ein Objekt übergibt, das über alle Datenbank-Verbindungsinformationen verfügt.

Zur Laufzeit des Prüflings würde eine Instantiierung von ‚Test_db' bedeuten, dass dieser Objektname über die gesamte Laufzeit des Prüflings bekannt sein müsste. Da der Prüfling ein ganzes Paket sein kann, würde dies einen erheblichen zusätzlichen Aufwand an Instrumentierungsanweisungen bedeuten. Eine Benutzung von ‚Test_db' zur Laufzeit des Prüflings als statische Klasse ist dagegen einfach und robust gegen Verbindungsverlust.

Vor der Kompilierung des instrumentierten Prüflings müssen die Anweisungen

`Test_db.open();` und

`Test_db.insertTestRun(packageName, Nummer der Instrumentierung);`

von Hand in den Prüfling eingefügt werden; und zwar so, dass sie vor der ersten Instrumentierungsanweisung abgearbeitet werden. Letzteres Statement führt dazu, dass in der Datenbank Datensätze zur Aufnahme der Messwerte dieses Testlaufes bereitgestellt werden und die laufende Nummer der Tests (die ‚Test_Nr') dieser Instrumentierungsnummer um eins erhöht wird. Falls weitere instrumentierte Packages oder Teile davon in diesem Testlauf einer Überdeckungsanalyse unterzogen werden sollen, müssen hier für jedes weitere Package weitere ‚insertTestRun(..)'-Anweisungen eingefügt werden.

Anhand der so erzeugten Datensätze werden nach dem Testlauf die Kennzahlen für die globalen Messwerte ermittelt. Falls ‚insertTestRun(..)' nicht ausgeführt wird und es bereits mindestens einen Testlauf gibt, werden die Ergebnisse unter der zuletzt vergebenen ‚Test_Nr' akkummuliert.

6.5.2 Dokumentation der Instrumentierungspunkte

Mit Hilfe des ‚InstrumentingVisitors' von ‚DoiT' werden während der Instrumentierung die Instrumentierungspunkte mit den zugehörigen Packages, Dateien, Klassen, Methoden und Statements/Zeilen-Nummern in den Tabellen ‚INSTR' und ‚ATOM_EXPR' dokumentiert. Die sicherlich mit am häufigsten benutzte Methode von ‚Test_db' heißt entsprechend ‚insertInstr(..)', die ausschließlich vom ‚InstrumentingVisitor' benutzt wird.

6.5.3 Messwerterfassung

Während der Ausführung des instrumentierten Prüflings werden nur noch Updates auf die Tabelle ‚TEST_DATA' geschrieben. Dort werden die Anzahl der Besuche, die Reihenfolge der Besuche und die Ergebnisse der Bewertungen von Bedingungen abgelegt. Die hierzu aufgerufene Methode von ‚Test_db' steht in der Aufrufhäufigkeit dem ‚insertInstr()' nicht nach und heißt ‚updateInstr(..)'. Die Methode ‚updateInstr(..)' ist wegen der vorkommenden verschiedenen Datentypen von Bewertungsergebnissen polymorph. Die Methodenaufrufe von ‚Test_db.updateInstr(..)' sind letztlich die eigentlichen Instrumentierungsanweisungen, die vom ‚InstrumentingVisitor' in einen Prüfling eingefügt werden.

Von ‚updateInstr(..)' wird auch ‚updateKontext(..)' aufgerufen, und zwar immer dann, wenn die Statement-Art ‚method-start' ist und der Wert nicht ‚static' ist. Im Übergabeparameter ‚Value' von ‚updateInstr(..)' wird dann der mit ‚this.getClass().toString()' ermittelte Objektkontext übergeben und in der Tabelle ‚KONTEXT' gesichert.

6.5.4 Lokale Messwertausgabe

Während der Evaluierung wird der Quelltext des Prüflings erneut geparst. An den Stellen an denen der ‚InstrumentingVisitor' Instrumentierungsanweisungen in den Quelltext eingefügt hat, versucht der ‚MeasuringVisitor' einen zugehörigen Messwert für Besuchshäufigkeit und Bewertungsergebnis aus der Datenbank zu holen.

Um diese Zuordnung von Quelltext-Strukturen zu Messergebnissen und die Rückgabe dieser Messwerte bereitzustellen, enthält ‚Test_db' die Methode ‚getInstr(..)'. Die Methode gibt, abhängig vom instrumentierten Statement, ein oder zwei ‚Instr'-Objekte zurück, die alle Informationen über den jeweiligen Instrumentierungspunkt zurückgeben. Ein oder zwei ‚Instr'-Objekte deshalb, weil bei Instrumentierungen in denen Bedingungen vorkommen oder die nur aus einer (atomaren) Bedingung bestehen, immer <u>beide</u> möglichen Wahrheitswerte für sich bewertet werden.

Zusätzlich gibt es zur Ausgabe der Objekt-Kontexte von Methodenaufrufen die Methode ‚getMethodKontext(..)', die eine ‚LinkedList' mit den vorkommenden Ausprägungen der Objekt-Kontexte des jeweiligen Methodenaufrufes und deren Häufigkeit zurückgibt. Diese Informationen werden jeweils am Beginn einer evaluierten Methode als Kommentar ausgegeben.

6.5.5 Globale Messwertausgabe

Mit Hilfe der Methode ‚getCov(..)' werden globale Kennzahlen für Überdeckungsmetriken für Packages, Dateien oder Klassen ermittelt. Die Methode ist so konstruiert, dass alle Metriken mit dieser einen Methode aus der Datenbank ermittelt werden können und zugleich weitere Metriken leicht hinzugefügt werden können, da das zugehörige SQL-Statement dynamisch, in Abhängigkeit von der Parametrierung der Methode, erzeugt wird.

6.5.6 Informationen für ‚DoitGraph'

Einige rein selektierende Methoden dienen der graphischen Benutzungsoberfläche ‚DoitGraph'. Mit Hilfe dieser Funktionalitäten lassen sich Angaben über Packages, Dateien, Klassen und Methoden gewinnen und in einer JTree-Darstellung in ‚DoitGraph' übersichtlich darstellen.

7 Beispiele

Nach einer allgemeinen Betrachtung and Anleitung zum Einsatz von ‚DoiT' sollen in diesem Kapitel Beispiele für Messungen an realen Java-Programmen gegeben werden. Nach diesem allgemeinen ‚Kochrezept' zur Anwendung von ‚DoiT' wird zunächst demonstriert, wie gering die Überdeckung des Programmcodes durch bloßes Durchspielen aller Funktionalitäten der Benutzungsschnittstelle sein kann. In einem weiteren Versuch werden dann die Testfälle, mit denen der Modultester JUnit sich selbst testet, instrumentiert.

7.1 Allgemeine Vorgehensweise beim Instrumentieren und bei der Überdeckungsanalyse mit ‚DoiT'

Zunächst wird eine Schritt-für-Schritt-Anleitung für das Überdeckungstesten mit ‚DoiT' gegeben:

- ‚DoiT' Start.
- Auswahl des zu testenden Packages/der zu testenden Dateien/Klassen.
- Eventuell Eingabe einer Versionsbezeichnung und einer freien Bemerkung für jede ausgewählte Programmeinheit.
- Start der Instrumentierung, wobei die Hinweise auf der ‚DoiT'-Konsole beachtet werden sollten.
- Manuelles Einfügen von ‚doit.Test_db.open()' und ‚insertTestRun(PackageName, Instr_nr)'. Diese Angaben werden auf der ‚DoiT'-Konsole angezeigt und können von dort per copy&paste in den Prüfling eingefügt werden.
- Re-Kompilation des instrumentierten Prüflings.
- Ausführen des Prüflings mit geeigneten Testfällen.
- Nach Beendigung des Testlaufes <Anzeigen von Testläufen> aus dem ‚DoiT' Datei-Menu wählen. Angezeigt werden die Paketnamen und dahinter die laufende Nr. der Instrumentierung und des Testlaufes, getrennt durch Unterstriche.
- Auswahl eines Eintrags in der JTree-Darstellung und Ausführen von <Auswerten von Testläufen>.
- Betrachtung der Überdeckungsanalyse im ‚DoiT'-Ausgabefenster mit der Option, das Ergebnis in einer Text- oder HTML-Datei zu speichern.

Das Überdeckungstesten mit ‚DoiT' lässt sich gut in den üblichen Entwicklungszyklus aus Test, Debugging und Änderung einbauen. Möchte man nach der Änderung am zu testenden

Programm prüfen, ob sich nun auch diese oder jene Überdeckungskennzahl verändert hat, kann das geänderte Programm sofort erneut instrumentiert werden.

Hier bietet es sich an, nicht gleich die gesamte Applikation zu instrumentieren, sondern man sollte sich wegen der Fülle der aus der Instrumentierung zu erwartenden Daten und Kennzahlen auf diejenigen Bereiche beschränken, die einen Überdeckungstest erforderlich und sinnvoll machen. So sind z.B. reine Datenklassen mit ‚get'- und ‚set'-Methoden hierfür eher uninteressant; es sei denn man ist gerade an der Methodenüberdeckung dieser Klassen interessiert. Klassen mit viel ‚Logik' und/oder Ausnahmebehandlungen können aber im allgemeinen sehr viel lohnender für einen Überdeckungstest sein.

Der gerade geänderte Quelltext wird beim Instrumentieren von ‚DoiT' unter einer neuen Instrumentierungsnummer im Sicherungspfad abgelegt. Nach der Instrumentierung sind die instrumentierten Programmmodule neu zu Kompilieren und der Test auszuführen. Danach ist in der ‚DoiT'-GUI sofort die Überdeckungsanalyse des Prüflings abrufbar bzw. die Instrumentierung für die weiteren Entwicklungsarbeiten wieder rückgängig zu machen.

Die folgende Grafik zeigt noch einmal schematisch den Ablauf einer Instrumentierung und eines Testlaufes mit dem ‚DoiT' Paket bzw. den Bestandteilen ‚barat_test' und ‚barat_mess'. Die grafische Benutzungsschnittstelle ist hier zur Vereinfachung weggelassen. Die Art der Darstellung lehnt sich an das in den späten 1990er Jahren von SUN entwickelte Werkzeug JavaScope an [JScope1997]. JavaScope wurde von Sun leider nicht weiter entwickelt und ist heute nicht mehr erhältlich (siehe 3.1.3).

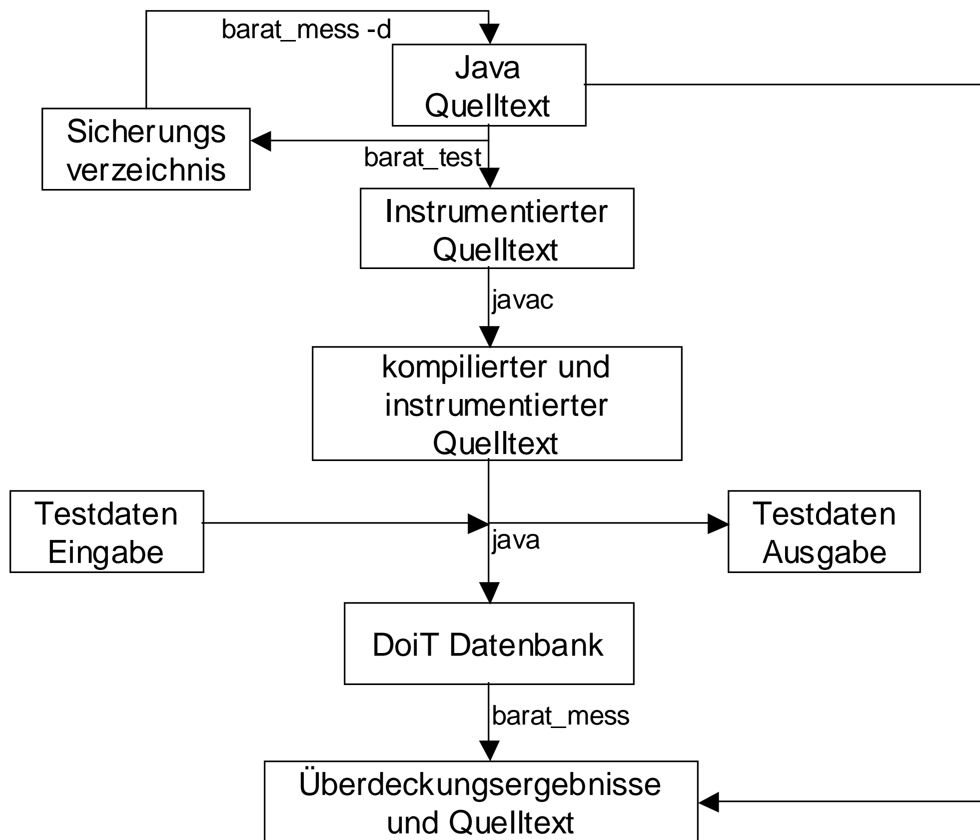

Abbildung 17: Ablaufplan für die Instrumentierung mit ‚DoiT'

7.2 Messungen an der ‚DoiT'-GUI

Während der Entwicklung wurde ‚DoiT' mit einer Testklasse (die nur eine Nonsens-Semantik besitzt, aber dafür viele verschiedene Java-Konstrukte enthält) fortlaufend auf korrekte Funktion geprüft – insbesondere auf syntaktische Korrektheit des instrumentierten Codes. Außerdem wurde - so weit es möglich war - auf eine ‚ansprechende' Formatierung des instrumentierten Codes geachtet.

