Ein interaktionsbasiertes Modell für den objektorientierten Integrations- und Regressionstest

Mario Winter

FernUniversität — GH Hagen, FB Informatik, Prakt. Inf. III, Feithstr. 142 / IZ, D-58084 Hagen, eMail: Mario.Winter@fernuni-hagen.de

Zusammenfassung. Bei der iterativen, inkrementellen Software-Entwicklung werden Klassen sukzessive spezifiziert, implementiert bzw. geändert oder erweitert, klassengetestet und zu einem ausführbaren Anwendungssystem integriert. Integrations- und Regressionstest stellen dabei besonders wichtige und häufig ausgeführte Testaktivitäten dar, die zudem aufgrund der vielfältigen Interaktionsmöglichkeiten in objektorientierten Programmen sehr komplex sind. Integrations- und Regressionstest werden in der Literatur stets separat behandelt, obwohl sie durchaus ähnliche Teilaktivitäten beinhalten. Bestehende Ansätze zum objektorientierten Integrationstest betrachten entweder strukturelle oder verhaltensorientierte Systembeschreibungen. In diesem Beitrag stellen wir das Klassen-Botschaftsdiagramm (KBD) vor, ein Struktur und Verhalten objektorientierter Programme gleichermaßen berücksichtigendes interaktionsbasiertes (Test-)Modell. Wir skizzieren Algorithmen zur Änderungsanalyse, Ableitung einer Integrationsstrategie und Auswahl von Regressionstestfällen nach der Modifikation bestehender Klassen. Abschließend werden die Ergebnisse einiger mit Smalltalk-80 Klassen durchgeführter Experimente zur Evaluierung des Verfahrens vorgestellt.

Schlüsselwörter. Test objektorientierter Programme, Integrationstest, Regressionstest, Test-Repräsentation, Klassen-Botschaftsdiagramm, Java

Abstract. The highly incremental and iterative development cycle for object-oriented software demands both many more changes and partially implemented resp. re-implemented classes. Much more integration and regression testing has to be done to reach stable stages, which turned out to be considerable more complex than testing conventional software. Despite many similarities, integration and regression testing are handled separately in the literature. Known integration testing techniques concentrate on either structural or behavioural descriptions of the system. In this presentation we propose the Class-Message Diagram (CMD), a diagram capturing all possible interactions in an object-oriented program. Then we sketch algorithms to identify integration resp. regression test strategies and all test cases to be executed after some implementation resp. modification activities. Finally, we summarize some experiments on Smalltalk-80 programs.

Keywords. Object-oriented testing, integration testing, regression testing, Class-Message-Diagram, Java

CR Subject Classification. D2.4, D2.5, D3.3

1 EINFÜHRUNG

Bei der iterativen, inkrementellen Software-Entwicklung werden Klassen sukzessive spezifiziert, implementiert bzw. geändert oder erweitert, klassengetestet und zu einem ausführbaren Anwendungssystem integriert [12]. Eine *Iteration* dauert wenige Tage bis Wochen, in denen neue Klassen dem entstehenden Anwendungssystem hinzugefügt bzw. bestehende geändert und klassengetestet werden. Jede Iteration mündet in einem *Inkrement*,

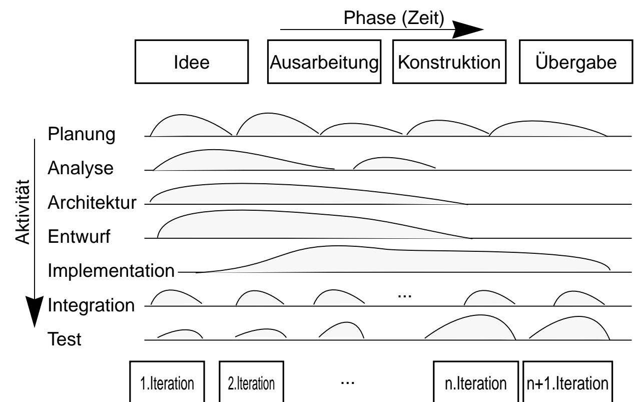

Abb. 1 Iterative, inkrementelle Entwicklung (aus: [12])

welches — zusammen mit den unveränderten Klassen — zu der aktuellen Version des Anwendungssystems zusammengesetzt wird (*build*). Abb. 1 zeigt, dass die entsprechenden Aktivitäten während der Entwicklung ständig ausgeführt werden, wobei ihr Fokus und ihre Intensität je nach Entwicklungsphase unterschiedlich sind.

Bei jeder Iteration wird das Zusammenspiel der Klassen im Integrationstest geprüft. Dabei wird die Funktionalität geänderter oder erweiterter Klassen (oder des gesamten Anwendungssystems) bezüglich der ursprünglichen Spezifikation mit entsprechenden Regressionstests validiert. Integrations- und Regressionstest stellen daher besonders wichtige und häufig ausgeführte Testaktivitäten in der objektorientierten Softwareentwicklung dar, die zudem aufgrund der vielfältigen Interaktionsmöglichkeiten in objektorientierten Programmen sehr komplex sind. Diese Komplexität hat im Wesentlichen drei Ursachen:

1. Klassen und insbesondere Methoden haben im Vergleich zu herkömmlichen Moduln und Prozeduren meistens eine wesentlich feinere Granularität und eine damit verbundene höhere Anzahl (vgl. [20][1]).
2. Dazu kommt die in der Anzahl quadratisch wachsenden Interaktionsmöglichkeiten der Klassen bzw. ihrer Methoden in objektorientierter Software (Benutzungsgraph). Im Vergleich dazu ist die Komplexität modularer Software in der Anzahl der Moduln meistens linear wachsend (Benutzungshierarchie).
3. Vererbung und der dynamisches Binden führen sogar zu einem exponentiellen Wachstum der Interaktionsmöglichkeiten.

Wir betrachten im Rest dieses Abschnitts einige bekannte Ansätze zum Integrations- und Regressionstest für objektorientierte Software. Danach legen wir einige Nachteile der verschie-

denen Ansätze dar und zeigen, dass der objektorientierte Integrations- und Regressionstest in weiten Teilen übereinstimmende Aktivitäten sind und somit gemeinsam behandelt werden können. In Abschnitt 2 untersuchen wir hierzu die möglichen Interaktionen in objektorientierten (Java-)Programmen und geben in Abschnitt 3 ein diese Interaktionen umfassendes Modell an. In den Abschnitten 4 und 5 stellen wir die Anwendung unseres Modells im Integrations- und Regressionstest für objektorientierte Software vor und skizzieren die Ergebnisse einiger Experimente, wobei wir uns auf die Ermittlung von Testreihenfolgen und die Auswahl geeigneter Testfälle konzentrieren. Die Ableitung oder Generierung der Testfälle ist nicht Gegenstand dieses Beitrags. Abschnitt 6 schließt den Beitrag ab.

1.1 Integrationstest

Der Integrationstest prüft das Zusammenspiel (von Instanzen) der Klassen des Anwendungssystems. Die Reihenfolge, in der die Klassen zusammenmontiert und (integrations-)getestet werden, ist durch die *Integrationsstrategie* (siehe z.B. [13][24]) bestimmt. Diese hängt eng mit der Implementations- und des Klassentestreihenfolge zusammen, da natürlich nur Klassen integriert (und getestet) werden können, deren Einzeltests bereits abgeschlossen sind. Dabei hat die Wahl der Integrationsstrategie einen großen Einfluss auf die Zahl der erforderlichen Testtreiber und -stellvertreter [18].

In der Literatur wird zwischen vorgehensorientierten und testzielorientierten Integrationsstrategien sowie zwischen inkrementellen und nicht-inkrementellen Ansätzen unterschieden [19]:

- *Vorgehensorientierte Strategien* gehen meist vom Komponentengraphen aus und leiten daraus die Integrationsreihenfolge ab.
- *Zielorientierte Integration* bedeutet, dass basierend auf den Testzielen konkrete Testfälle bestimmt und dann die zur Ausführung der Testfälle benötigten Komponenten zusammenmontiert werden. Ein solches Ziel kann beispielsweise sein, codierte und getestete Klassen so schnell wie möglich zu integrieren.
- *Inkrementell* vorzugehen heißt, einzelne sehr kleine Gruppen von Klassen zu integrieren, während *nicht-inkrementelles Vorgehen* die gleichzeitige Integration aller oder einer größeren Anzahl von Klassen (z.B. aller Klassen eines Teilsystems) bedeutet.

Da neuere Vorgehensmodelle bei der Entwicklung von (objektorientierter) Software ein inkrementelles Vorgehen empfehlen ([5][12][4]), erläutern wir kurz einige inkrementelle Strategien:

- Integration *nach der Verfügbarkeit* heißt, dass die Reihenfolge durch die Implementierung bestimmt wird.
- Bei einer *anwendungsfallbasierten Strategie* werden diejenigen Komponenten entwickelt, integriert und getestet, die von einem ausgewählten Anwendungsfall betroffen sind.
- Bei einer *funktionsbezogenen Strategie* werden funktionale Testfälle (z.B. aus der Beschreibung der Operationen im Klassenmodell) spezifiziert und überprüft. Die betroffenen Komponenten werden schrittweise integriert und getestet, wobei wieder die *Top down-*, *Bottom up-*, *Inside out-* und die *Outside in-Vorgehensweise* unterschieden werden [19].
- *Hardest first* heißt, zunächst die kritischen, d.h. besonders kompliziert zu testenden oder potentiell fehlerbehafteten Komponenten zu bearbeiten.

Im Folgenden stellen wir einige bekannte Integrationsstrategien für objektorientierte Software vor. Wir unterscheiden vererbungsbezogene, vererbungs- und nutzungsbezogene sowie funktionsbezogene Strategien:

Vererbungsbezogen vorzugehen bedeutet, ein Programm (oder ein Teilsystem) auf der Grundlage des Klassendiagramms von den (abstrakten) Klassen an der Wurzel der Vererbungshierarchie hin zu den (konkreten Klassen) an den Blättern der Vererbungshierarchie zu implementieren, zu integrieren und zu testen. Zum Test abstrakter Klassen sind die abstrakten Operationen mit jeweils „minimaler" Funktionalität zu implementieren [18].

Ging man ursprünglich davon aus, dass bereits getestete Methoden aus Oberklassen in den Unterklassen ohne erneute Tests wiederverwendet werden können, so wurde dies durch die Arbeit von Perry und Kaiser widerlegt [20]. Wir skizzieren die möglichen Abhängigkeiten innerhalb einer Vererbungshierarchie und ihre Auswirkungen im Einzelnen:

- Ändert man die Implementierung einer Klasse **K**, so muss man zunächst einen Klassentest für **K** ausführen und dann alle **K** benutzenden Klassen erneut testen. Letzteres deshalb, weil durch gegenseitige Aufrufe gemeinsam verwendeter Klassen und der geänderten Klasse Wechselwirkungen auftreten können, die beim Test der Klasse **K** allein nicht festgestellt werden.
- Legt man eine Klasse **K'** als Unterklasse einer Klasse **K** an, so sind zunächst alle neuen Methoden von **K'** zu testen, danach alle redefinierten Methoden und letztendlich alle von **K** geerbten Methoden.

  Im Falle redefinierter Methoden muss man zusätzlich zu den vorhandenen funktionalen Testfällen der entsprechenden Methoden der Oberklasse neue strukturelle Testfälle erstellen und ausführen.
  Auch für den Test der unverändert geerbten Methoden reicht es nicht aus, die für die entsprechenden Methoden der Oberklasse erstellten funktionalen und strukturellen Testfälle auszuführen. Man muss zusätzlich für unverändert geerbte Methoden, die von der Unterklasse redefinierte Methoden benutzen, neue funktionale Testfälle erstellen und ausführen.
  Stellt die Unterklasse lediglich eine Spezialisierung der Oberklasse dar — definiert also lediglich neue Attribute und Methoden, die mit den geerbten nicht interferieren —, erspart man sich die Ausführung vorhandener sowie das Erstellen neuer Testfälle für die geerbten Attribute und Methoden.

Diese Testabhängigkeiten werden von dem Algorithmus von McGregor und Harrold zum inkrementellen Klassentesten in Vererbungshierarchien berücksichtigt [9], der zu einer vererbungsbezogenen Integrationsstrategie führt.

Als kombinierte *vererbungs- und nutzungsbezogene Strategie* gibt Overbeck aufbauend auf [20] und [9] ein Verfahren zur Ermittlung von (Test-)Reihenfolgen auf der Grundlage des Klassendiagramms an, das sich an den Vererbungsbeziehungen top-down, also von Basisklassen zu abgeleiteten Klassen, orientiert [18]. Nutzungsbeziehungen (gerichtete Assoziationen) zwischen Klassen [15] werden dabei bottom-up berücksichtigt, d.h. von rein dienstanbietenden Klassen bis hin zu rein dienstnutzenden Klassen. Die mit diesem Verfahren ermittelten Integrationsstrategien minimieren die Anzahl der für den Test benötigten Testtreiber und -stellvertreter. Probleme, die aus zyklischen Nutzungsbeziehungen resultieren, sollen durch „Aufbrechen" der Zyklen mit entsprechenden Teststellvertretern behandelt werden. Overbeck baut das Verfahren in sechs aufeinander aufbauenden Stufen auf, mit denen Integrationsstrategien für Programmkonstrukte wachsender

Komplexität ermittelt werden können.

Funktionsbezogen: Spillner schreibt zum (modularen) funktionsbezogenen Integrationstest:

> Aufgabe des funktionsbezogenen Tests ist die Prüfung, ob die einzelnen realisierten Operationen in Bezug auf ihre zur Verfügung gestellte Funktionalität entsprechend der Spezifikation zusammenwirken. Dieser Testansatz entspricht dem weitverbreiteten Verständnis des Integrationstests: die Prüfung der einzelnen Systemteile in Hinblick auf ihre Funktionalität. [...] Folgende Fehler können auftreten und müssen aufgedeckt werden: falsche Funktionalität, fehlende Funktionalität, zusätzliche Funktionalität. [24]

Eine rein funktionsbezogene Reihenfolge geht von den einzelnen Methoden und den Aufruf- bzw. Benutzungsbeziehungen zwischen ihnen aus. Jorgensen's und Erickson's Ansatz zum objektorientierten Integrationstest orientiert sich an der Funktionalität des Anwendungssystems unter Berücksichtigung des "Inter-Klassen Kontrollflusses" [11]. Aufbauend auf der Arbeit von Deutsch [6] werden hierbei Stimulus/Antwort-Elemente (Methoden-Botschafts-Pfade, method-message-path's, MM-Paths) identifiziert und zu atomaren Funktionen (atomic system functions, ASFs) zusammengesetzt. Zuerst werden einzelne MM-Pfade, dann deren Zusammenspiel in ASFs getestet.

1.2 Regressionstest

Änderungen an existierenden Klassen können korrektiv (Spezifikation verletzt), inkrementell (Spezifikation erweitert), adaptiv (Spezifikation geändert) oder optimierend (Spezifikation unverändert) sein. In keinem Fall darf hierbei eine Regression auftreten, d.h., die geänderten Klassen müssen weiterhin (den unveränderten Teilen) der Spezifikation genügen. Dies wird durch entsprechende *Regressionstest*s geprüft. Zwei grundlegende Probleme hierbei sind

- die *Änderungsanalyse* (change impact analysis, vgl. [3]) zur Bestimmung der *Regressionsmenge*, d.h. der von einer Änderung möglicherweise betroffenen und somit neu zu testenden Teile des Anwendungssystems sowie
- die darauf basierende *Testfallauswahl* (selective regression testing, vgl. [22]), also die Bestimmung aller im Regressionstest erneut auszuführenden Testfälle.

Letzteres ist z.B. dann notwendig, wenn sich die Ausführung aller Tests nach jeder Iteration aus Zeitgründen verbietet. Eine sichere Testfallauswahl [22] muss hierbei alle Testfälle berücksichtigen, deren Ausführung Fehler im modifizierten Anwendungssystem aufdecken können.

McGregor et al. stellen im Hinblick auf interprozedurale Kontroll- und Datenflussanalysen sowie Alias-Untersuchungen von C++ Programmen den *Object-Oriented Program Dependency Graph* (OPDG) als ein heterogen zusammengesetztes Modell vor [14]. Der OPDG besteht aus den vier Teilgraphen

- Klassendiagramm (class hierarchy subgraph, CHS),
- Kontrollflussgraphen der Methoden (control dependency subgraphs, CDS),
- Datenflussgraphen der Methoden (data dependency subgraphs, DDS) und
- Objektkonstellationen (object dependency subgraphs, ODS) als Ergebnisse abstrakter Interpretationen des CHS, CDS und DDS.

Rothermel und Harrold geben ein auf dem OPDG aufbauendes, datenflussbezogenes Verfahren zur Änderungsanalyse und Testfallauswahl an, welches die Verfolgbarkeit von Testfällen zu Programmanweisungen z.B. durch Instrumentierungsprotokolle erfordert [22]. Aufgrund der feinen Granularität und der damit verbundenen Größe der Graphen bemerken die Autoren bezüglich dynamisch gebundener Aufrufe:

> For each call, we can detect a set of methods it can invoke ... when this set is too large, further graph construction and traversal through that call is impractical [22].