Nach der vorläufigen Fertigstellung des Programms, wurde ‚DoiT' zunächst an sich selbst, das heißt am Code der Benutzungsoberfläche getestet. Das Projekt hat ca. 3000 Zeilen Java Code und dieser konnte in weniger als einer Minute instrumentiert werden. Nach Neuübersetzung konnte die instrumentierte ‚DoiT'-GUI auf einem 450 MHz PII Prozessor mit kaum merklicher Verlangsamung ausgeführt werden. Hier zahlt es sich aus, dass nur Updates und keine Inserts zur Laufzeit des Prüflings in die Datenbank geschrieben werden. Die Überdeckungs-Messungen an ‚DoiT' selbst (ohne den Objekt-Kontext von Methodenaufrufen), führten beim einmaligen,

manuellen Ausführen aller Funktionen der Graphischen Benutzungsschnittstelle (mit Ausnahme der Datenbankadministrationsfunktionen) zu folgenden globalen Messwerten:

```
// Ergebnisse des instrumentierten Testlaufes vom 01.12.2002 11:19
// lfd. Test-Nr. dieser Inst-Nr: 2
// Bemerkung zur Instrumentierung: Test der Doit-GUI
//
// Globale Messergebnisse für Package doitgui :
// Klassenüberdeckung                      : 55 %
// Methodenüberdeckung                     : 71 %
// Zweigüberdeckung                        : 44 %
// Bedingungsüberdeckung                   : 52 %
// Minimale Mehrfachbedingungsüberdeckung : 52 %
// Schleifenüberdeckung                    : 64 %
// Ausnahmeüberdeckung                     : 3 %
```

7.3 Messungen an und mit JUnit

Ein weiterer Test von ‚DoiT' wurde mit Hilfe von JUnit durchgeführt. Im Standard-Lieferumfang von JUnit sind Standard-Testfälle enthalten (junit.tests.*). Diese Testklassen wurden mit ‚DoiT' instrumentiert und dann ausgeführt (java junit.tests.AllTests). Die Messwerte wurden zunächst ohne die Dokumentation des Objekt-Kontextes durch ‚DoiT' gewonnen. Die Ergebnisse sind im folgenden zusammengestellt:

Für das Package ‚junit.tests.extensions':

```
// Package-Name: junit.tests.extensions, Version: 3.8.1,
Instrumentierung Nr. 1, instrumentiert am 26.12.2002
// Ergebnisse des instrumentierten Testlaufes vom 26.12.2002 15:27
// lfd. Test-Nr. dieser Inst-Nr: 1
// Bemerkung zur Instrumentierung: Test von DoiT
//
// Globale Messergebnisse für Package junit.tests.extensions :
// Klassenüberdeckung                      : 93 %
// Methodenüberdeckung                     : 93 %
// Zweigüberdeckung                        : 93 %
// Bedingungsüberdeckung                   : 100 %
// Minimale Mehrfachbedingungsüberdeckung : 100 %
// Schleifenüberdeckung                    : 100 %
// Ausnahmeüberdeckung                     : 100 %
```

Für das Package ‚junit.tests.framework':

```
// Package-Name: junit.tests.framework, Version: 3.8.1,
Instrumentierung Nr. 1, instrumentiert am 26.12.2002
// Ergebnisse des instrumentierten Testlaufes vom 26.12.2002 15:27
// lfd. Test-Nr. dieser Inst-Nr: 1
// Bemerkung zur Instrumentierung: Test von DoiT
//
// Globale Messergebnisse für Package junit.tests.framework :
// Klassenüberdeckung                      : 88 %
// Methodenüberdeckung                     : 88 %
// Zweigüberdeckung                        : 90 %
// Bedingungsüberdeckung                   : -1 %
```

```
// Minimale Mehrfachbedingungsüberdeckung : -1 %
// Schleifenüberdeckung                   : -1 %
// Ausnahmeüberdeckung                    : 100 %
```

Für das Package ‚junit.tests.runner':

```
// Package-Name: junit.tests.runner, Version: 3.8.1, Instrumentierung
Nr. 1, instrumentiert am 26.12.2002
// Ergebnisse des instrumentierten Testlaufes vom 26.12.2002 15:27
// lfd. Test-Nr. dieser Inst-Nr: 1
// Bemerkung zur Instrumentierung: Test von DoiT
//
// Globale Messergebnisse für Package junit.tests.runner :
// Klassenüberdeckung                     : 86 %
// Methodenüberdeckung                    : 80 %
// Zweigüberdeckung                       : 78 %
// Bedingungsüberdeckung                  : 75 %
// Minimale Mehrfachbedingungsüberdeckung : 75 %
// Schleifenüberdeckung                   : 100 %
// Ausnahmeüberdeckung                    : -1 %
```

Als weiterer Test wurde dann das JUnit-Rahmenwerk selbst instrumentiert und mit einem nichtinstrumentierten Testfall ‚junit.tests.AllTests' getestet. Für das JUnit-Rahmenwerk selbst wurden bei der Ausführung von ‚junit.tests.AllTests' folgende Werte ermittelt:

Für das Package ‚junit.extensions':

```
// Package-Name: junit.extensions, Version: 3.8.1, Instrumentierung
Nr. 2, instrumentiert am 29.12.2002
// Ergebnisse des instrumentierten Testlaufes vom 29.12.2002 18:58
// lfd. Test-Nr. dieser Inst-Nr: 2
// Bemerkung zur Instrumentierung: Testen von DoiT mittels JUnit
//
// Globale Messergebnisse für Package junit.extensions :
// Klassenüberdeckung                     : 100 %
// Methodenüberdeckung                    : 83 %
// Zweigüberdeckung                       : 80 %
// Bedingungsüberdeckung                  : 87 %
// Minimale Mehrfachbedingungsüberdeckung : 87 %
// Schleifenüberdeckung                   : 100 %
// Ausnahmeüberdeckung                    : 50 %
```

Für das Package ‚junit.framework':

```
// Package-Name: junit.framework, Version: 3.8.1, Instrumentierung Nr.
2, instrumentiert am 29.12.2002
// Ergebnisse des instrumentierten Testlaufes vom 29.12.2002 18:58
// lfd. Test-Nr. dieser Inst-Nr: 2
// Bemerkung zur Instrumentierung: Testen von DoiT mittels JUnit
//
// Globale Messergebnisse für Package junit.framework :
// Klassenüberdeckung                     : 100 %
// Methodenüberdeckung                    : 77 %
// Zweigüberdeckung                       : 74 %
// Bedingungsüberdeckung                  : 77 %
// Minimale Mehrfachbedingungsüberdeckung : 77 %
// Schleifenüberdeckung                   : 100 %
// Ausnahmeüberdeckung                    : 41 %
```

Für das Package ‚junit.runner':

```
// Package-Name: junit.runner, Version: 3.8.1, Instrumentierung Nr. 2,
instrumentiert am 29.12.2002
// Ergebnisse des instrumentierten Testlaufes vom 29.12.2002 18:58
// lfd. Test-Nr. dieser Inst-Nr: 2
// Bemerkung zur Instrumentierung: Testen von DoiT mittels JUnit
//
// Globale Messergebnisse für Package junit.runner :
// Klassenüberdeckung                     : 55 %
// Methodenüberdeckung                    : 50 %
// Zweigüberdeckung                       : 40 %
// Bedingungsüberdeckung                  : 42 %
// Minimale Mehrfachbedingungsüberdeckung : 41 %
// Schleifenüberdeckung                   : 75 %
// Ausnahmeüberdeckung                    : 9 %
```

Die mit '-1' gekennzeichneten Messwerte deuten darauf hin, dass in den genannten Packages kein Statement der entsprechenden Kategorie vorkam. Also beispielsweise Ausnahmeüberdeckung = -1 => keine **try-catch** Klauseln im Package angetroffen.

Die genannten globalen Messwerte für die verschiedenen Überdeckungsmaße beim JUnit-Test sind bereits relativ hoch (>80%). Nach [Bin2000] sind in größeren und komplexen Programmen 80-85% Zweigüberdeckung ein guter Wert für eine Test-Suite. In einem noch besseren Licht erscheinen die Zahlen, wenn man sich die Ergebnisse für die getesteten Klassen im Einzelnen ansieht und diejenigen Klassen im jeweiligen Package aus der globalen Betrachtung herauslässt, die vom Test offenbar gar nicht berührt worden sind, wie z.B. die Klasse ‚junit.tests.runner.LoadedFromJar'. Ebenso werden die Klassen-Messwerte durch Methoden verwässert, die nur zu Testzwecken eingefügt wurden (wie z.B. main-Methoden, um eine Klasse als einzelnes Modul testen zu können). Bei der Beurteilung der numerischen Kennzahlen für die Überdeckungsmaße, sollte also immer auch der von ‚DoiT' ausgegebene, bewertete Quelltext analysiert werden, um zu erkennen ob und in welchem Umfang die wirklich kritischen Bereiche eines Programms von einem Test berührt wurden oder nicht.

Der Testlauf mit dem instrumentierten Rahmenwerk erbrachte z.T. geringe Überdeckungen insbesondere im Package ‚junit.runners' durch den ‚junit.tests.AllTests'. Tendenziell zeichnet sich aber die gleiche Reihenfolge ab wie bei dem instrumentierten Testfall ‚junit.tests.AllTests': Beste Überdeckung in ‚junit.extensions', dicht gefolgt von ‚junit.framework', und am schlechtesten ist die Überdeckung in ‚junit.framework'. Erstaunlich dabei ist, dass noch nicht einmal eine 100%ige Methodenüberdeckung erreicht wird.

Durch den Test von ‚DoiT' gegen den JUnit-Test konnten auch quantitative Messwerte für die Verzögerung der Ausführung des Prüflings durch die Instrumentierung ermittelt werden:

Messung	Ausführung von	Objektkontext	Messwert [s]
1	java junit.tests.AllTests ohne Instrumentierung	--	2,2
2	java junit.tests.AllTests mit Instrumentierung von junit.tests	ohne	16
3	java junit.tests.AllTests mit Instrumentierung von junit.framework, junit.extensions, junit.runner	ohne	127
4	Wie vor, aber unter Protokollierung des jeweiligen Methodenkontextes	mit	207

Alle angegebenen Werte wurden mit MySQL als RDBMS, das auf dem gleichen Rechner wie die Testumgebung installiert war, ermittelt. Die speziell schon im vorletzten Fall (3) lange Laufzeit, resultiert aus der besonderen Struktur der JUnit-Tests, wobei bestimmte Tests und damit bestimmte Datenbank I/O-Anweisungen wiederholt in Schleifen ausgeführt werden.

Der letztgenannte Wert (4) bedeutet für diesen Anwendungsfall einen zusätzlichen Zeitaufwand um den Faktor 1,6. Ein vertretbarer Wert, wenn man bedenkt, dass bei jedem Methodenaufruf im Prüfling nun zusätzlich eine Introspektion und ein Datenbank-Insert oder –Update vorzunehmen ist. Die Unterscheidung zwischen beiden letzteren Operationen erfordert zusätzlich noch jeweils einen Enthaltenseins-Test. Außerdem muss bei den JUnit-Testfällen berücksichtigt werden, dass viele Methoden des JUnit-Rahmenwerks jeweils etliche hundertmal aufgerufen werden. Die globalen Messwerte bleiben durch die Dokumentation des Objektkontextes natürlich unbeeinflusst. Deshalb wird hier auf ihre neuerliche Wiedergabe verzichtet. Beispielhaft soll hier aber die Evaluierung der Methode aus ‚junit.framework.TestCase' wiedergegeben werden. An diesem Beispiel ist schön zu erkennen, auf welche Objekte diese Methode im Verlauf des Tests wie oft angewendet wurde:

```
public void setName(String name) {
  /* Die Methode (junit.framework.TestCase.setName) wurde 87 mal aufgerufen. */
  /* Beobachtet wurden folgende(r) Kontext(e) und Häufigkeit(en): */
  /* Die Methode wurde im Kontext von class junit.tests.extensions.ActiveTestTest 4 mal aufgerufen. */
  /* Die Methode wurde im Kontext von class junit.tests.extensions.ExceptionTestCaseTest 5 mal aufgerufen. */
  /* Die Methode wurde im Kontext von class junit.tests.extensions.ExtensionTest 4 mal aufgerufen. */
  /* Die Methode wurde im Kontext von class junit.tests.framework.AssertTest 14 mal aufgerufen. */
  /* Die Methode wurde im Kontext von class junit.tests.framework.ComparisonFailureTest 11 mal aufgerufen. */
  /* Die Methode wurde im Kontext von class junit.tests.framework.DoublePrecisionAssertTest 7 mal aufgerufen. */
  /* Die Methode wurde im Kontext von class junit.tests.framework.InheritedTestCase 2 mal aufgerufen. */
  /* Die Methode wurde im Kontext von class junit.tests.framework.NoArgTestCaseTest 2 mal aufgerufen. */
  /* Die Methode wurde im Kontext von class junit.tests.framework.NotPublicTestCase 1 mal aufgerufen. */
  /* Die Methode wurde im Kontext von class junit.tests.framework.NotVoidTestCase 1 mal aufgerufen. */
  /* Die Methode wurde im Kontext von class junit.tests.framework.OneTestCase 2 mal aufgerufen. */
  /* Die Methode wurde im Kontext von class junit.tests.framework.OverrideTestCase 1 mal aufgerufen. */
  /* Die Methode wurde im Kontext von class junit.tests.framework.Success 1 mal ufgerufen. */
  /* Die Methode wurde im Kontext von class junit.tests.framework.TestCaseTest 13 mal aufgerufen. */
  /* Die Methode wurde im Kontext von class junit.tests.framework.TestImplementorTest 1 mal aufgerufen. */
  /* Die Methode wurde im Kontext von class junit.tests.framework.TestListenerTest 3 mal aufgerufen. */
  /* Die Methode wurde im Kontext von class junit.tests.runner.BaseTestRunnerTest 1 mal aufgerufen. */
  /* Die Methode wurde im Kontext von class junit.tests.runner.SimpleTestCollectorTest 1 mal aufgerufen. */
  /* Die Methode wurde im Kontext von class junit.tests.runner.SorterTest 1 mal aufgerufen. */
  /* Die Methode wurde im Kontext von class junit.tests.runner.StackFilterTest 1 mal aufgerufen. */
  /* Die Methode wurde im Kontext von class junit.tests.runner.TestCaseClassLoaderTest 2 mal aufgerufen. */
  /* Die Methode wurde im Kontext von class junit.tests.runner.TextFeedbackTest 5 mal aufgerufen. */
  /* Die Methode wurde im Kontext von class junit.tests.runner.TextRunnerTest 4 mal aufgerufen. */
  {
    this.fName = name;
  }
}
```

Abbildung 18 : Beispiel für den Objekt-Kontext einer Methode

8 Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde ein datenbankgestützter Instrumentierer/Evaluierer für Überdeckungsanalysen entwickelt, der das strukturelle Testen von Java-Programmen unterstützt.

Dazu wurden zunächst die speziellen Eigenschaften der Sprache Java untersucht, soweit sie für das Messen und Erfassen von Überdeckungsmaßen wichtig erschienen.

Im Anschluss daran erfolgte eine gründliche Bestandsaufnahme des kommerziellen Marktes und des Opensource-Angebots für Java-Überdeckungstestwerkzeuge. Das Angebot auf dem Markt für Software zum automatischen Instrumentieren von Java-Quelltexten ist nicht besonders groß, und wie auf jedem Markt besitzen die einzelnen Marktteilnehmer Stärken und Schwächen. Keiner der ermittelten Anbieter/Autoren, weder aus dem Opensource, noch aus dem kommerziellen Bereich, bietet heute ein datenbankgestütztes, komfortabel zu bedienendes Werkzeug zur Überdeckungsanalyse an. Dabei werden meist auch nur recht einfache Testmetriken von den Werkzeugen angeboten.

Schon sehr früh zeigte sich deshalb, dass ein Werkzeug, welches einerseits imstande wäre, möglichst viele der Standardmetriken, aber auch mehr objektorientierte Kennzahlen zu liefern, und das andererseits als Datenbankanwendung implementiert ist, nicht erhältlich ist. Hier fiel also die eingangs beschriebene Wahl zwischen Anpassung und Neukonzeption relativ leicht. Wegen der größeren Flexibilität wurde ein gut konzeptionierter und gewarteter Parser aus dem Opensource-Bereich als Grundlage für die Neuentwicklung herangezogen.

Mit dem entwickelten, datenbankgestützten Überdeckungstest-Werkzeug ‚DoiT' können Java-Klassen, -Dateien oder ganze Java-Pakete schnell und einfach für Überdeckungstests instrumentiert werden. Beim Instrumentieren der Java-Quellen werden die Instrumentierungspunkte zur Block-, Entscheidungs-, Schleifen-, Selektions-, Klassen-, Methoden- und Ausnahme-Instrumentierung in einer relationalen Datenbank zusammen mit einer Schlüsselnummer der Instrumentierung für dieses Paket abgelegt. Nach Neuübersetzung der instrumentierten Quellen schreiben diese während der Ausführung Messwerte über Besuchshäufigkeiten und –reihenfolge in die Datenbank.

Besonders vorteilhaft ist dabei der Einsatz einer relationalen Datenbank, um während der Instrumentierung die ermittelten strukturellen Daten eines Prüflings redundanzfrei abzulegen bzw. zur Laufzeit des instrumentierten Prüflings, die an diesen Punkten anfallenden, bewertenden Daten aufzunehmen. Bei der Evaluierung fällt es dann leicht, die so archivierten Daten an-

hand zusammengesetzter Schlüssel aus der Datenbank zu selektieren und dem Tester zu präsentieren.

Mit Hilfe eines besonderen Algorithmus zur Zerlegung von zusammengesetzten logischen Ausdrücken aus Konditionalanweisungen und Schleifensteuerungen, ist es gelungen in ‚DoiT' auch die Instrumentierung und seiteneffektfreie Bewertung von logischen Ausdrücken zu realisieren und die zur Laufzeit ermittelten Bewertungen und Besuchshäufigkeiten ebenfalls in der Datenbank abzulegen.

Ein weitere Besonderheit von ‚DoiT' ist die Fähigkeit, nicht nur Methodenaufrufe innerhalb einer Klasse zu dokumentieren, sondern die verschiedenen Objektkontexte der Aufrufe einer polymorphen Methode auch aus abgeleiteten Klassen sichtbar zu machen. Auf diese Weise kann getestet werden, ob und wie oft Methoden der jeweiligen Oberklasse in davon abgeleiteten Unterklassen aufgerufen werden.

Mit Hilfe der ermittelten Messwerte, lassen sich Kennzahlen für Überdeckungsmaße wie C_0-, C_1-, C_2-, minimale-Mehrfachbedingungs-, Klassen-, Methoden- und Ausnahmeüberdeckung angeben. Außerdem kann der getestete Quelltext zusammen mit den ermittelten Werten und Kontexten ansprechend als HTML-Text dargestellt werden. Die vom Test unberührten Anweisungen werden dabei farblich hervorgehoben.

Bei diversen Tests zeigte ‚DoiT' ein gut beherrschbares Verhalten. Ein Test gegen den Quelltext seiner eigenen GUI mit 3000 Zeilen erledigte das Werkzeug in vertretbarer Zeit und mit nur geringem Performanzverlust des instrumentierten Prüflings.

Für zukünftige Erweiterungen von ‚DoiT' sind folgende Features wünschenswert und angedacht:

- Überdeckungsmetrik für den ‚modified condition/decision coverage Test' der z.B. in der Avionik für sicherheitskritische Software in Flugzeugen gefordert wird. Dabei muss u.a. jede atomare Teilentscheidung einer zusammengesetzten Logik ihren Einfluss auf das Gesamtergebnis einer logischen Kette ‚ceteris paribus' - also unter Beibehaltung der Werte der übrigen Glieder - nachweisen. [Lig2001]
- Einbau der ‚Unvollständigen Evaluierung' (short-circuit-evaluation) von zusammengesetzten logischen Ausdrücken
- Verbesserung des Laufzeitverhaltens auch bei I/O-intensiven Testläufen durch asynchrones Schreiben der Datenbank-Updates. Dies kann beispielsweise durch einen SQL-Puffer und einen Wächter-Prozess bzw. -Thread erreicht werden, der erst beim Er-

reichen einer gewissen Anzahl von DB-Updates asynchron aus dem Puffer in die Datenbank überträgt.

- Versuche zum Instrumentieren von ternären Konditionaloperatoren
- Erweiterung des Datenmodells zur gemeinsamen Instrumentierung ganzer Softwareprojekte
- Erkennung und Berücksichtigung von Vererbungshierarchien und auf Wunsch Instrumentierung entsprechender Oberklassen (falls diese im Quelltext vorliegen)
- Test und Anbindung weiterer Datenbankmanagementsysteme

9 Glossar

Abstrakter Syntax Baum (AST)

Sei N ein Alphabet von Nichtterminalen, sei Σ ein Alphabet von Terminalen, $P \subseteq N \times (N \cup \Sigma)^*$ sei eine Menge von Produktionsregeln und $S \in N$ sei ein Startsymbol.

Sei $G=(N, \Sigma, P, S)$ eine kontextfreie Grammatik. Sei T ein Baum, dessen innere Knoten mit Nichtterminalen und dessen Blätter mit Terminalen von G oder mit dem leeren Wort ε markiert sind. T heißt (abstrakter) Syntaxbaum oder Ableitungsbaum für das Wort $w \in \Sigma^*$ und für $X \in N$, falls gilt:

(i) Für jeden inneren Knoten p, der mit $Y \in N$ markiert ist und dessen Söhne (von links nach rechts) $q_1,...,q_n$ mit $Q_1,...,Q_n \in (N \cup \Sigma)$ markiert sind, gibt es eine Produktion $Y \rightarrow Q_1....Q_n$ in P. Falls p einen einzigen Sohn hat, der mit ε markiert ist, so existiert eine Produktion $Y \rightarrow \varepsilon$.

(ii) Die Wurzel des Baumes ist mit X markiert, und die Konkatenation der Markierungen der Blätter ergibt w.

Der Unterschied zwischen konkreten und abstrakten Syntaxbäumen besteht darin, dass bei ersteren eine vereinfachte Darstellung verwendet wird, bei denen die Struktur weniger an grammatischen Kategorien, als an der Bedeutung des Konstrukts, d.h. an den durchzuführenden Operationen orientiert ist [GütErw1999].

Ant

'Another Neat Tool'. Ant ist ein 'make' Programm für Java-Projekte. Damit lassen sich Abhängigkeiten definieren und damit das Kompilieren und Pflegen von großen Java-Applikationen automatisieren. Im Gegensatz zu 'make' werden die Steuerdateien in XML Syntax verfasst [IntAnt].

Fehler

Ein Fehler ist das Abweichen eines berechneten, beobachteten oder gemessenen Wertes oder eines Zustandes der Betrachtungseinheit von dem spezifizierten oder theoretisch richtigen Wert bzw. [erwarteten] Zustand [PagSix1994]. Man unterscheidet innere (Fehlerursache) und äußere Fehler (Fehlerwirkung).

GUI

'Graphic User Interface': Graphische Benutzungsoberfläche eines Programms.

Instrumentieren

Erweiterung eines Programms um Messpunkte, die bestimmte Zustände eines Programms an bestimmten Stellen ausgeben können ('Tracen')[ClaSchw1993].

JDBC — Java Database Connectivity. Eine ‚Middleware' genannte Sammlung von systemspezifischen Klassen die Datenbanksysteme so verkapseln , dass die Verbindung zur Datenbank aus der Sicht eines Java-Programms unabhängig wird vom verwendeten Datenbankfabrikat. Die z.T. stark unterschiedlichen Dialekte der Abfragesprache (wie z.B. SQL) werden dadurch allerdings nicht beseitigt.