Kung et al. verwenden ein ähnliches Modell [13]. Zur Identifikation der Änderungen generieren sie für jede Methode einen um Datenflussinformationen angereicherten Kontrollgraphen (block branch diagramm, BBD). Anhand des ebenfalls aus dem Programmcode generierten Klassendiagramms (object relation diagramm, ORD) werden dann sowohl die Regressionsmenge („Klassen-Feuerschutzmauer", class firewall) als auch die Testreihenfolge (test order) bestimmt.

1.3 Diskussion

Zunächst fällt auf, das die z.B. in [9], [13] oder [18] rein strukturell aus dem Klassendiagramm abgeleiteten Reihenfolgen bzw. Integrationsstrategien nicht immer unmittelbar umsetzbar sind, weil

- im Klassendiagramm oft zyklische Nutzungsbeziehungen auftreten ([1][18][13]), die zu Problemen bei der Ermittlung der Testreihenfolge führen und die Konstruktion von Testtreibern und -stellvertretern für ganze Klassen erzwingen, die gerade in der objektorientierten Welt sehr aufwendig sein kann ([1][23]),
- Testfälle und -daten für die während der Integration benötigten partiellen, nicht im vollständigen Programm vorhergesehenen Objektkonstellationen z.T. schwer zu ermitteln sind und
- funktionsfähige Cluster im schlechtesten Fall erst nach Abschluss der Testphase bereitstehen und besonders risikoreiche Cluster nicht gesondert behandelt werden können.

Rein funktionsbezogene Ansätze wie z.B. der von Jorgensen und Erickson berücksichtigen jedoch objektorientierte Eigenschaften wie Vererbung und dynamisches Binden nicht.

Weiterhin fallen folgende Gemeinsamkeiten des objektorientierten Integrations- und Regressionstests auf:

- Sowohl für den Integrationstest als auch für den Regressionstest müssen geeignete Ausführungsreihenfolgen ermittelt werden, welche die aus der Konstruktion von Testtreibern und -stellvertretern resultierenden Testkosten minimieren.
- Eine neue Klasse ist immer Unterklasse mindestens einer vorhandenen Klasse (z.B. Object). Der Integrationstest bzgl. der neuen Klasse erfordert also immer auch den Regressionstest bzgl. der Oberklasse(n). Umgekehrt muss jede geänderte Klasse erneut integriert werden.
- Für beide Testarten sind vorhandene Testfälle auszuwählen, die gezielt das Zusammenspiel der zu integrierenden Klassen bzw. die Änderungen an vorhandenen Klassen prüfen.

Unser Testmodell kombiniert daher objektorientierte struktur- und funktionsbasierte Aspekte geeignet, um sie sowohl im Integrations- als auch im Regressionstest zu berücksichtigen. Hierbei wird besonderes Augenmerk auf die in Klassendiagrammen häu-

fig auftauchenden zyklischen Abhängigkeiten gelegt. Da diese oft zu einzelnen, nicht-zyklischen Abhängigkeiten (Aufrufen) zwischen Methoden zerfallen ([16], vgl. auch Abschnitt 5 dieses Beitrags), liegt unserem Ansatz die Granularität von Methoden, Botschaften (Methodenaufrufen) und Variablen zugrunde [25]. Auf dieser Ebene werden Abhängigkeiten auf interne Interaktionen, also Interaktionen zwischen Objekten, zurückgeführt.

2 INTERAKTIONEN

In diesem Abschnitt zeigen wir, welche internen Interaktionen in objektorientierten Programmen prinzipiell möglich sind und in einem Testmodell für objektorientierte Software berücksichtigt werden müssen. Da im Folgenden keine Verwechslung mit externen Interaktionen, also Interaktionen zwischen einem Programm und den Akteuren, zu befürchten ist, sprechen wir kurz von Interaktionen und meinen interne Interaktionen.

Wir präzisieren den Begriff der (internen) *Interaktion* in objektorientierten Programmen folgendermaßen:

- Sendet ein Objekt bei der Ausführung einer Methode eine Botschaft an ein (nicht notwendigerweise verschiedenes) Objekt bzw. greift (lesend oder schreibend) auf eine Klassen- oder Instanzvariable zu, so sprechen wir von einer *direkten Aufruf- bzw. Dateninteraktion*.
- Beeinflusst eine direkte Interaktion die Ausführung einer Methode, so nennen wir dies eine *indirekte Interaktion*. Bei der hier betrachteten Klasse objektorientierter Programm sind indirekte Interaktionen nur über die Manipulation von Klassen- oder Instanzvariablen, also durch direkte Dateninteraktionen möglich, sodass wir auch von *Zustandsinteraktionen* reden.

Weiterhin unterscheiden wir, ob die Interaktion stattfindet

- innerhalb eines Objekts (Intra-Klassen-/Intra-Objekt),
- zwischen zwei Objekten der gleichen Klasse (Intra-Klassen-/ Inter-Objekt),
- zwischen zwei Objekten verschiedener Klassen innerhalb einer Vererbungshierarchie, wobei die eine Unterklasse der anderen ist (und umgekehrt, Intra-Klassenhierarchie) oder
- zwischen zwei Objekten verschiedener Klassen, die nicht über die Vererbung verbunden sind (Inter-Klassenhierarchie).

Wir orientieren uns bei den folgenden Betrachtungen an den sequentiellen Konstrukten der Programmiersprache Java (s. [8]). Zur Vereinfachung der Darstellung gehen wir davon aus, dass alle Methoden sowie ggf. Klassen- bzw. Instanzvariablen für alle Klassen eines Programms sichtbar sind (**public**), denn die in (guten) objektorientierten Programmen oft verwendeten Sichtbarkeiten geschützt (**protected**) und privat (**private**) schränken die Anzahl möglicher Interaktionen höchstens ein. Da in Java enge Wechselwirkungen zwischen den Kapselungskonstrukten und den Regeln zur Bindung von Methoden bestehen, die im Extremfall bis hin zu Anomalien bei der dynamischen Bindung von Methodenaufrufen bzw. Botschaften an die ausgeführte Methode führen können (vgl. [17]), betrachten wir hier auch den Paketmechanismus von Java nicht.

Die Java-Sprachspezifikation unterscheidet Klassen- und Instanzmethoden, wobei erstere durch das Schlüsselwort **static** in der Deklaration gekennzeichnet sind. Wird eine Klassenmethode in einer Unterklasse überschrieben (*hiding*), so ist die überschriebene Methode durch Verwendung des Schlüsselworts **super** innerhalb der Methoden der Unterklasse bzw. nach einem „*cast*" eines Objekts der Unterklasse in eine Instanz der Oberklasse auch „von Aussen" (Inter-Objekt) sichtbar. Überschriebene Instanzmethoden (*overriding*) sind im Gegensatz dazu von Aussen nicht sichtbar.

Wir präzisieren im Folgenden die möglichen Interaktionen. Hierbei unterscheiden wir jeweils im Kontext der Klasse des aufrufenden bzw. aufgerufenen Objekts

- neu definierte (N),
- redefinierte (R) sowie
- unverändert geerbte (G) Methoden (vgl. [9][13][20]) und zusätzlich auch noch
- überschriebene Methoden (overridden/hidden, O) (vgl. [2]).

Daneben betrachten wir den Fall des Aufrufs einer identischen Methode jeweils gesondert (Rekursion, s. auch [2]). Zur Illustration zeigen wir dieses Vorgehen für direkte Intra-Klasseninteraktionen. Zum Schluss skizzieren wir kurz den Fall indirekter Interaktionen.

2.1 Direkte Intra-Klassen-Interaktionen in Java

Für die Intra-Klassen-Interaktionen ergeben sich die in Tabelle 1 gezeigten Kombinationen von aufrufender Methode m1 in Objekt o1 („Sender") und aufgerufener Methode m2 in Objekt o2 („Empfänger"). Unterschiede bei Klassen- und Instanzmethoden sind durch vorangestelltes I bzw. C kenntlich gemacht. Betrachten wir zunächst die direkten Intra-Klassen/Intra-Objekt-Interaktionen in Tabelle 1, Spalte A, für die Abb. 2 jeweils Beispiele zeigt. Insbesondere die Interaktionsarten 10.A und 14.A spiegeln die Grundidee der Erweiterung objektorientierter Klassen wider: das gezielte Überschreiben von Methoden, deren Aufruf in sogenannten „Templatemethoden" erfolgt (s. [7]). Bezüglich der direkten Intra-Klassen/Inter-Objekt-Interaktionen betrachten wir Tabelle 1, Spalte B sowie Abb. 3. Da überschriebene Instanzmethoden in Java im Gegensatz zu C++ (s. auch [2]) auch nicht durch eine explizite Typangabe (*primary* bzw. *cast*, [8]) außerhalb einer Klassendefinition sichtbar werden, berücksichtigen wir sie bei Inter-Objekt-Interaktionen nicht. Wohl aber berücksichtigen wir, dass überschriebene Instanzmethoden durch vorangestelltes **super** von einer Methode der überschreibenden Klasse ausgeführt und somit selbst Aufrufer werden können[1].