Parser — ‚Zerteiler'. Ein Parser realisiert die syntaktische Analyse von Programmen gemäß einer bestimmten Grammatik. Die syntaktische Analyse soll die über die lexikalische Analyse hinausgehende Struktur eines Programms herausfinden. [WiMau1997]

Programmverifikation — Nachweis der Korrektheit eines Programms durch mathematischen Korrektheitsbeweis bzgl. der Programmspezifikation. Verifizierende Verfahren erfordern eine formale Programmspezifikation, sind sehr komplex und nur eingeschränkt automatisierbar. Wichtigster Vertreter ist das Hoare-Kalkül.

Relationales Datenbank Management System (RDBMS) — Ein Softwaresystem, das es ermöglicht, eine relationale Datenbank zu definieren, Daten zu speichern, zu verändern und zu löschen, sowie Anfragen an die Datenbank zu stellen.
Eine Datenbank ist eine integrierte Ansammlung von Daten, die allen Benutzern eines Anwendungsbereiches als gemeinsame Basis aktueller Information gilt [Schlag1994]

Short Circuit Evaluation — Auch ‚Lazy-Evaluation' genannt. Auswertungsstrategie bei der Bewertung zusammengesetzter logischer Ausdrücke in denen UND- bzw. ODER- Verknüpfungen vorkommen, im deutschen Sprachgebrauch ‚Unvollständige Evaluation' genannt. Ist der erste Teilausdruck FALSCH, können die folgenden Glieder eines Konjunktions-Ausdrucks den Wahrheitswert des Gesamtausdruckes nicht mehr beeinflussen. Das Gleiche gilt für den Wahrheitswert WAHR und die Disjunktion [Bin2000].

SQL — Structured Query Language. Die Standardabfragesprache für relationale Datenbanksysteme. Trotz aller Standardisierungsbemühungen gibt es zwischen verschiedenen Datenbanksystemen hinsichtlich der SQL-Syntax nur so etwas wie einen kleinsten gemeinsamen Nenner. Insbesondere die enthaltenen SQL-Funktionen zum Formatieren und Bearbeiten von Daten unterscheiden sich meist von System zu System [Schlag1994].

Testen	Als ‚Testen' bezeichnet man das gezielte Suchen nach Fehlern. Getestet werden sämtliche Dokumente, die bei der Softwareentwicklung entstehen. Dokumente, die Testverfahren unterworfen werden, bezeichnet man als ‚Prüflinge'. Dazu gehört neben anderen Dokumenten natürlich der Quelltext des Programms. Testen ist eine Kontrollfunktion im Softwareentwicklungsprozess. Sie umfasst nicht die Fehlerkorrektur [PagSix1994].
Trace	‚Spur' bzw. ‚Fährte'. Als Trace bezeichnet man eine Aufzeichnung über die Ablaufreihenfolge der Anweisungen eines Programms, eventuell angereichert mit den Resultaten bestimmter Zwischenergebnisse und Entscheidungen. Ein Trace wird im allgemeinen zur Fehlersuche und zur Überdeckungs-Analyse beim Ausführen von Tests eingesetzt.
UML	‚Unified Modeling Language': Eine Metasprache zur Modellierung und grafischen Darstellung objektorientierter Software.
XML	‚EXtensible Markup Language' ist eine Teilmenge der SGML (Standard Generalized Markup Language). XML ist eine Sprache, die es erlaubt, Informationen über die Struktur eines Dokumentes zwischen unterschiedlichen Systemen auszutauschen [IntW3C].
XP	‚EXtreme Programming': Eine pragmatische Methodologie für kleine bis mittlere Teams von Programmierern, die Software angesichts vager oder schnell veränderlicher Anforderungen entwickeln müssen.

10 Literaturverzeichnis und eingesetzte sonstige Hilfsmittel

[Bin2000] R. W. Binder: Testing Objectoriented Systems; Addison-Wesley, Boston; 2000

[BokDah1998] B. Bokowski, M. Dahm: Poor Man's Genericity for Java; Proceedings of JIT 1998; Springer Verlag, Berlin; 1998

[BokSpi1998] B. Bokowski, André Spiegel: Barat – A Front-End for Java. Technical Report B-98-09; Freie Universität Berlin, FB Mathematik und Informatik, Institut für Informatik; Dezember 1998

[ClaSchw1993] V. Claus, A. Schwill: Duden Informatik; Dudenverlag, Mannheim, Leipzig, Wien, Zürich; 1993

[ChiMil1994] J.J. Chilenski, S.P. Miller: Applicability of modified condition/decision coverage to software testing; Software Engineering Journal; September 1994

[DaDiHo1972] Dahl O.J, Dijskstra E.W., and Hoare C.A.R.: Structured Programming; Academic Press; 1972

[Gamma1995] E. Gamma, R. Helm, R. Johnson, J.Vlissides: Design patterns – Elements of Reusable Object-Oriented Software; Addison Wesley; Boston; 1995

[Güt1992] R. H. Güting: Datenstrukturen und Algorithmen; B. G. Teubner Verlag; Stuttgart; 1992

[GütErw1999] R. H. Güting, M. Erwig: Übersetzerbau; Springer Verlag; Berlin; 1999

[JScope1997] Sun Microsystems: Javascope User Guide; SUN; 1997

[Lig1990] P. Liggesmeyer: Modultest und Modulverifikation; BI-Wissenschaftsverlag; Mannheim; 1990

[Lig2001] P. Liggesmeyer: Bedingungsüberdeckungstechniken: Vergleich, Bewertung und Anwendung in der Praxis; Vortrag 17. Treffen der TAV der Gesellschaft für Informatik; Erlangen; Oktober 2001; Kurzfassung unter http://www.fbe.hs-bremen.de/spillner/TAV17/Liggesmeyer.pdf

[PagSix1994] B.-U. Pagel, H.-W. Six: Software Engineering I; Addison Wesley; Bonn; 1994

[Schlag1994] G. Schlageter: Datenbanksysteme; Fernuniversität Hagen; 1994

[SneWin2002] H.M. Sneed, M.Winter: Testen objektorientierter Software; Hanser Verlag; 2002

[WiMau1997] R. Wilhelm, M. Maurer: Übersetzerbau; Springer Verlag; Berlin; 1997

[WinSix2001] M. Winter, H.-W. Six: Objektorientierte Softwareentwicklung; Fernuniversität Hagen; 2001

zum Entwurf und zur Implementation benutzte Literatur:

[Darwin2002] I.F. Darwin: Java Kochbuch; O'Reilly Verlag; 2. Aufl.;2002

[Eckel2000] B. Eckel: Thinking Java; Prentice Hall Verlag; 2nd Edition; 2000

[EcLoWo1999] R. Eckstein, M. Loy; D. Wood: Java Swing; O'Reilly; 1999

[Eddy1997] S.E. Eddy: HTML in Plain English; MIS Press; New York; 1997

[Flanag2000] D. Flanagan: Java in a Nutshell; O'Reilly Verlag; 3. Aufl.;2000

[SixWin2002] H.-W. Six, M. Winter: Grundkonzepte der objektorientierten Softwareentwicklung; Fernuniversität Hagen; 2002

zum Entwurf und zur Implementation benutzte Werkzeuge:

Computer Associates: ErWin 4.1, ER-Datenmodellierungssoftware

TogetherSoft Corporation: Together 5.5, Designwerkzeug und Java-Entwicklungsumgebung

SUN: Forte 4.0, Java-Entwicklungsumgebung

11 Internet-Links

[IntAnt] http://jakarta.apache.org/ant

[IntBcel] http://sourceforge.net/project/bcel

[IntBarat] http://sourceforge.net/project/barat

[IntCort] http://www.thecortex.net/clover/index.html

[IntGJ] http://www.cis.unisa.edu.au/~pizza/gj/Distribution

[IntHoare] T.Hoare: Der neue Turmbau zu Babel; Rede zur Verleihung des Turing-Preises der Association for Computing Machinery, 1980, http://iug.uni-paderborn.de/iug/projekte/kik/dokumente/hoare.html

[IntMccl] http://www.glenmccl.com

[IntMySQL] http://www.mysql.com

[IntOpti] http://www.borland.com/optimizeit

[IntKoa] http://www.koalog.com

[IntJC] http://www.mmsindia.com/JCover.html

[IntJun] http://sourceforge.net/project/junit

[IntHan] http://sourceforge.net/project/hansel

[IntJIE] http://www.forum2.org/eran/jie/

[IntW3C] The XML 1.0 Recommendation, http://www.w3.org/TR/1998/REC-xml-19980210 , 1998

12 Anhang

12.1 Installation

Die Installation des ‚DoiT'-Paketes ist relativ einfach. Man kopiert von der Installations- CD den Pfad ‚doithome' in ein Verzeichnis im Dateisystem des Zielrechners. Auf dem Zielrechner muss ein Java JDK der Version 1.2 oder höher installiert sein. Die von Java ausgewertete Umgebungsvariable ‚CLASSPATH' muss das ‚DoiT' Installationsverzeichnis enthalten, damit die ‚DoiT'-Klassen gefunden werden. Der in ‚doithome' angesiedelte Parser Barat benötigt ebenfalls zwei Einträge in CLASSPATH, nämlich ‚doithome/Barat-1.6.0' und ‚doithome/Barat-1.6.0/bcel.jar'.

Auf dem Installationsrechner oder im zugänglichen Netzwerk muss sich ein Datenbankmanagementsystem befinden in dem eine ‚DoiT'-Datenbank angelegt worden ist. Außerdem wird in diesem RDBMS eine Benutzer ID benötigt, die neben dem Recht Daten einzufügen und zu löschen, auch das Recht benötigt, Datenbanken anzulegen und eigene Datenbanken zu löschen. Solche Rechte-Bündel werden meist als DBA-Rollen bezeichnet. Bei einer Datenbank im Netz ist sicherzustellen, dass sich die für dieses System erforderlichen JDBC-Treiber Klassen im Zugriff des ‚DoiT' Hostes bzw. im CLASSPATH befinden. Auf der Installations-CD findet sich dazu das Datenbankmanagementsystem ‚MySQL' in der Version 3.23 für Microsoft-Windows nebst JDBC Treiberklassen (unterhalb von org/..). Unter dem ‚doithome'-Verzeichnis sind - wie oben schon erwähnt - außerdem der Parser ‚Barat' in der Version 1.6 und die Bytecode-Engineering Library ‚BCEL' abgelegt. Weitere Hinweise zur Installation von Barat und MySQL finden sich auf der Programm-CD in den entsprechenden Verzeichnissen.

Im Installationsverzeichnis müssen sich außerdem die Dateien ‚doit.ini' und ‚DB_props.ini' befinden. Vor dem ersten Start von ‚DoiT' sollten in diesen Dateien die Einstellungen für Datenbank und Verzeichnisse überprüft werden. Dies kann aber auch nach dem Programmstart mit Hilfe der ‚DoiT'-GUI geschehen.

Gestartet wird das Programm aus dem ‚doithome'-Verzeichnis mittels ‚java doitgui.DoitGraph', unter der Voraussetzung, dass ‚java' im Suchpfad des Host-Systems enthalten ist.

12.2 Hinweise zur Weiterentwicklung von ‚DoiT'

In diesem Abschnitt soll kurz darauf eingegangen werden, welche Umgebung zur Weiterentwicklung von ‚DoiT' erforderlich ist und wie Ergänzungen in ‚DoiT' eingebaut werden können.

12.2.1 Entwicklungsumgebung

‚DoiT' ist weitestgehend mit der Entwicklungsumgebung ‚Forte 4J' von SUN (jetzt ‚One') entwickelt worden. Der Quelltext von ‚DoiT' – insbesondere der GUI – enthält daher zahlreiche Pragmas aus der ‚Forte'-Umgebung. Da diese aber in ganz normalen Java-Kommentaren stehen, behindert dies nicht die Arbeit an ‚DoiT' mit anderen Entwicklungsumgebungen.

12.2.2 Java-Grammatik

Zukünftige Änderungen/Erweiterungen an der Java-Grammatik müssen über Barat implementiert werden. Die derzeitige Version 1.6 unterstützt die Java-Grammatik 1.4. Für eine Erweiterung von Barat in Eigenregie wird auf [BoSpie1998] bzw. [IntBarat] verwiesen.

12.2.3 Erweiterung von ‚DoiT'

Zur Erweiterung von ‚DoiT' um weitere Überdeckungsmetriken und weitere Statementklassen, müssen der ‚InstrumentingVisitor' und der ‚MeasuringVisitor' entsprechend geändert und/oder erweitert werden. Dazu ist im ‚InstrumentingVisitor' eine Methode ‚visit*NeuesStatement'* bzw. ‚instr*NeuesStatement'* nach dem Muster der bereits vorhandenen Methoden einzubauen. Ebenso ist dies im ‚MeasuringVisitor' vorzunehmen, um den neuen Instrumentierungspunkt entsprechend auswerten zu können. Außerdem ist in der ‚Test_db'-Klasse die Methode ‚getCov(..)' so zu erweitern, dass die Kennzahl für die neue Metrik korrekt berechnet wird. Für die neue Statement-Klasse bzw. für den Instrumentierungspunkt muss dazu außerdem ein geeignetes Kürzel für die Repräsentation in der Datenbank vergeben werden. Zur Zeit sind beispielsweise 'class', 'constructor', 'method-start', 'method-end', 'while-Rumpf', 'try-catch', 'if-then', 'else', 'for-Rumpf', 'initializer-start', 'initializer-end', 'do-while-Rumpf', 'switch', 'case' und 'default' vorhanden. Schließlich ist auch 'barat_mess' noch um einen entsprechenden Methodenaufruf an ‚Test_db.getCov(..)' zu erweitern, um die neue Metrik auszugeben. Das Einfügen der Instrumentierungsanweisungen in den Prüfling erfolgt, wie bei allen anderen Statementklassen auch, immer mittels ‚Test_db.insertInstr(..)' bzw. per ‚println("Test_db.updateInstr(....);");' (siehe hierzu die bereits vorhandenen Methoden im ‚InstrumentingVisitor' und die dortigen Kommentare).

12.2.4 Weitere Datenbankmanagementsysteme

Da ‚DoiT' zur Zeit MySQL und Oracle als Datenbankmanagementsysteme unterstützt, sind viele der SQL-Statements in ‚Test_db' zweifach vorhanden bzw. stehen innerhalb von Auswahlanweisungen hinsichtlich des DB-Produktes. Durch die Beispiele MySQL und Oracle ist es einfach, weitere RDBMS für ‚DoiT' anzupassen.

Falls ‚DoiT' für weitere Systeme angepasst werden soll, ist zunächst in ‚Test_db.open()' die richtige JDBC-Treiber-Klasse für das neue RDBMS anzugeben bzw. die Datei ‚DB_props.ini' passend zu füllen. Eventuell sollte auch die Klasse ‚doitgui.dboptions' so angepasst werden, dass die Datei ‚DB_props.ini' mit Hilfe der GUI gepflegt werden kann. Dann ist der SQL-Dialekt des neuen Systems auf Inkompatibilitäten zu der vorhandenen Syntax zu prüfen. Falls solche bestehen, muss nach semantisch äquivalenten SQL-Konstrukten im neuen System gesucht werden. Dies dürfte am ehesten im Bereich der SQL-Funktionen (Datum/Zeit, Null-Test, etc.) aber auch im strukturellen Bereich (Datentypen) zu erwarten sein. Letzteres wird einfach durch Austausch der entsprechenden SQL-Skripts im Verzeichnis ‚doitgui/sql' bewerkstelligt (siehe 6.1).

12.3 Benutzerhandbuch

1. Programmzweck

Zweck des Programmpaketes ‚DoiT' ist es, Java Programme automatisch mit Instrumentierungsanweisungen auszustatten und mit deren Hilfe während des Programmlaufes diverse Kennzahlen zu Überdeckungsmetriken zu erhalten. Die Ergebnisse werden in einer relationalen Datenbank abgelegt und können so jederzeit zusammen mit dem Programmquelltext dargestellt werden. Zur Zeit läuft ‚DoiT' gegen das RDBMS MySQL - wegen der Verwendung von Standard JDBC sind jedoch auch andere Datenbanken anschließbar. Erfolgreich getestet wurde neben MySQL bisher Oracle 8.1.6.

2. Aufbau

‚DoiT' besteht zurzeit im Wesentlichen aus sechs Teilen:

-dem Parser Paket ‚Barat'
-der Byte-Code-Engineering-Library ‚BCEL'
-dem Datenbankmanagementsystem ‚MySQL' (wahlweise auch ‚Oracle')
-der GUI ‚DoitGraph'

-dem Instrumentierer ‚barat_test'
-dem Evaluierer ‚barat_mess'

Plattform von ‚DoiT' ist das Parserpaket Barat, entwickelt von Boris Bokowski und André Spiegel an der FU Berlin. Zur Funktion von Barat ist außerdem die von Markus Dahm entwickelte Byte-Code-Engineering Library ‚BCEL' erforderlich. Beide Softwarepakete sind über [IntBarat] bzw. [Intbcel] zu beziehen.

Der Instrumentierer ‚barat_test' ist ebenso wie der Evaluierer ‚barat_mess' ein Kommandozeilen Werkzeug, dass direkt vom Betriebssystem-Prompt mit geeigneten Parametern Java-Code instrumentieren bzw. Ergebnisse instrumentierter Testläufe anzeigen kann. Dadurch ist der Einsatz des Instrumentierers auch in Batch-Anwendungen möglich.

Die GUI ‚DoitGraph' verkapselt dieses eher ‚spröde' Benutzer-Interface in einer komfortablen Benutzeroberfläche. Dabei werden entweder die zu instrumentierenden Dateien bzw. Java Packages oder die bereits durchgeführten Testläufe aus der Datenbank in einer baumartigen Struktur zur Auswahl bereitgestellt.

3. Bedienung

Es folgt eine knappe Einführung in die Bedienung des Programms. Die Screenshots in dieser Anleitung wurden auf einem Microsoft Windows PC erstellt, daher orientiert sich das dargestellte ‚Look-and-Feel' automatisch an den Microsoft-Konventionen. In einer X11 Umgebung oder auf einem MacIntosh PC würde das Programm sich automatisch mit dem entsprechenden Aussehen präsentieren. An der Funktionalität ändert sich dadurch natürlich nichts.

3.1 Instrumentierung

Zur Instrumentierung von Quelltexten gibt es zwei Möglichkeiten. Entweder man wählt aus dem Menü <Datei> den Eintrag <Quelltext-Datei öffnen> oder <Quelltext-Paket öffnen>. Im ersteren Fall erhält man ein Datei-Auswahl-Fenster mit dessen Hilfe sich ein oder mehrere Java-Dateien auswählen lassen. Die Auswahl ist dabei immer auf ein Verzeichnis bzw. ein Java-Paket beschränkt.

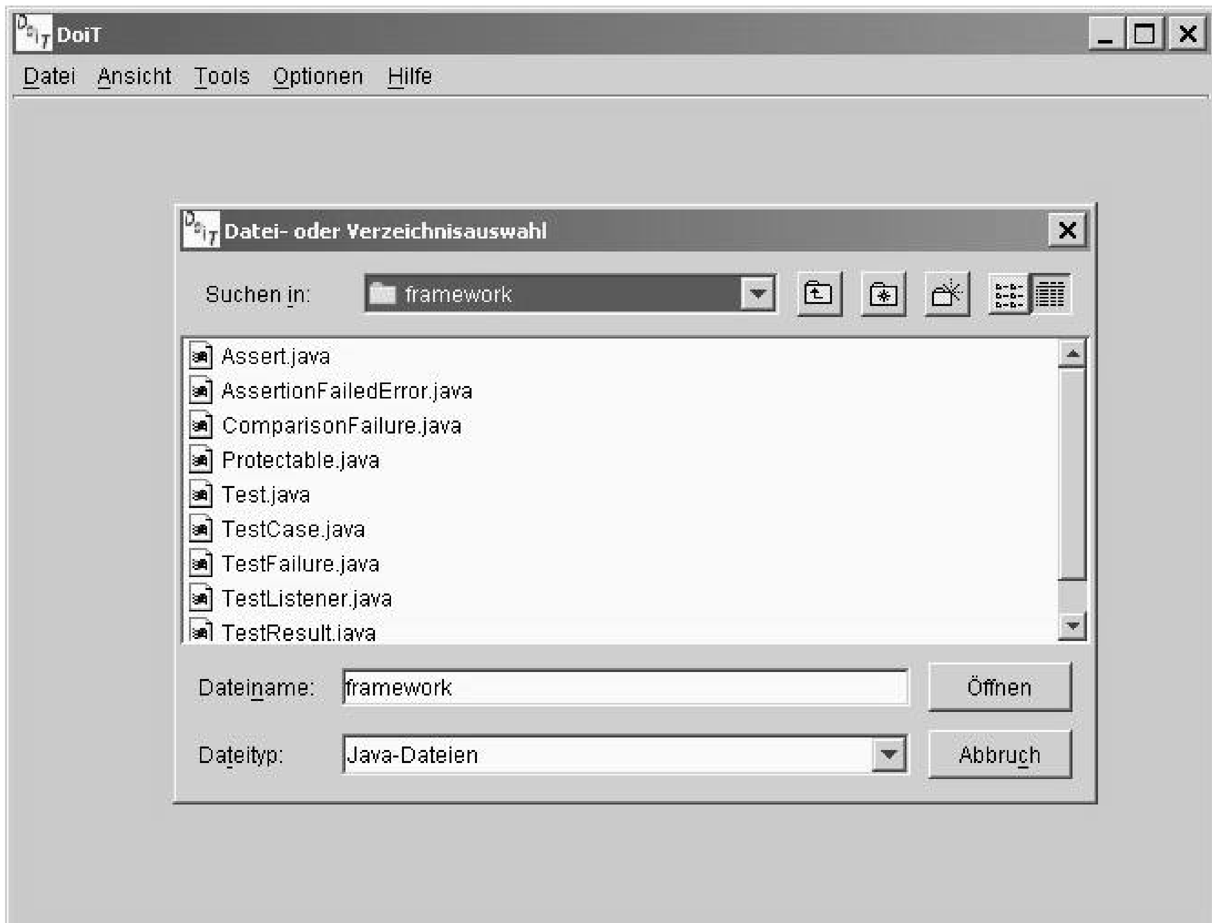

Abbildung 19: Auswahl der Quelldateien

Die Menü-Option <Quelltext-Paket-öffnen> bietet sich an, wenn ein ganzes Paket inklusive Sub-Paketen instrumentiert werden soll. Der Name des zu instrumentierenden Paketes muss dabei in einem Dialogfenster eingetragen werden.