1. [9], [10] und [20] betrachten überschriebene Methoden nicht.

Nr.	Typ m1	Typ m2	A: Intra-Objekt o1=o2	B: Inter-Objekt o1<>o2	
1.	N	N	Standard-Aufruf (auch **this**)	Standard-Aufruf	
2.	N	R	Wie 1.	Wie 1.	
3.	N	G	Wie 1. (redundantes **super**)	Standard, „Vertikale Kopplung“	
4.	N	O	Fraglicher Gebrauch von **super**	I: In Java nicht möglich, K: nach Cast	
5.	R	N	Wie 1.	Wie 1.	
6.	R	R	Wie 1.	Wie 1.	
7.	R	G	Wie 3.	Wie 3.	
8.	R	O	Standardgebrauch von **super**	I: In Java nicht möglich, K: nach Cast	
9.	G	N	In Java nicht möglich	I: In Java nicht möglich, K: nach Cast	
10.	G	R	Polymorphismus (auch **this**)	Polymorphismus	
11.	G	G	Wie 1.	Wie 1.	Legende
12.	G	O	In Java nicht möglich	I: In Java nicht möglich, K: nach Cast	N: Neu definierte Methode
13.	O	N	In Java nicht möglich	I: In Java nicht möglich, K: nach Cast	R: redefinierte Methode
14.	O	R	Polymorphismus (nur nach **super**)	Nur nach **super** (Template/Strategy)	G: geerbte Methode
15.	O	G	Nur nach **super**	Nur nach **super**	O: überschriebene Methode
16.	O	O	In Java nicht möglich	I: In Java nicht möglich, K: nach Cast	I: Instanzmethode
17.	x	x	m1=m2 , „echte“ Rekursion	m1=m2, strukturelle Rekursion (Iterator)	K: Klassenmethode

Tab. 1. Direkte Intra-Klassen-Interaktionen in Java

Wir fassen unsere bisherigen Überlegungen zusammen: Als direkte Aufrufinteraktionen haben wir die durch die Schlüsselworte **this** bzw. **super** erkenntlichen dynamisch bzw. statisch gebundenen reflexiven (vgl. [2]) Aufrufe sozusagen als Möglichkeiten der vertikalen Wiederverwendung oder Kopplung (innerhalb einer Vererbungshierarchie) dargestellt. Alle anderen Aufrufe bedeuten eine horizontale Wiederverwendung oder Kopplung im Sinne der *Delegation* (vgl. [7]), also dynamisch gebundene einfache Aufrufe. Überschriebene (*overridden*) Instanzmethoden sind „von außen“ nicht sichtbar; überschriebene (*hidden*) Klassenmethoden sind nach einem Cast „von außen“ sichtbar.

2.2 Indirekte Interaktionen in Java

Haworth et al. bezeichnen sowohl neu- als auch redefinierte Methoden bzw. Variablen als „local“ und unverändert geerbte Methoden bzw. Variablen als „inherited“ ([10]). Sie notieren indirekte Intra-Klassen/Intra-Objekt-Interaktionen in der Form <m1, v, m2>, wobei m1 den Typ der schreibenden Methode, v die Art der betroffenen Variablen und m2 die Art der lesenden, also beeinflussten Methode bedeuten. Hierbei können zwischen je einem modifizierenden Zugriff und lesenden Zugriffen liegende Interaktionen dann vernachlässigt werden, wenn sie die modifizierte Variable nicht erneut modifizieren — Haworth et al. betrachten also im Prinzip analog zum datenflussbasierten Test sogenannte „def-uses-Paare“ (vgl. [19][21]). So bedeutet z.B. das Tripel <N, I, R>, dass nach der Modifikation einer Variablen durch

```
class A {
    public void opA() {
        this.opC();    // 1A, wenn in A-Instanz ausgeführt,
                       // 10A,   -"-   B-Instanz      -"-
        this.opA();    // 17A,   -"-    A-Instanz      -"-
    }
    public void opB() {
        this.opA();    // 11A, wenn in B-Instanz ausgeführ
    }
    public void opC() {
        this.opD();    // 14A,              -"-
        this.opA();    // 15A,              -"-
    }
}
class B extends A {
    public void opC() {
        this.opE();    // 5A
        this.opD();    // 6A
        this.opA();    // 7A
        super.opC(); // 8A, Standardgebrauch von super
    }
    public void opE() {
        this.opC();    // 2A
        this.opA();    // 3A
        super.opC(); // 4A, fragwürdiger Gebrauch von super
    }
}
```

Abb. 2 Direkte Intra-Klassen, Intra-Objekt-Interaktionen in Java

eine neu definierte Methode eine geerbte Methode die modifizierte (geerbte) Variable liest.

```
class A {
    public A myA;
    public void opA() {
        myA.opC();   // 10B, wenn in A-Instanz ausgeführt,
                     //      und myA.Class = B
        myA.opA();   // 11B + 17B,      -"-
    }
    public void opC() {
        myA.opD();   // 14B,            -"-
        myA.opA();   // 1B,             -"-
    }
}
class B extends A {
    public B myB;
    public void opC() {
        myB.opE();   // 5B
        myB.opD();   // 6B
        myB.opA();   // 7B
        super.opC(); // 14B + 15B, Standardgebrauch von super
    }
    public void opE() {
        myB.opE();   // 1B + 17B
        myB.opD();   // 2B
        myB.opA();   // 3B
    }
}
```

Abb. 3 Direkte Intra-Klassen-, Inter-Objekt-Interaktionen in Java

Im Gegensatz zu überschriebenen Instanzmethoden sind überschriebene Instanzvariablen durch entsprechende Typkonvertierungen (*cast*) außerhalb der Klasse sichtbar. Gosling et al. schreiben:

> Indeed, there is no way to invoke the [overridden] getX method of class Point for an instance of class RealPoint from outside the body of RealPoint, no matter what the type of the variable we may use to hold the reference to the object. Thus, we see that fields and methods behave differently: hiding is different from overriding. [8]

Letztendlich müssen wir noch den Unterschied zwischen Klassen- und Instanzvariablen sowie die Effekte der gleichzeitigen Referenzierung (und Manipulation) eines Objekts über mehrere Variablen (*aliases*) berücksichtigen. Während die Modifikation einer Instanzvariablen nur den Zustand eines Objekts unmittelbar beeinflusst, wirkt sich die Modifikation einer Klassenvariablen (in Java mit dem Schlüsselwort **static** deklariert) auf alle Instanzen der Klasse (und ihrer Unterklassen) aus. Im Falle eines Alias können Probleme auftreten, wenn der Zustand eines mehrfach referenzierten Objekts unkoordiniert geändert wird.

Die Testproblematik verschärft sich also bei der Einbeziehung von Instanzvariablen atomaren Typs hin zu Instanzvariablen vom Referenztyp ohne Aliases über Instanzvariablen mit der Möglichkeit von Aliases. Kommen darüber hinaus noch Klassenvariablen vom atomaren Typ oder sogar vom Referenztyp mit der Möglichkeit „globaler" Aliases („Singleton", s. [7]) hinzu, sollten zur adäquaten Prüfung des Programms zusätzlich globale Kontroll- und Datenfluss- sowie Aliasanalysen durchgeführt werden (vgl. [14]).

3 DAS KLASSEN-BOTSCHAFTSDIAGRAMM

Wir streben ein (möglichst einfaches) Modell an, welches unter Beachtung der Vererbungsstruktur die möglichen Interaktionen zwischen Objekten erfasst. Als Interaktionen betrachten wir im obigen Sinne den „Botschaftsfluss" sowie mögliche Zustandsänderungen. Wir gehen hierbei von folgenden Beobachtungen aus:

- Die Quelle einer Botschaft liegt syntaktisch eindeutig identifizierbar in einer (im Kontext eines Objekts ausgeführten) Methode, die wir *Quellmethode* der Botschaft nennen.
- Das *Zielobjekt* einer Botschaft ist Instanz einer Klasse, die Element einer Menge polymorph substituierbarer — bei getypten Sprachen syntaktisch ermittelbarer — Klassen ist. „Versteht" das Zielobjekt die Botschaft, so führt dies zur Ausführung einer dynamisch gebundenen Methode, der *Zielmethode* der Botschaft.
- Der aktuelle *Zustand* als Äquivalenzklassen von Werten der Klassen- und Instanzvariablen (vgl. [1]) bestimmt den Ablauf der Methoden, wobei diese Variablen innerhalb von Methoden gelesen und modifiziert werden.