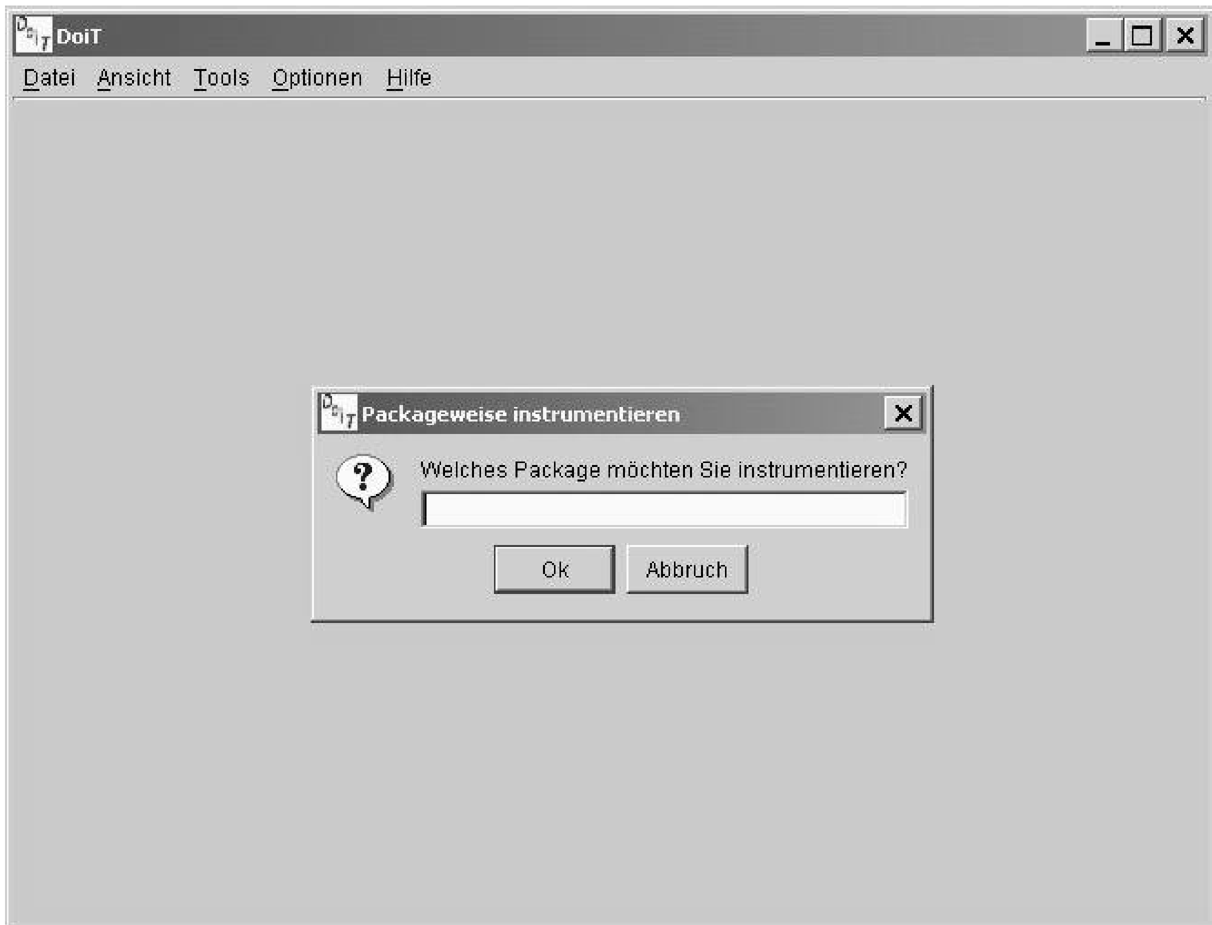

Abbildung 20: Paketweises Instrumentieren

In beiden Fällen wird danach eine Baum-Struktur dargestellt, an dessen Wurzel sich das jeweilige Paket befindet. Waren die Quelldateien keinem Paket zugeordnet, so wird hier ‚default' als Paket-Name angezeigt. Die nächste Ebene des Baumes zeigt die zugehörigen bzw. zur Instrumentierung ausgewählten Dateien. Klickt man mit dem Mauszeiger auf diese Datei-Einträge, werden in der nächsten Ebene die enthaltenen Klassen angezeigt.

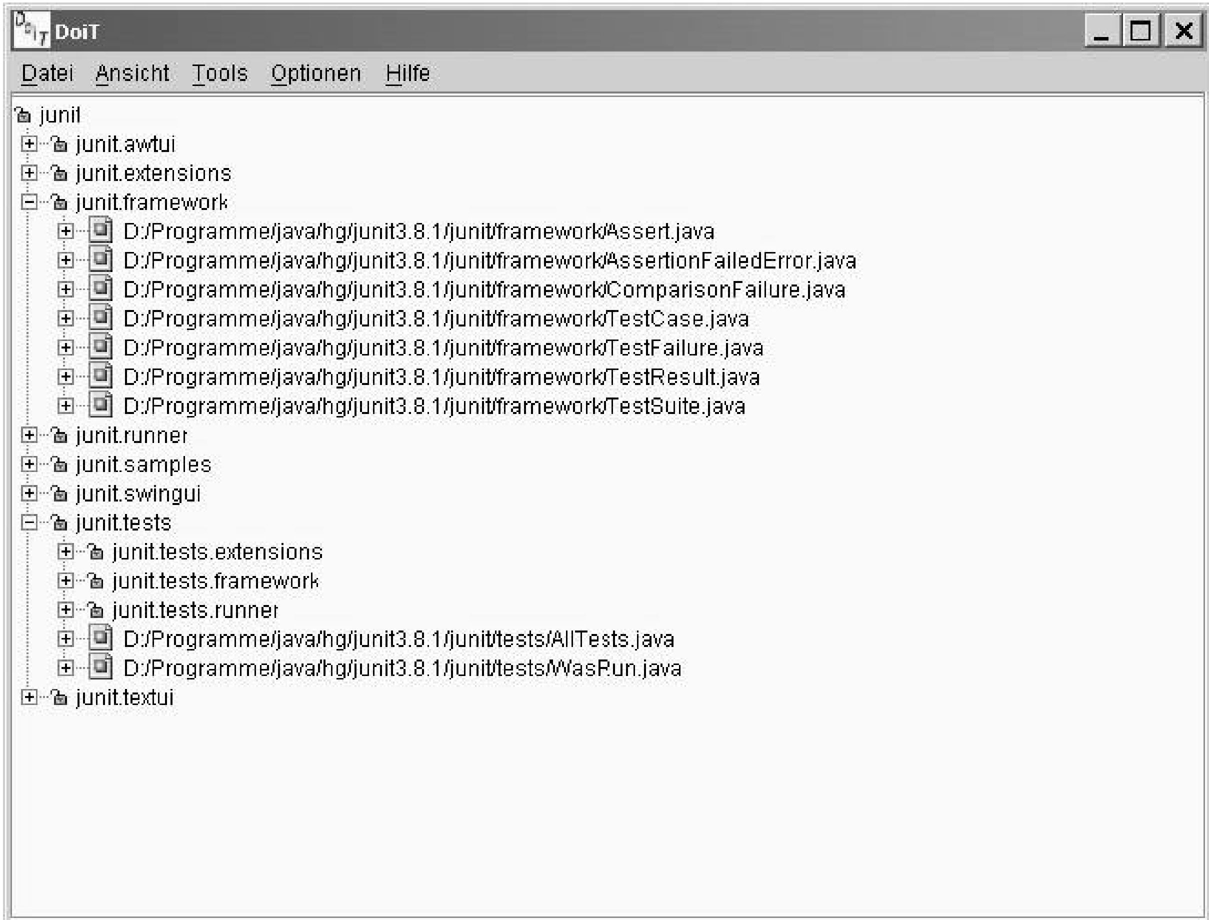

Abbildung 21: Zur Auswahl beim datei-/klassenweisen Instrumentieren

Aus dieser Darstellung ergeben sich drei mögliche Arten der Instrumentierung: paket- , datei- oder klassenorientiert. Es kann jeweils nur ein Verfahren gewählt werden; aus der Baum-Struktur können somit durch Klicken mit der Maus entweder das Paket, Datei(en) oder Klasse(en) gewählt werden (Bei Mehrfachselektion muss dazu wie allgemein üblich die Umschalttaste gedrückt werden).

Ist die Auswahl der zu instrumentierenden Programmeinheiten abgeschlossen, wird entweder aus dem Menü <Tools> <Start Instrumentierung> gewählt oder durch Drücken der rechten Maustaste ein Kontextmenu geöffnet, aus dem sich ebenfalls <Start Instrumentierung> wählen lässt. Daraufhin wird der Instrumentierer ‚barat_test' gestartet, der die Quelldateien ‚in situ', das heißt an ihrem Originalstandort, instrumentiert. Die unveränderten Originaldateien werden im Pfad ‚doit.savedir' bzw. ‚java.io.tmpdir' abgelegt. Die angegeben Variablen aus den System-Properties der JVM können beim Start von Java per Option ‚-D' übergeben werden. Bleibt der Parameter ‚doit.savedir' ungesetzt, wird dafür der Standard-Parameter ‚java.io.tmpdir' genommen (verweist meist auf $TMP bzw. %TEMP%). Unterhalb dieses Verzeichnisses - das

existieren muss! - wird folgende Struktur eingehalten: „../DBName/Paketname/Nr. der Instrumentierung/Dateiname“. Sollte aus einer früheren Instrumentierung hier bereits eine Quelldatei mit gleichem Namen existieren, wird diese nicht überschrieben! Dies kann wegen der bei jeder Instrumentierung inkrementierten Instrumentierungsnummer aber nur dann geschehen, wenn die Datenbank zwischenzeitlich gelöscht worden ist.

Die laufende Nummer der Instrumentierung ist immer paketbezogen und wird in der ‚DoiT'-Datenbank gepflegt. Während der Instrumentierung gibt ‚barat_test' seine Ausgaben auf der ‚DoiT'-Konsole über den Bildschirm aus. Hier wird auch die aktuell vergebene Instrumentierungs-Nr. zusammen mit weiteren Hinweisen genannt. Außerdem werden die Anweisungen aufgelistet, die in den Prüfling an geeigneter Stelle manuell eingefügt werden müssen, um die Testumgebung zu initialisieren.

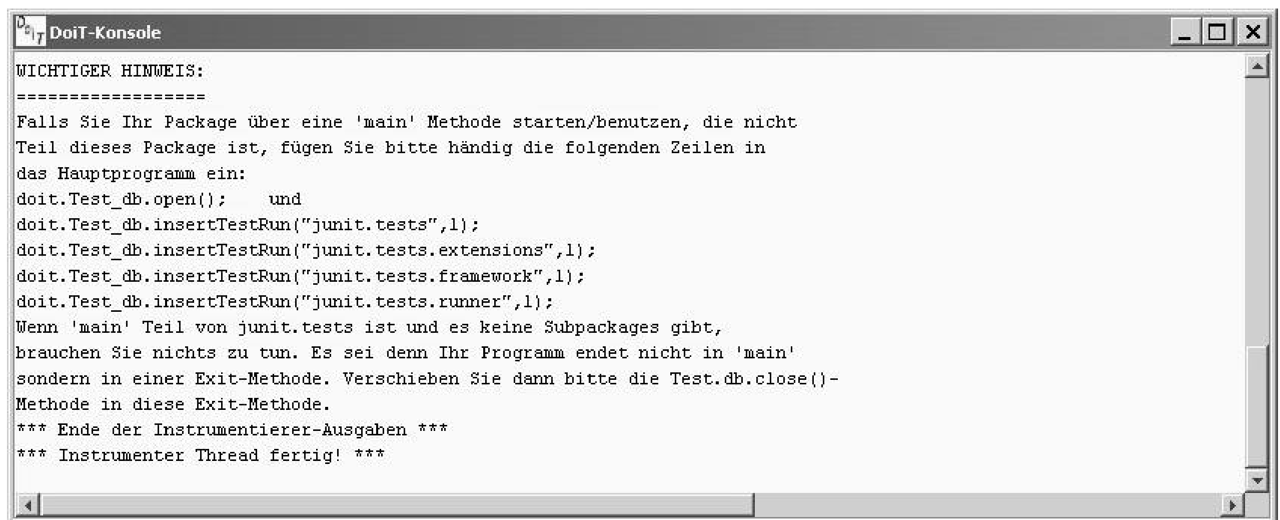

Abbildung 22: Beispielausgabe der ‚DoiT'-Konsole

Vor einem Testlauf müssen die instrumentierten Prüflinge natürlich zusammen mit ihren zugehörigen Bibliotheken neu kompiliert werden. Bei jedem Testlauf, der mit dem Prüfling danach ausgeführt wird, werden die Ergebnisse der Überdeckungsmessungen in die Datenbank geschrieben. Dabei kann jeder Testlauf mit einem separaten Datensatz (= einer fortlaufenden Testlauf-Nr.) abgelegt werden, oder es können mehrere Testläufe unter eine Testlauf-Nr. akkumuliert werden. Dies hängt davon ab, ob in den Prüfling die Anweisung ‚insertTestRun' eingefügt wurde oder nicht.

3.2 Auswertung

Bei Auswahl dieses Menüeintrages werden aus der Datenbank die bereits durchgeführten Testläufe und die bereits instrumentierten Dateien selektiert und analog zur Darstellung beim In-

strumentieren angezeigt. Die Wurzel wird nun allerdings vom Datenbank-Namen gebildet, gefolgt von Paket-, Datei- und Klassennamen. An den jeweiligen Paketnamen werden per ‚_' die lfd. Nr. der Instrumentierung und die lfd. Nr. des Testlaufes angehängt, dabei steht Testlauf Nr. 0 für noch ungetestete aber bereits instrumentierte Programmeinheiten.

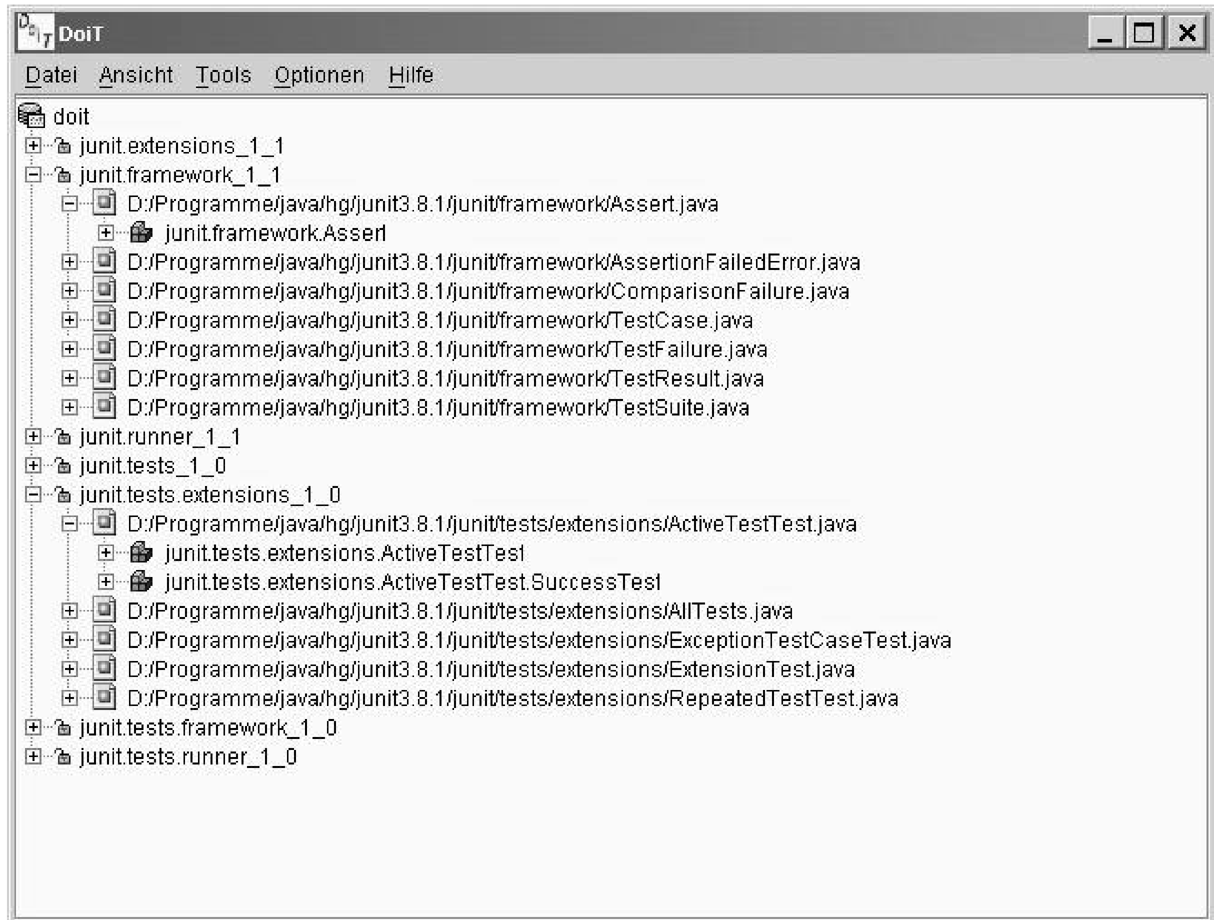

Abbildung 23: Auswahl der zu evaluierenden Testläufe

Hier kann z.Z. jeweils eine Programmeinheit (Paket, Datei oder Klasse) ausgewählt werden. Aus dem Menü <Tools> oder alternativ aus einem Kontextmenu kann ‚barat_mess' gestartet werden. Die Ausgabe dieses Programms erfolgt dann entweder als kompilierbarer Java-Quelltext, der die Originaldatei zusammen mit den Messwerten in Kommentarzeichen enthält, oder als HTML Text, der die Messwerte auch farblich veranschaulicht (rot = nicht durchlaufener Code, blau = Messwerte, schwarz = ausgeführter Code). Die Einstellung HTML ‚ja' oder ‚nein' kann im Menü <Optionen> vorgewählt werden. Das Messergebnis kann auf Wunsch aus dem Darstellungsfenster mittels der Schaltfläche <Speichern> abgespeichert werden. Die Datenbank enthält darüber hinaus noch weitere Informationen, bspw. hinsichtlich der durchlaufenen Code-Sequenz.

```
Ausgabe im HTML-Format

// Package-Name: junit.runner, Version: 3.8.1, Instrumentierung Nr. 1, instrumentiert
// Ergebnisse des instrumentierten Testlaufes vom 31.12.2002 18:57
// lfd. Test-Nr. dieser Inst-Nr: 1
// Bemerkung zur Instrumentierung: Testen von DoiT mit JUnit
//
// Globale Messergebnisse für Package junit.runner :
// Klassenüberdeckung                         : 66 %
// Methodenüberdeckung                        : 53 %
// Zweigüberdeckung                           : 43 %
// Bedingungsüberdeckung                      : 42 %
// Minimale Mehrfachbedingungsüberdeckung     : 42 %
// Schleifenüberdeckung                       : 75 %
// Ausnahmeüberdeckung                        : 19 %
//
//
//
// File-Name: D:/Programme/java/hg/junit3.8.1/junit/runner/BaseTestRunner.java, Versi
// Ergebnisse des instrumentierten Testlaufes vom 31.12.2002 18:57
// lfd. Test-Nr. dieser Inst-Nr: 1
// Bemerkung zur Instrumentierung: paketweise Instrumentierung
//

Speichern                                                                    Weiter
```

Abbildung 24: Beispiel für Test-Evaluierung mit 'barat_mess'

Im gleichen Kontext ist auch die De-Instrumentalisierung von Dateien implementiert. Bei der Anwahl eines Pakets, eines Files oder einer Klasse und nachfolgendem Start von <Instrumentierung aufheben> aus dem <Tools> oder Popup-Menü, wird die gesicherte Originaldatei bzw. die gesicherten Originaldateien wieder an ihren ursprünglichen Platz kopiert und die instrumentierte Datei dabei überschrieben. Der Prüfling muss danach natürlich neu kompiliert werden.

3.3 Einstellungen

Im Menü <Optionen> befinden sich einige Programmeinstellungen. Über die Schalter <Bedingungsüberdeckungsmessung>, <minimale Mehrfachbedingungsüberdeckung> und <Ausgabe als HTML Text> lassen sich diese Funktionen ein- oder ausschalten. Die ersten beiden Optionen wirken auf Instrumentierungsläufe. Wenn <Bedingungsüberdeckungsmessung> eingeschaltet ist, werden aggregierte logische Ausdrücke, die den Kontrollfluss eines Programms beeinflussen, in atomare Ausdrücke zerlegt. Diese werden dann zur Laufzeit einzeln bewertet, um einen Messwert für die Bedingungsüberdeckung (‚C_2-Überdeckung') zu erhalten. Ist die

<minimale Bedingungsüberdeckung> eingeschaltet, werden zusätzlich alle bei der Zerlegung auftretenden Teilausdrücke bewertet.

Die Funktion des Schalters <Ausgabe als HTML Text> wurde im Abschnitt 3.2 bereits erläutert.

Über <DoiT Pfad-Optionen> kann der als ‚doit.savedir' an den Instrumentierer übergebene Pfad eingestellt werden (Default ist ‚java.io.tmpdir', also $TEMP oder %TEMP%). Im gleichen Dialog kann der Pfad in dem die Programmdateien gesucht werden, verändert werden. Voreinstellung ist hier ‚user.dir' also das aktuelle Arbeitsverzeichnis.

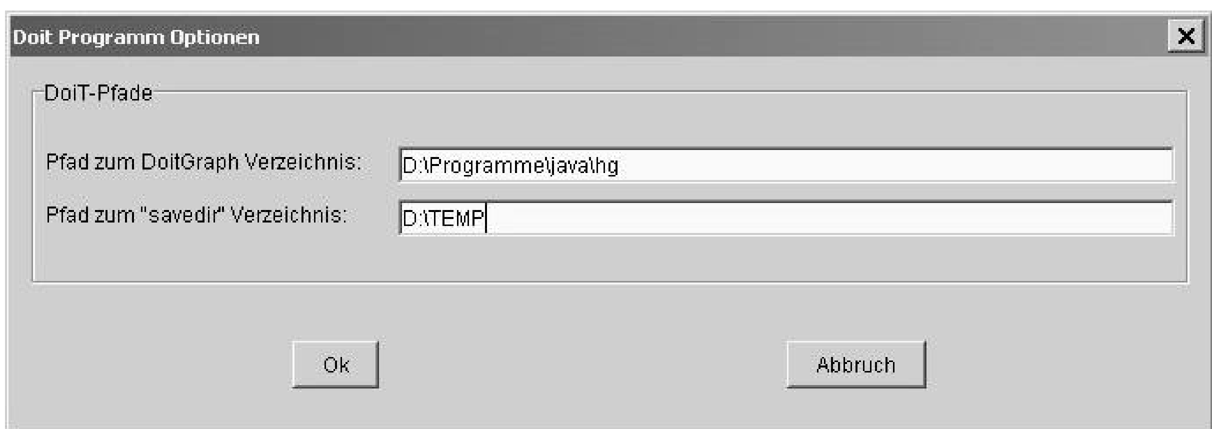

Abbildung 25: Einstellung der ‚DoiT' Datei-Pfade

3.4 Datenbankverwaltung

Unter <Datenbankoptionen> verbirgt sich ein Dialog, der sich aus der Datei ‚DB_props.ini' die Eigenschaften der Datenbanken besorgt und diese per ‚DB_props.ini' an ‚barat_test' und ‚barat_mess' weitergibt. Da hier alle RDBMS-relevanten Parameter hinterlegt sind, kann das Datenbanksystem gegen ein beliebiges anderes System ausgetauscht werden. Vorraussetzung ist, dass eine entsprechende JDBC Treiberklasse verfügbar ist. Getestet wurden bisher MySQL und Oracle 8.1.6. Außerdem kann mit Hilfe dieses Dialogs auch zeitweise auf eine andere Datenbank, auf andere Rechner oder andere Benutzer ‚umgeschaltet' werden. Für die Verwendung von Oracle 8.1.6 müssen SQL-Skripts ausgetauscht werden. Die Scripts befinden sich im Verzeichnis „../doitgui/sql/*RDBMS-Name*".

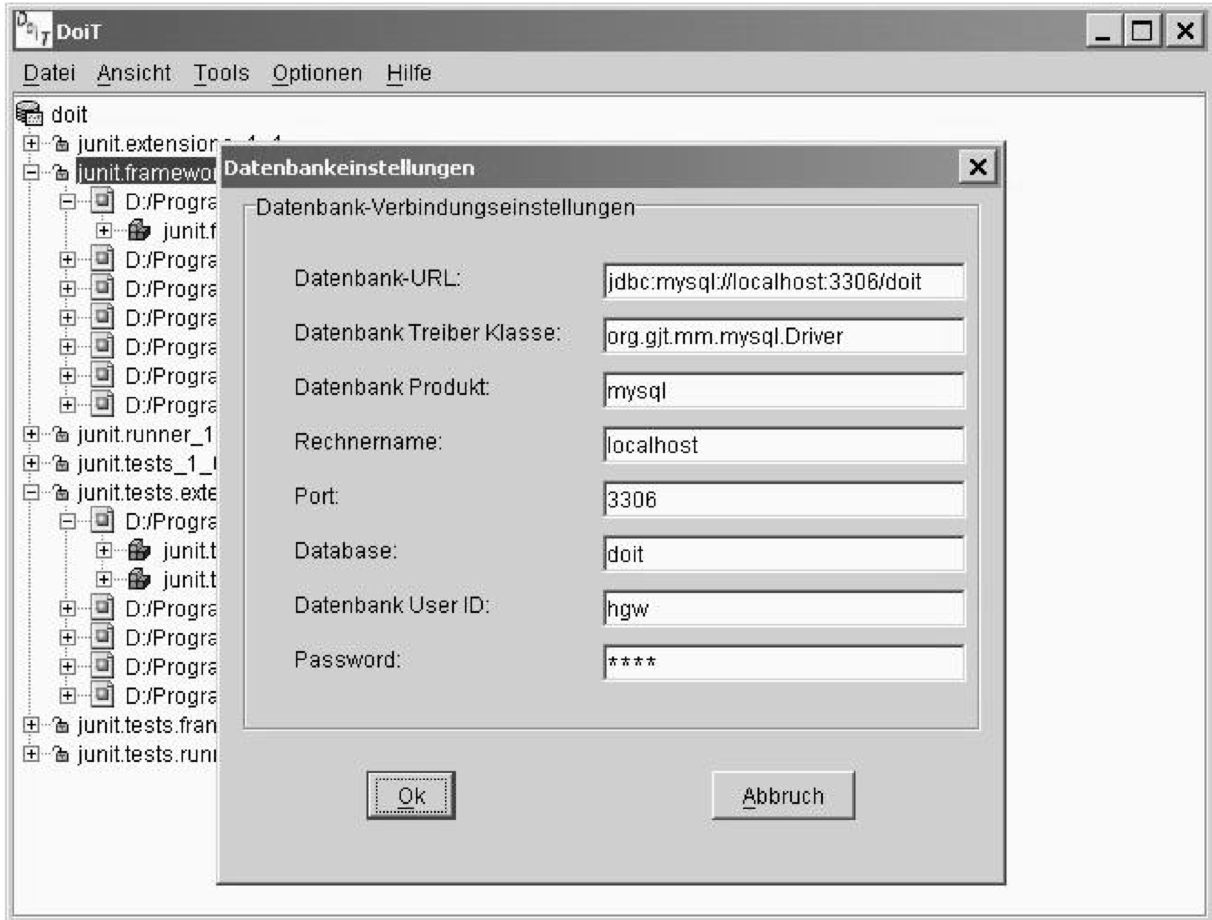

Abbildung 26: Dialog zur Datenbank-Anbindung

Mit Hilfe des letzten Menüpunktes <Datenbank anlegen und löschen> wird ein neues Datenbankschema erzeugt, Tabelleninhalte gelöscht, gesichert oder geladen.