Im Folgenden skizzieren wir, wie ausgehend von diesen Beobachtungen aus Java-Quellcode das *Klassen-Botschaftsdiagramm* (*KBD*) abgeleitet wird. Ein Algorithmus zur Konstruktion des KBD aus Java-Quellcodes findet sich im Anhang von [26].

3.1 Elemente des Klassen-Botschaftsdiagramms

Als Knoten des KBD kommen prinzipiell Klassen, Objekte, Methoden oder einzelne Anweisungen in Frage. Klassen scheiden als zu grobgranular aus, da Botschaften nur innerhalb von Methoden erzeugt werden. Objekte als Knoten würden das Modell auf die jeweils betrachtete Objektkonstellation einschränken und somit in einem Modell nicht alle möglichen Interaktionen erfassen. Würden wir den Knoten einzelne „Anweisungen" zuordnen, so müssten wir zusätzlich eine dem Kontrollfluss innerhalb der Methoden entsprechende Kantenart zufügen, die jedoch keiner Botschaft entspräche. Dies würde der Forderung, dass wir uns auf die (durch Botschaften) möglichen Interaktionen zwischen Objekten in einem Programm konzentrieren wollen, zuwiderlaufen. Darüber hinaus wäre die Granularität des so entstehenden Graphen zu fein für den Test eines vollständigen Programms.

Wir wählen daher „member" [8] oder „features" [15] als die den Knoten des KBD entsprechenden Elemente der Implementation aus, also Methoden sowie Klassen- und Instanzvariablen. Im Weiteren bezeichnen **A** und **B** Klassen. Falls Verwechslungen ausgeschlossen sind, sagen wir im Kontext des KBD einfach „Methodenknoten **A**::a" anstatt „der die Methode a in Klasse **A** repräsentierende Knoten mit dem Namen <A::a>".

Wir betrachten zunächst Methodenaufrufe (*direkte Aufrufinteraktionen*). Eine einfache gerichtete *Botschaftskante* verbindet Methodenknoten **A**::a mit Methodenknoten **B**::b, wenn bei der Ausführung von a in einer Instanz der Klasse **A** eine Botschaft zu einem Objekt der Klasse **B** gesendet werden kann, welche die Ausführung der Methode b bewirkt. In anderen Worten: Methode **A**::a benutzt Methode **B**::b.

Als nächstes beziehen wir die Vererbung, den Polymorphismus sowie die damit verbundenen Schlüsselworte bzw. Konstrukte **this** und **super** zur Steuerung der (dynamischen) Methodenbindung mit ein. Jedes Vorkommen eines Methodenaufrufs der Form **this** a in einer Methode x der Klasse **A** wird als mit der Marke this gekennzeichnete Botschaftskante von Methodenknoten x zu dem Methodenknoten dargestellt, welcher der ersten Deklaration oder Redefinition der Methode a „aufwärts" in der Vererbungshierarchie der Klasse **A** (inklusive **A**) entspricht. In der gleichen Art und Weise wird jedes Vorkommen eines Methodenaufrufs der Form **super** a in einer Methode x der Klasse **A** als mit der Marke super gekennzeichnete Botschaftskante von Methodenknoten x zu dem Methodenknoten dargestellt, welcher der ersten Deklaration oder Redefinition der Methode a „aufwärts" in der Vererbungshierarchie der Klasse **A** (exklusive **A**) entspricht.

Ist eine Klasse **B** Unterklasse von **A** (**B** < **A**) und Methode **B**::a redefiniert Methode **A**::a (**B**::a < **A**::a), so fügen wir eine als *Bindungskante* bezeichnete, gerichtete und mit der Marke inheritance gekennzeichnete Kante von Methodenknoten **B**::a zu Methodenknoten **A**::a ein. Zusätzlich duplizieren wir in diesem Fall alle nicht mit super markierten Botschaftskanten, die in Methodenknoten **A**::a enden, für den Methodenknoten **B**::b. Die duplizierten und „umgeleiteten" Botschaftskanten repräsentieren explizit, dass „Aufrufe" der Methode a dynamisch an Objekte der (polymorph substituierbaren) Klassen **A** und **B** gebunden werden können.

Für die Darstellung des KBD treffen wir folgende Vereinbarungen:

- Wir stellen Operationen durch ein Rechteck mit abgerundeten Kanten dar. Im Rechteck steht der Name der Operation, mit „::" abgesetzt wird diesem der Name der Klasse, in der die Operation definiert ist, vorangestellt.
- Botschaftskanten stellen wir durch offene Pfeile von aufrufenden zu aufgerufenen Operationen dar, die ggf. mit der Marke gekennzeichnet werden.
- Bindungskanten werden durch einen geschlossenen, nicht ausgefüllten Pfeil von redefinierenden zu überschriebenen Methoden gezeichnet.
- Die Ausgangspunkte von Botschaftskanten, die dynamisches Binden eines einzigen Aufrufs repräsentieren, werden zusammengelegt.

Beispiel 3.1 Abb. 4 zeigt ein Java-Codefragment mit zwei Klassen **A** und **B**, wobei **B** direkte Unterklasse von **A** ist.

```
class A {
    public void opA() {
        this.opB();
    }
    public void opB() {...}
}
class B extends A {
    public void opB() {...}
```

Abb. 4 Java-Quellcode

Das KBD für die Klassen **A** und **B** zeigt Abb. 5. Zunächst sehen wir fünf Methodenknoten anstelle von drei, da die „Default-Konstruktoren" für beide Klassen berücksichtigt sind (vgl. [8]). Mit der Bindungskante wird die Redefinition der Methode opB in der Klasse **B** berücksichtigt. Die Botschaftskante von **A**::opA zu Methode opB in Klasse **A** reflektiert den Gebrauch des Schlüsselworts **self** in **A**::opA und die mögliche dynamische Bindung an **B**::opB.

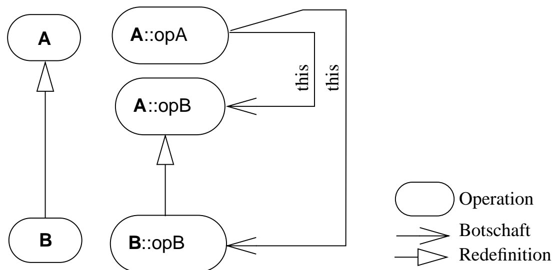

Abb. 5 KBD zum Java-Code aus Abb. 4

❑

Nun betrachten wir die (explizite) Erzeugung von Objekten im Programm. Sei **A**(k0, ..., kn) ein Konstruktor der Klasse **A**. Jedes Vorkommen eines Methodenaufrufs der Form **new A**(k0, ..., kn) in einer Methode x wird als eine mit der Marke new gekennzeichnete Botschaftskante von Methodenknoten x zu dem (Konstruktor-)Methodenknoten mit entsprechender Signatur in der Klasse **A** dargestellt. Wir nennen solche Kanten auch Instanziierungskanten.

Da in Java (wie auch in Eiffel, vgl. [15]) im Gegensatz zu C++ die „explizite" Zerstörung von Objekten nicht vorgesehen ist („garbage collection", siehe z.B. [8][15]), gibt es im KBD bei einer Implementation in Java keine mit destroy gekennzeichneten Kanten. Die explizite Entfernung persistenter Objekte (z.B. aus einer Datenbank) spiegelt sich in der Implementation in Methoden wider, die zurück zu entsprechenden destroy-Standardoperationen im Domänen-Klassenmodell der Anforderungsspezifikation verfolgbar sind.

Zuletzt betrachten wir noch die Zustandsspeicher, also Klassen- bzw. Instanzvariablen und entsprechende Zugriffe (*direkte Dateninteraktionen*). Für jede Variable fügen wir dem KBD einen mit dem qualifizierten Namen der Variablen gekennzeichneten Variablenknoten hinzu. Zugriffe auf die Variable werden dann auf speziell markierte Botschaftskanten abgebildet:

- Eine mit uses gekennzeichnete Kante vom Methodenknoten a zum Variablenknoten x bedeutet, dass während der Ausführung von Methode a nicht-modifizierend auf Variable x zugegriffen werden kann.
- Analog bedeutet eine mit def markierte Kante vom Variablenknoten x zum Methodenknoten a, dass bei der Ausführung von Methode a die Variable x modifiziert werden kann. Anders herum: Variable x „benutzt" Methode a zu ihrer Definition.

Variablen und Zugriffe auf Variablen werden im KBD folgendermaßen dargestellt:

- Knoten, die Instanz- oder Klassenvariablen zugeordnet sind, zeichnen wir als Parallelogramme, in die wir die qualifizierten Namen der Variablen schreiben.
- Mit new, def oder uses markierte Kanten stellen wir wie Botschaftskanten dar, also durch offene Pfeile, die mit der jeweiligen Marke gekennzeichnet sind.

Beispiel 3.2 Abb. 6 zeigt ein Java-Codefragment mit zwei Klassen **C** und **D**, wobei **D** direkte Unterklasse von **C** ist.

```
class C {
    public integer a;
    public void opA() {
        a = 1;
    }
}
class D extends C {
    public C einC;
    public void opA() {
        super.opA;
        a = a + 1;
        einC.opA
    }
```

Abb. 6 Java-Quellcode

Das KBD für die Klassen **C** und **D** zeigt Abb. 7. Eine mit def markierte Kante verbindet die Variablenknoten mit den jeweiligen Konstruktorknoten. Die Redefinition der Methode opA in der Klasse **D** wird mit der Bindungskante berücksichtigt. Die Botschaftskante von **D**::opA zu Methode opA in Klasse **C** reflektiert den Gebrauch des Schlüsselworts **super** in **D**::opA und die mögli-

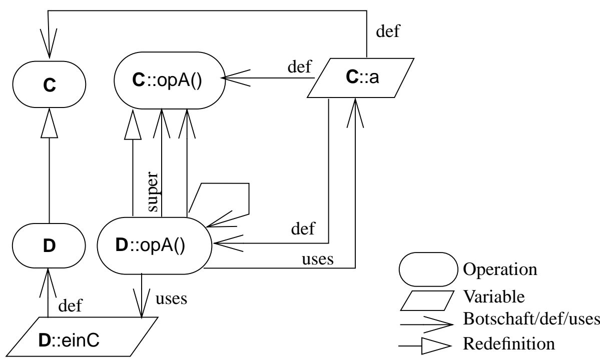

Abb. 7 KBD zum Java-Code aus Abb. 6

che (polymorphen) Bindungen der Botschaftsquelle einC.opA. Die Anweisung einC.opA in der Methode **D**::opA führt dementsprechend auch zur reflexiven Botschaftskante mit dem Ziel **D**::opA.

Die mit def markierten Kanten vom Variablenknoten **C**::a zu den drei Methodenknoten **C**, **C**::opA und **D**::opA zeigen die (mögliche) Modifikation der Instanzvariablen a und damit direkte Dateninteraktionen an. Die mit uses markierten, von der Methode **D**::opA ausgehenden Kanten bedeuten die mögliche, nicht-modifizierende Verwendung der Instanzvariablen einC und a in dieser Methode. ❑

Diese Sachverhalte fassen wir in der folgenden Definition zusammen.

Definition 3.1 Seien **K** die Interfaces und Klassen („Universum“) und OPS die Menge aller Operationen bzw. Methoden in einem Programm. Das *Klassen-Botschaftsdiagramm* (KBD) ist ein gerichteter, knoten- und kantenmarkierter Multigraph (V, E, P) über den Markenmengen L_V und L_I zusammen mit einer zweistelligen Relation P über E:

- $V \subseteq$ OPS ist die Menge aller expliziten *Methodendefinitionen* im Universum **K**.
- L_V = {class, instance} × {method, field} ist die Menge aller *Knotenmarkierungen* mit der Bedeutung:

  class: Knoten gilt für alle Instanzen einer Klasse;
  instance: Knoten gilt für eine Instanz;
  method: Knoten entspricht einer Methode;
  field: Knoten entspricht einer Variablen.
- L_I = {inheritance, message, **this**, **super**, **new**, def, uses} ist die Menge aller *Kantenmarkierungen* mit der Bedeutung:

  inheritance: Bindungskante.
  message: dynamisch gebundener Aufruf;
  this: reflexiver dynamisch gebundener Aufruf;
  super: statisch gebundener Aufruf;
  new: Instanzierung eines neuen Objekts;
  def: modifizierender Zugriff auf eine Variable;
  uses: lesender Zugriff auf eine Variable.
- $E \subseteq V \times L_I \times V$ ist die Menge aller direkten Aufruf- und Dateninteraktionen vereinigt mit der Menge aller zwischen Methoden bzw. Variablen (Features) bestehenden Vererbungsbeziehungen.
- Die Relation $P \subseteq E \times E$ berücksichtigt dynamisch gebundenen Methodenaufrufe bzw. Variablenzugriffe. ❑

Es folgen noch einige Erläuterungen zu Definition 3.1. Die Menge $E_V \subseteq E$, $x \in E_V \Leftrightarrow$ x=(m, inheritance, n) enthält die *Bindungskanten* des Klassen-Botschaftsdiagramms; es ist (m, inheritance, n) $\in E_V$, wenn Operation m die Operation n redefiniert und zusätzlich gilt

$\forall$ (**b**::m, inheritance, **a**::m) $\in E_V$: $\forall$ **k** $\in$ **K**:
b < **k** < **a** $\Rightarrow \neg$(m$\in$ **k**.operations).

E_V ist irreflexiv, asymmetrisch und injektiv, d.h., eine Operation redefiniert im Rahmen einer Vererbungshierarchie höchstens eine andere Operation. Für j $\in$ {message, **this**, **super**, **new**} bezeichnet $E_I \subseteq E$, $x \in E_I \Leftrightarrow$ x=(m, j, n) die Menge der *Aufrufinteraktionskanten* des Klassen-Botschaftsdiagramms. Es ist (m, j, n) $\in E_I$, wenn in Methode m ein Aufruf der Methode n erfolgen kann. Für j $\in$ {def, uses} bezeichnet $E_D \subseteq E$, $x \in E_D \Leftrightarrow$ x=(m, j, n) die Menge der *Dateninteraktionskanten* des Klassen-Botschaftsdiagramms. Es ist (m, j, n) $\in E_D$, wenn Variable m innerhalb der Methode n definiert wird bzw. in Methode m eine Benutzung der Variablen n erfolgen kann. Für j = inheritance gilt (m, j, n) $\in E_I$, wenn Methode m Schnittstelle und/oder Implementation von Methode n erbt bzw. die Definition der Variablen m von der Definition der Variablen n überschrieben wird. E_D, E_I und E_V partitionieren E. E_V ist irreflexiv und asymmetrisch. Bei einfacher Vererbung ist E_V darüber hinaus auch injektiv, d.h., ein Feature redefiniert höchstens ein anderes Feature.

Stehen Kanten zueinander in der Relation P, so wird abhängig von der tatsächlichen Klasse des Zielobjekts, durch den Aufruf genau eine der den Zielknoten entsprechenden Methoden ausgeführt bzw. auf die entsprechend definierte Variable zugegriffen. P ist reflexiv und transitiv.

Wir erkennen die direkten Aufrufinteraktionen anhand der Kanten des KBD. Zustandsbedingte Abhängigkeiten zwischen den Methoden (indirekte Interaktionen) spiegeln sich im KBD in Pfaden von einem Methodenknoten zu einem (nicht notwendigerweise verschiedenen) Methodenknoten wider, die mindestens eine[1] mit def oder mit uses markierte Kante beinhalten.

Fassen wir zusammen: Das Klassen-Botschaftsdiagramm stellt mit jedem Knoten eine Methode oder eine Variable dar. Die Kanten stellen entweder direkte Aufruf- bzw. Dateninteraktionen oder aber die Redefinition von Methoden innerhalb von Generalisierungshierarchien dar. Im Gegensatz zu den im ersten Abschnitt vorgestellten heterogenen Modellen von McGregor et al. [14] und Kung et al. [13] bildet das Klassen-Botschaftsdiagramm somit einen bzgl. der Interaktionen passend abstrahierenden, unifizierenden Schnitt durch das Klassendiagramm und die Kontroll- und Datenflussgraphen.

4 ANWENDUNGEN

Gemäß unseren Ausführungen in Abschnitt 2 betrachten wir den Integrations- und den Regressionstest gemeinsam. Ziel ist es, die Anzahl der erforderlichen Testtreiber und -stellvertreter sowie die der erneut auszuführenden Testfälle einzugrenzen. Unser Verfahren gliedert sich in die drei Schritte

- Änderungsanalyse,
- Reihenfolgeermittlung und
- Testfallauswahl.

1. Da def- bzw. uses- Kanten immer vom Variablen- zum Methodenknoten bzw. umgekehrt gerichtet sind, enthalten die indirekten Interaktionen entsprechenden Pfade immer gleich viele, jeweils aufeinanderfolgende def- und uses- Kanten.

4.1 Änderungsanalyse

Die Menge der in einer Iteration geänderten, hinzugefügten oder gelöschten Methoden und Variablen erhalten wir aus dem Konfigurationsmanagement oder durch Vergleich der KBD's vor und nach der Iteration. Die darauf folgende Änderungsanalyse erfolgt über das KBD der geänderten Version, wobei die Bindungskanten ignoriert werden können, da sie im Rahmen der Änderungsanalyse von den polymorphen Botschaftskanten überdeckt werden:

- Zunächst werden alle geänderten und neuen Knoten im KBD markiert. Sie bilden die initiale Regressionsmenge.
- Dann werden im transponierten und als schlichten Graph interpretierten KBD über eine Tiefensuche transitiv alle von den markierten Knoten erreichbaren Knoten ermittelt und der Regressionsmenge hinzugefügt.