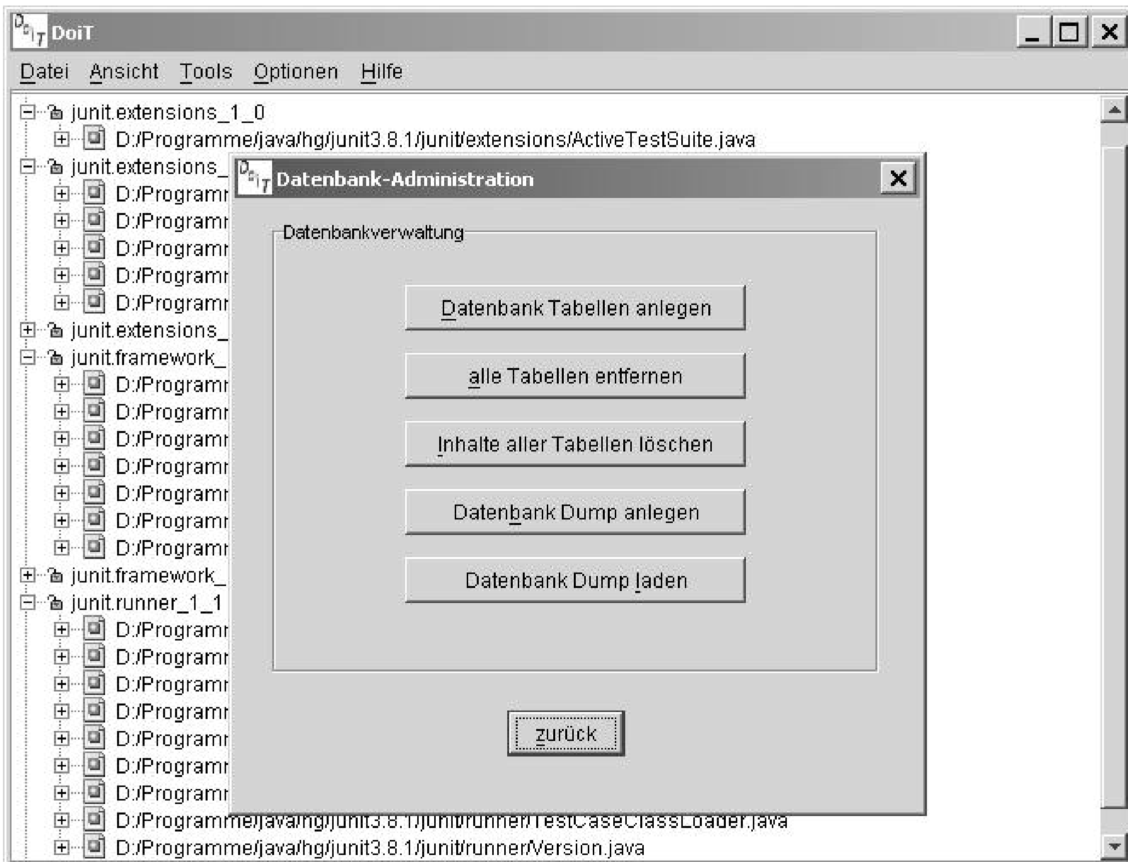

Abbildung 27: Dialog zur Datenbank-Verwaltung

Die Datenbank-Operationen erfolgen auf der Basis von austauschbaren SQL-Skripts, da diese Operationen im Gegensatz zu reinen SQL-Abfragen (SELECT ...) sich wenig an Standards halten. Wenn das Datenbankprodukt sich ändert, müssen evtl. auch diese Skripts angepasst werden. Sie finden sich im Unterverzeichnis ‚sql' im ‚DoiT'-Programm-Verzeichnis als ‚crea.sql', ‚delete.sql', ‚drop.sql', ‚dump.sql', ‚load.sql'. Der Datenexport erfolgt unter MySQL in einem CSV-ähnlichen Format, das leicht mit anderen Programmen weiterverarbeitet werden kann. Da Oracle zum Datenexport und –import spezielle Werkzeuge (‚imp', ‚exp') verwendet, funktioniert der Import/Export nur unter MySQL.

12.4 Einsatzmöglichkeiten der Batch-Version

Im Batch-Modus kann der Instrumentierer per

```
java [-Ddoit.savedir=<SaveDirPath>] \
barat_test [-CC|-MC] [-c|-f | class|file version bemerkung | @steuerdatei] | \
[-p version bemerkung]
```

gestartet werden. Über den Java-Properties Parameter ‚doit.savedir' kann der Basispfad für die Sicherung der Original-Quelltextdateien übergeben werden. Wird hier nichts angegeben, wird der Temp-Pfad des Betriebssystems, also bspw. $TMP oder %TEMP% genommen.

Der ‚barat_test' Schalter ‚-CC' schaltet die Instrumentierung zum Erfassen von Daten zur Bedingungsüberdeckung ein. Der Schalter ‚-MC' schaltet die minimale Mehrfachüberdeckungsanalyse ein. Die Schalter ‚-c' und ‚–f' schalten entweder die klassen- oder dateiorientierte Instrumentierung ein. Voreinstellung ist die dateiorientierte Instrumentierung. In beiden Fällen wird danach der Name der Klasse/der Datei und eine Kennung für die Version der Klasse/der Datei erwartet und danach ein Bemerkungs-String zur Instrumentierung. Alternativ zu Name/Version/Bemerkung kann auch der Name einer Parametrierungsdatei angegeben werden. Dieser Name muss durch das Voranstellen eines ‚@' als Dateiname kenntlich gemacht werden. Das Format dieser Steuerdatei ist denkbar einfach:

(Zeile Nr-1) mod 3 = 0: Name der Klasse/Datei
(Zeile Nr-1) mod 3 = 1: Version der Klasse/Datei
(Zeile Nr-1) mod 3 = 2: Bemerkung zur Klasse/Datei

Zur paketorientierten Instrumentierung wird der Schalter ‚-p' angegeben. Danach erfolgt ebenfalls die Angabe von Paketname/Version/Bemerkung.

Beim Ausführen von ‚barat_test' innerhalb eines Batch-Jobs ist zu beachten, dass die Datei ‚DB_props.ini' gültige Einträge für das Datenbank-Managementsystem enthält. Das Format dieser Datei ist wie folgt:

```
#Parameter für den Datenbankconnect (mysql)
#Thu Oct 31 13:46:08 CET 2002
dburl=jdbc\:mysql\://localhost\:3306/doit
user=hgw
port=3306
dbproduct=mysql
password=test
host=localhost
database=doit
driver=org.gjt.mm.mysql.Driver
```

Abbildung 28: Struktur der Datei ‚DB_props.ini'

Zu beachten ist, dass Sonderzeichen wie ‚:' plattformunabhängig mittels ‚\' als solche kenntlich gemacht (‚escaped') werden, wie das allgemein bei UNIX-Systemen üblich ist. Die Datei ‚DB_props.ini' muss sich in dem Verzeichnis befinden, in dem ‚barat_test' ausgeführt wird.

Der Evaluierer/Deinstrumentierer ‚barat_mess' kann im Batch-Modus mittels

```
java [-Ddoit.savedir=<SaveDirPath>] \
barat_mess [-h|-d] [-p|-f|-c] [package|file|class] | inst_nr test_nr
```

gestartet werden. Über den Java-Properties Parameter ‚doit.savedir' wird genau wie bei ‚barat_test' der Basispfad für den Sicherungsort der Original-Quelltextdateien übergeben. Mit dem ‚barat_mess'-Schalter ‚-h' wird eine HTML-Ausgabe eingeschaltet. Dadurch werden durchlaufene bzw. nicht durchlaufene Quelltext-Teile und als falsch, wahr bzw. gar nicht ausgewertete Bedingungen farblich hervorgehoben. Der Schalter ‚-d' dient dazu, ein Paket, eine Datei oder eine Klasse zu de-instrumentieren, d.h. die instrumentierte Version der Datei(en) wird/werden wieder durch das/die Original(e) ersetzt. Die Schalter ‚-h' und ‚-d' können nur alternativ angewandt werden.

Mit den Schaltern ‚-f' und ‚-c' wird die Ausgabe einer Datei bzw. einer Klasse angestoßen. Letztlich wird natürlich immer die Datei (mit der gewählten Klasse darin) ausgegeben. Der Unterschied ist, dass im Falle der klassenorientierten Ausgabe neben klassen- auch dateiumfassende Kennzahlen für Überdeckungsmetriken ausgegeben werden. Bei Angabe des Schalters ‚-p' erfolgt die Ausgabe eines ganzen Paketes, soweit in der Datenbank Messwerte für die Dateien dieses Paketes enthalten sind.

In jedem Fall erfolgt die Ausgabe auf den Standard-Ausgabekanal des Systems, sodass in der Regel eine Ausgabeumlenkung in eine Datei erforderlich ist.

12.5 Datenlexikon der ‚DoiT'-Datenbank

Feld-Name	Datentyp	Erläuterung
A_EXPR	VARCHAR(160)	Teilausdruck eines größeren aggregierten logischen Ausdrucks
C_NAME	VARCHAR(80)	Vollqualifizierter Klassenname
C_VERSION	VARCHAR(10)	Vom Benutzer angegebene Version
CLASS_BEM	VARCHAR(120)	Freie Bemerkung zur Klasse
DATUM	DATETIME	Zeitpunkt zu dem der jeweilige Datensatz angelegt wurde.
EXPR	VARCHAR(160)	Ausdruck aus einer Steueranweisung wie if-then, for, while, do, etc.
EXPR_NR	INTEGER	Lfd. Nr. der Teilausdrücke eines zerlegten logischen Ausdrucks. Negative EXP_NR bezeichnen zusammengesetzte Teilausdrücke, positive bezeichnen atomare nicht weiter zerlegbare Ausdrücke und 0 bezeichnet den Ursprungsausdruck.
F_NAME	VARCHAR(120)	File-Name
F_VERSION	VARCHAR(10)	Vom Benutzer angegebene File-Version
FILE_BEM	VARCHAR(120)	Freie Bemerkung zur Klasse
INST_NR	INTEGER	Lfd. Nr. der Instrumentierung eines Paketes
KONTEXT	VARCHAR(80)	Enthält den Objektkontext eines Methodenaufrufes
M_NAME	VARCHAR(120)	Vollqualifizierter Methodenname
P_NAME	VARCHAR(80)	(Sub-)Package-Name
P_VERSION	VARCHAR(10)	Vom Benutzer angegebene Package Version
PACK_BEM	VARCHAR(120)	Freie Bemerkung zum Package
PROG_ZEILE	INTEGER	Zeilen-Nummer in einer Programmdatei
RUN_TIME	DATETIME	Zeitpunkt des Starts eines Überdeckungstests
SAVE_PATH	VARCHAR(80)	Sicherungspfad einer Datei
SEQ_NR	INTEGER	Sequenz-Nummer eines bestimmten Instrumentierungspunktes (wird jeweils beim ersten Besuch eines Statements gesetzt, aber trotzdem bei jedem Besuch eines Instrumentierungspunktes inkrementiert).

STATEMENT	VARCHAR(80)	Statement-Klasse (if-then, for-Rumpf, while-Rumpf, etc.)
T_DATE	DATETIME	Zeitpunkt des letzten Tests
TEST_NR	INTEGER	Lfd. Ausführungsnummer zu einer INSTR_NR
VALUE	VARCHAR(80)	Zur Laufzeit ermittelter Wert eines Ausdruckes
VISIT	INTEGER	Häufigkeit des Besuches zur Laufzeit