Jeder so ermittelte Knoten der Regressionsmenge ist von mindestens einer Änderung betroffen, sodass die entsprechende Methode bzw. Datenverwendung erneut getestet werden muss.

4.2 Reihenfolgeermittlung

Nun ermitteln wir — wiederum auf der Basis des KBD [25] — eine Testreihenfolge auf der Granularitätsebene von Methoden und Variablen. Kurz gesagt leiten wir aus dem KBD eine partielle Ordnung für die Methoden ab und lösen dabei ggf. durch zyklische Aufrufbeziehungen entstehende Konflikte auf. Die Berechnung erfolgt ähnlich zu [13] in zwei Schritten, in denen das vollständige KBD (mit Bindungskanten) verwendet wird:

- Knoten in einer starken Zusammenhangskomponente werden zu einem künstlichen Knoten (Clusterknoten) komprimiert.
- Der resultierende zyklenfreien Graph wird topologisch sortiert.

Anhand der sich ergebenden partiellen Ordnung der Methoden- bzw. Clusterknoten wird dann bottom-up integriert. Methoden, deren Knoten bzgl. der partiellen Ordnung äquivalent sind, können in beliebiger Reihenfolge getestet und integriert werden, wobei wir empfehlen, mit den Regressionstests für unveränderte Methoden zu beginnen.

Der Vorteil unseres Vorgehens ist, dass viele der auf Klassen- und Assoziationsebene zu beobachtenden Zyklen auf der Ebene von Methoden und Variablen zerfallen. Zum Aufbrechen evtl. verbleibender Zyklen sind lediglich Teststellvertreter für einzelne Methoden zu erstellen, die meistens unmittelbar erkenntlich sind.

Beispiel 4.1 Als Beispiel zur Ermittlung einer Testreihenfolge betrachten wir das Entwurfsmuster „Mediator“ bzw. die konkrete “Musterinstanz“ DialogDirector in Abb. 8. Insbesondere weisen wir auf die zyklische Abhängigkeit zwischen den Klassen **FontDialogDirector** und **ListBox** bzw. **Field** hin, die über die Assoziationen und die Vererbungsbeziehungen induziert werden.

Das entsprechende Klassen-Botschaftsdiagramm zeigt Abb. 9. Wie zu erkennen ist, wird die im Klassendiagramm zu beobachtende zyklische Abhängigkeit durch die feinere Granularität im Klassen-Botschaftsdiagramm aufgelöst.

Die resultierende Integrationsstrategie ist in Abb. 10 dargestellt. Nach dem Test des Konstruktors der Klasse **Widget** können die Konstruktoren der Klassen **Field** und **Listbox** in beliebiger Reihenfolge integriert werden. Danach werden nach dem Test des Konstruktors der Klasse **DialogDirector** die Operationen **Field**::SetText(), **ListBox**::GetSelection() und **DialogDirector**::WidgetChanged() in beliebiger Reihenfolge integriert etc.

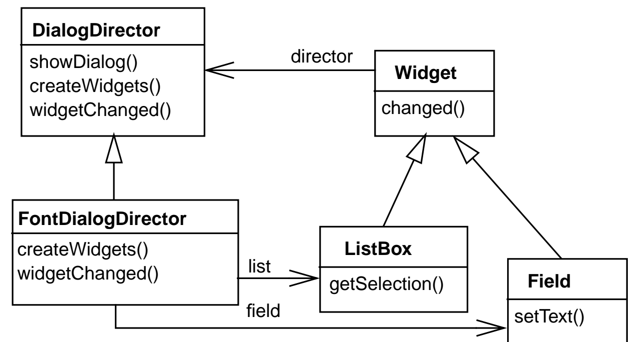

Abb. 8 Mediator: DialogDirector

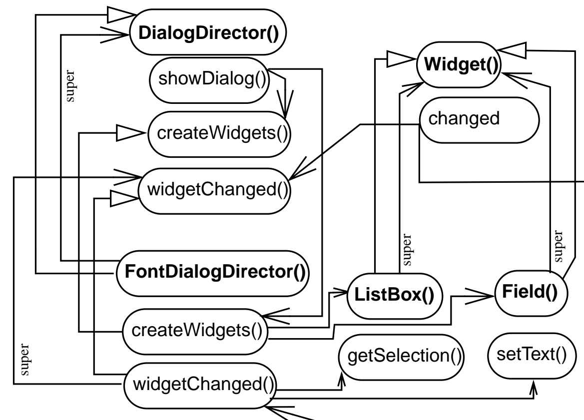

Abb. 9 DialogDirektor: Klassen-Botschaftsdiagramm

Widget(), {**Field**(), **ListBox**()}, **DialogDirector**(),
{**Field**::SetText(), **ListBox**::GetSelection(),
    DialogDirector::WidgetChanged(),
    FontDialogDirector() },
DialogDirector::CreateWidgets(),
FontDialogDirector::WidgetChanged(),
{**FontDialogDirector**::CreateWidgets(),
    Widget::Changed(),
    DialogDirector::ShowDialog() }

Abb. 10 DialogDirektor: Integrationsstrategie

4.3 Testfallauswahl

Zur Auswahl der erneut auszuführenden Testfälle skizzieren wir zwei Möglichkeiten. In beiden Fällen müssen zunächst die aufgrund von Änderungen z.B. der Spezifikation oder der Signatur einer Operation nicht mehr verwendbaren Testfälle ausgesiebt werden ([22]).

Sind die Testfälle z.B. durch Überdeckungsprotokolle zu dem durch sie ausgeführten Code verfolgbar, so können sie den geänderten bzw. von der Änderung betroffenen Methoden sofort zugeordnet werden ([22]). Hierbei reicht eine Überdeckungsmessung auf Methodenebene aus. Problematisch ist, dass sich aufgrund von Änderungen auch Ablaufpfade und damit die Überdeckungseigenschaft der Testfälle geändert haben können, was die Sicherheit des Verfahrens beeinträchtigt. Rothermel und Harrold behandeln diese Problematik durch nachgeschaltete Kontroll- und Datenflussanalysen ([22]).

Sind die Testfälle jedoch in Form von Programmcode z.B. in speziellen Testklassen oder Testmethoden festgehalten, können sie in unserem Ansatz bei der Generierung bzw. Aktualisierung des KBD direkt berücksichtigt werden. In dem für den Test- und Produktionscode erstellten (transponierten) KBD sind hierzu lediglich die von den in Abschnitt 4.1 als von der Änderung betroffen markierten Knoten erreichbaren „Testfallknoten" zu ermitteln. Da so alle Änderungen berücksichtigt werden, ergibt sich (bei gleichbleibender Sicherheit) eine höhere Selektivität der Testfallauswahl und somit eine Senkung der Testkosten.

Im Einzelfall ist abzuwägen, ob sich der „trade-off" durch die Einsparung bei der Testausführung und die Kosten der Testfallauswahl rechnet.

4.4 Testprozess

Es ergibt sich der in Abb. 11 skizzierte kombinierte Prozess für den Regressions- und Integrationstest. Die mit (*) markierten Schritte werden durch das KBD direkt unterstützt.

```
FOR jedes Inkrement DO
BEGIN
1     Identifiziere Änderungen;
2     Änderungsanalyse; (*)
3     (Re-) Test Strategie generieren; (*)
4     Funktionale Testfälle auswählen/generieren;
5     Funktionale Testfälle ausführen;
6     UNTIL Überdeckung erreicht DO
7     BEGIN
8           Strukturelle Testfälle auswählen (*) / generieren;
9           Strukturelle Testfälle ausführen;
10    END
END
```

Abb. 11 Integrations- und Regressionstestprozess

Die in den Zeilen 4 und 5 auszuwählenden bzw. auszuführenden funktionalen Tests, bei denen das Anwendungssystem gegen die Anforderungsspezifikation geprüft wird, können durch den sogenannten „SCORES grey-box Systemtest" einbezogen werden. Dieser basiert auf Anwendungsfällen [4], die mit sogenannten Schrittgraphen (geeignet stereotypisierte bzw. erweiterte Aktivitätsdiagramme) sowie mit Interaktionsdiagrammen an das Klassenmodell gekoppelt werden [26]. Hierbei wird mit dem KBD der Übergang von der Anforderungsspezifikation in die in Java programmierte Implementation realisiert.

5 EXPERIMENTE

Mit einem exemplarisch für Programme in Smalltalk-80 implementierten Testwerkzeug wurden Experimente durchgeführt [16]. Insbesondere wurden die in den generierten Klassendiagrammen (KD) bzw. den entsprechenden KBD's auftretenden starken Zusammenhangskomponenten bzgl. Anzahl und Größe verglichen[1]. Diese stellen ein direktes Maß für die Anzahl erforderlicher Testtreiber und Teststellvertreter dar, da jede zyklische Abhängigkeit für den Integrationstest aufzubrechen ist [18].

Tabelle 2 zeigt die Ergebnisse für einen Ausschnitt der Behälterklassen in der Smalltalk-80-Klassenbibliothek. Die erste Zeile besagt, dass 30 Klassen mit 426 Methoden untersucht wurden. Die zweite Zeile gibt an, dass 395 Aufrufe (aus max. 870 möglichen) zwischen Klassen, und 3325 (aus 181,050 möglichen) Aufrufe zwischen Methoden erkannt wurden. Der maximale Eingangsgrad einer Klasse, d.h. die Anzahl direkt von dieser

	KD	KBD
Knoten	30	426
Kanten	395	3325
Max. Eingangsgrad der Knoten	30	335
Anzahl. Zus. Komponenten	1	3
Klassen in Zus. Komponenten	26	22
Methoden in Zus. Komponenten	424	157

Tab. 2. Experiment: Smalltalk-80-Behälterklassen

Klasse abhängiger Klassen, ist 30. Die maximale Anzahl von Aufrufen einer Methode ist 335 (!). Die letzten drei Zeilen geben eine aus 26 Klassen mit insgesamt 424 Methoden bestehende starke Zusammenhangskomponente an. Diese zerfällt auf der Methodenebene in drei starke Zusammenhangskomponenten, die noch 22 Klassen bzw. 157 involvierte Methoden enthalten. Dies bedeutet eine Eingrenzung der zum Aufbrechen der Zusammenhangskomponente(n) zu untersuchenden Methoden um über 50%. Auch auf der Methodenebene ist der Test aufgrund der überaus starken Kopplung erschwert, die durch die tiefe Vererbungshierarchie induziert wird.

Ein weiteres Experiment mit 6 Klassen und 30 Methoden ist in Tabelle 3 zusammengefasst. In diesem Fall ist die Vererbungshierarchie eher flach und die aus zwei Klassen bestehende starke Zusammenhangskomponente zerfällt auf der Methodenebene vollständig. Dies bedeutet, das die Integration vollständig auf Testtreiber und -stellvertreter verzichten kann.

	KD	KBD
Knoten	6	30
Kanten	12	36
Max. Eingangsgrad der Knoten	5	4
Anzahl. Zus. Komponenten	1	0
Klassen in Zus. Komponenten	2	0
Methoden in Zus. Komponenten	19	0

Tab. 3. Experiment: Verkaufsautomat

Insgesamt ist zu erkennen, das durch die direkte Betrachtung der Methoden und Methodenaufrufe bzw. Variablenzugriffe im KBD z.T. deutliche Einsparungen bei dem Testaufwand erzielt werden können.

6 ZUSAMMENFASSUNG UND AUSBLICK

In diesem Beitrag haben wir das Klassen-Botschaftsdiagramm als ein die Struktur und das Verhalten objektorientierter Programme gleichermaßen berücksichtigendes interaktionsbasiertes Testmodell vorgestellt und einen Algorithmus sowohl zur Ableitung einer Integrationsstrategie als auch zur Auswahl von Regressionstestfällen nach der Modifikation bestehender Klassen skizziert. Abschließend haben wir die Ergebnisse einiger mit einem Testwerkzeug für Klassen der Programmiersprache Smalltalk-80 durchgeführter Experimente zur Evaluierung des Verfahrens umrissen.

Wir experimentieren zur Zeit mit unserer prototypischen Implementation des Klassen-Botschaftsdiagramms für die Programmiersprache Java. Die ersten Ergebnisse sind aufgrund der statischen Typisierung in Java sehr ermutigend und wir planen den Ausbau des Prototyps zu einer integrierten Testumgebung. In diesem Bereich komplettieren wir eine Komponente zur Unter-

1. Die ermittelten Werte bilden aufgrund der Typfreiheit von Smalltalk-80 obere Grenzen für diese Problematik.

stützung des „grey-box“ Systemtests nach SCORES [26].

Danksagung

Mein Dank gilt den Kollegen des TAV-Arbeitskreises „Testen objektorientierter Programme“ sowie den anonymen Gutachtern für ihre hilfreichen Kommentare zu diesem Beitrag.

7 LITERATUR

[1] Robert Binder *Testing Object-Oriented Software — A Survey* Software Testing, Verification, and Realiability Vol. 6, 1996, S. 125-252

[2] Günther Blaschek, Joachim Hans Fröhlich *Recursion in Object-Oriented Programs* Journal of Object-Oriented Programming, Nov./Dez. 1998, S. 28-35

[3] Shawn A. Bohner, Robert S. Arnold *Software Change Impact Analysis* IEEE Press, Los Alamitos, 1996

[4] Grady Booch, Ivar Jacobson, James Rumbaugh UML Document Set V1.3, 1999, http://www.rational.com/uml

[5] Michael A. Cusumano, Richard W. Selby *Microsoft Secrets* The Free Press, New York, 1995

[6] Michael S. Deutsch *Software Verification and Validation - Realistic Project Approaches* Prentice Hall, Englewood Cliffs, 1982

[7] Erich Gamma, Richard Helm, Ralph E. Johnson, John Vlissides (the Gang of Four) *Design Patterns - Elements of Object-Oriented Reusable Software* Addison-Wesley, Reading, Massachusetts, 1994

[8] J. Gosling, B. Joy, G. Steele *The Java Language Specification* Addison Wesley, Reading, Mass., 1996

[9] Mary Jean Harrold, John D. McGregor, Kevin J. Fitzpatrick *Incremental Testing of Object-Oriented Class Structures* Proc. International Conference on Software Engineering, ACM, Mai 1992, S. 68-80

[10] Brigid Haworth, Mark Roper *Towards the Development of Adequacy Criteria for Object-Oriented Systems* Proc. EuroSTAR‘97, Edinburgh, UK, 1997

[11] Paul C. Jorgensen, Carl Erickson *Object-Oriented Integration Testing* CACM Vol. 37, No. 12, Sep. 1994, S. 30-38

[12] Philippe Kruchten *A Rational Development Process* CrossTalk, 9 (7) July 1996, S.11-16

[13] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen *On regression testing of object-oriented programs* Journal of Systems and Software, Vol. 32, No. 1, Jan. 1996, S. 21 - 40

[14] John D. McGregor, Brian A. Malloy, Rebecca L. Siegmund *A Comprehensive Program Representation of Object-Oriented Software* Annals of Software Engineering, Jahrg. 2, 1996, S. 51-91

[15] Bertrand Meyer *Object-Oriented Software Construction* Prentice Hall, Hemel Hempstead 1997

[16] Gabriele Mohl *Regressionstesten objektorientierter Software* Diplomarbeit, FernUniversität Hagen, Aug. 1997

[17] Peter Müller, Arnd Poetzsch-Heffter *Kapselung und Methodenbindung: Javas Designprobleme und ihre Korrektur* in: C. H. Cap. (Ed.): JIT '98 Java-Informations-Tage 1998, Informatik Aktuell, Springer-Verlag, 1998

[18] Jan Overbeck *Integration Testing for Object-Oriented Software* Dissertation, TU Wien, 1994

[19] Bernd-Uwe Pagel, Hans-Wernder Six *Software Engineering I* Addison Wesley, Bonn, 1994

[20] Dewayne E. Perry, Gail E. Kaiser *Adequate Testing and Object-Oriented Programming* JOOP Vol. 3, No. 2, January/ February 1990, S. 13-19

[21] Eike Hagen Riedemann *Testmethoden für sequentielle und nebenläufige Software-Systeme* B.G. Teubner, Stuttgart, 1997

[22] Gregg Rothermel, Mary Jean Harrold *Selecting Regression Tests for Object-Oriented Software* Technischer Report TR-94-111, Clemson University, 1994

[23] Harry M. Sneed *Objektorientiertes Testen* Informatik Spektrum, Vol. 18, Nr. 1, Feb. 1995, S. 6-12

[24] Andreas Spillner *Dynamischer Integrationstest modularer Softwaresysteme* Dissertation, Universität Bremen, 1990

[25] Mario Winter *Managing Object-Oriented Integration and Regression Testing* Proc. 6th EuroSTAR 98, München, Dez. 1998, S. 189-200

[26] Mario Winter *Qualitätssicherung für objektorientierte Software: Anforderungsermittlung und Test gegen die Anforderungsspezifikation*, Verlag dissertation.de, Berlin, 2000, zugl. Dissertation, FernUniversität Hagen, Sept. 1999